DISSERTATION

The Time-Triggered System-on-Chip Architecture

ausgeführt zum Zwecke der Erlangung des
akademischen Grades eines

Doktors der technischen Wissenschaften

unter der Leitung von

o. Univ. - Prof. Dr. Hermann Kopetz
Institut für Technische Informatik 182/1

eingereicht an der

Technischen Universität Wien,
Fakultät für Informatik

durch

Dipl. - Ing. Christian Peter Paukovits, Bakk. techn.
Matr. - Nr. 0127145
Linzer Straße 429 / 5 / 5204, A–1140 Wien

Wien, im Dezember 2008 .....

Markenzeichen und ähnliche geschützte Begriffe sind teilweise nicht speziell gekennzeichnet, und es liegt daher in der Verantwortung des Lesers, diese im Zweifelsfalle zu überprüfen. Ich sehe es nicht als meine Aufgabe als Wissenschaftler, Markennamen zu recherchieren.

Trademarks and similar terms protected by law are partially not particularly denoted in this thesis. If in doubt, it is in the responsibility of the reader to verify the terms used. It is not my task as a scientist to investigate trademarks.

The Time-Triggered System-on-Chip Architecture

The Time-Triggered System-on-Chip (TTSoC) architecture provides a component-based design methodology, which addresses complexity management of System-on-Chip designs equipped with billions of transistors. Abstraction, determinism, and encapsulation are the means to achieve a consequent decoupling of computational components from the communication infrastructure, which entails error containment and promotes composability.

This thesis presents a real implementation of the TTSoC architecture based on FPGA technology. Intellectual Property (IP)-cores contain the processing units, in which jobs of application subsystems are executed. Each IP-core has one Trusted Interface Subsystem (TISS) attached, which realizes the major part of the TTSoC architecture's core services such as the time-triggered communication service. The TISS offers these core services to the IP-cores through the Uniform Network Interface (UNI). Each pair of IP-core and attached TISS forms a micro component, which is regarded as architectural unit. Micro components are interconnected through the Time-Triggered Network-on-Chip (TTNoC).

Micro components communicate with each other by means of encapsulated communication channels. Their endpoints – the ports – contain application-level messages, which are exchanged between micro components. The real communication within encapsulated communication channels is abstracted from the IP-core. Communication is synchronized by means of a notion of a global time base, which entails a periodic control system in order to temporally align communication. The TISS harnesses the global time base in order to determine the instants, when a communication activity takes place. Additionally, it controls the flow of messages from send port to receive port of an encapsulated communication channel.

On-the-fly reconfiguration allows the TTSoC architecture to change system parameters during live operation, which is used to adapt the system to changing resource demands or environmental conditions.

To let a given IP-core take part in the TTSoC architecture, a TISS causes a hardware overhead of 10 % and below on FPGA technology. The monetary costs on ASIC technology can be estimated in the magnitude of fractions of one-digit dollar cent.

Die Time-Triggered System-on-Chip Architektur

Die Time-Triggered System-on-Chip (TTSoC) Architektur ermöglicht eine komponentenbasierte Designmethodologie, welche Komplexitätsmanagement von System-on-Chip Designs mit Milliarden von Transistoren anspricht. Abstraktion, Determinismus und Kapselung sind die Mittel zur konsequenten Entkoppelung von Rechnerkomponenten und Kommunikationsinfrastruktur, wodurch Fehlerisolation und Kompositionalität erzielt wird.

Diese Dissertation präsentiert eine auf FPGA Technologie basierende, reale Implementierung der TTSoC Architektur. Intellectual Property (IP)-cores enthalten die Rechnereinheiten, in denen Jobs von Applikationssubsystemen exekutiert werden. Jeder IP-core hat ein Trusted Interface Subsystem (TISS) angeschlossen, das den Großteil der Systemdienste wie den zeitgesteuerten Kommunikationsdienst der TTSoC Architektur realisiert. Das TISS bietet diese Systemdienste über das Uniform Network Interface (UNI) den IP-cores an. Jedes Paar von IP-core und TISS formt eine Mikrokomponente, die als Architektureinheit betrachtet wird. Mikrokomponenten sind durch das Time-Triggered Network-on-Chip (TTNoC) verbunden.

Mikrokomponenten kommunizieren miteinander mittels gekapselter Kommunikationskanäle. Deren Enden – die Ports – enthalten Nachrichten auf Applikationsebene, die zwischen Mikrokomponenten ausgetauscht werden. Die tatsächliche Kommunikation innerhalb der gekapselten Kommunikationskanäle wird für die IP-cores abstrahiert. Kommunikation wird anhand einer globalen Zeitbasis synchronisiert, die ein periodisches Kontrollsystem mitbringt, um Kommunikation zeitlich auszurichten. Das TISS nutzt die globale Zeitbasis zur Bestimmung der Zeitpunkte, wann Kommunikationsaktivitäten stattfinden. Zusätzlich kontrolliert es den Fluss von Nachrichten zwischen Sende- und Empfangsports eines gekapselten Kommunikationskanals.

Fliegende Rekonfiguration erlaubt der TTSoC Architektur, Systemparameter im laufenden Betrieb zu ändern, was zur Anpassung des Systems an wechselnde Ressourcenanforderungen oder Umweltbedingungen benutzt wird.

Zur Befähigung zur Teilnahme an der TTSoC Architektur für einen IP-core verursacht ein TISS einen Hardwareaufschlag von höchstens 10 % in FPGA Technologie. Die Kosten in ASIC Technologie können in die Größenordnung von Bruchteilen einstelliger Dollarcent Beträge geschätzt werden.

Acknowledgments

This work was conducted during my affiliation as a research assistant with the Institut für Technische Informatik, Technische Universität Wien.

I address my gratitude to the advisor of this thesis, Prof. Dr. Hermann Kopetz, who has enabled this affiliation. He has been supporting me with valuable suggestions and formed my academic career. In this context, I would like to mention Dr. Wilfried Elmenreich for his engagement as consultor for this work.

Furthermore, I would like to thank the fellows at the Institut für Technische Informatik, Technische Universität Wien for the countless exhilarating discussions on all kinds of topics. In this context, I would like to mention[1] Sven Bünte, Bernhard Frömel, Albrecht Kadlec, and Michael Zolda.

Special thanks to Christian El Salloum for lots of constructive discussions about the design and implementation of the Time-Triggered System-on-Chip architecture.

Finally, I give my appreciation to Maria Ochsenreiter for her engagement in organisation and her support to come to grips with my affiliation at the Institut für Technische Informatik. Thanks to Leo Mayerhofer for his technical aid during the past two years.

— *Christian Paukovits*
December 2008

[1] in alphabetical order

Danksagung

Diese Arbeit entstand im Rahmen meiner Forschungstätigkeit am Institut für Technische Informatik, Abteilung Echtzeitsystem, an der Technischen Universität Wien.

Besonderen Dank richte ich an den Betreuer dieser Dissertation, Prof. Dr. Hermann Kopetz, der mir die Forschungstätigkeit am Institut für Technische Informatik ermöglicht hat. Er hat mich stets durch wertvolle Anregungen unterstützt und so meinen wissenschaftlichen Werdegang geprägt. In diesem Sinne möchte ich Dr. Wilfried Elmenreich für sein Engagement als Berater für diese Arbeit erwähnen.

Außerdem bedanke ich mich bei den Kollegen des Instituts für Technische Informatik, Technische Universität Wien für die zahllosen anregenden Diskussionen zu verschiedensten Themen. In diesem Sinne möchte ich erwähnen[2]: Sven Bünte, Bernhard Frömel, Albrecht Kadlec und Michael Zolda.

Mein besonderer Dank geht an Christian El Salloum für die Vielzahl an konstruktiven Diskussionen zu Design und Implementierung der Time-Triggered System-on-Chip Architektur.

Schließlich drücke ich meine Anerkennung für Maria Ochsenreiter für ihr Engagement in organisatorischen Belangen und ihre Hilfestellung, um mich am Institut für Technische Informatik zurecht zu finden, aus. Danke auch an Leo Mayerhofer für die technische Unterstützung während der letzten zwei Jahre.

— *Christian Paukovits*
Dezember 2008

[2] in alphabetischer Reihenfolge

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the past forty years, the semiconductor industry has developed chips of enormous complexity. The number of transistors per chip has increased so much that we are beyond a billion. This has enabled designers to pack several functional units onto one silicon die. The inherent miniaturization has led to better yield of wafers, thus reducing costs in production. Inspired by these improvements, embedded computing has taken the role of the most important segment of the computer industry.

Nevertheless, the increased complexity has pointed out cognitive limits of human designers. Nowadays, chip designs are so vast that they can hardly be understood. According to the 2005 semiconductor industry roadmap [Wil05], designers will face system design complexity and designer's productivity as key challenges on the way to the giga-scale System-on-Chip (SoC). This challenge can only be tackled if we lift the design process to a higher level of abstraction [Kop08a].

Even though the semiconductor industry has made considerable progress, the basic computational model has not changed significantly during this time. Pollack's rule [Gel01] says that the increase in performance of a sequential computer is only about the square root compared to the increase in the number of transistors. This implies that doubling the transistor count leads to a performance gain of about 40%.

Fortunately, the inherent concurrency in a typical embedded application (such as automotive electronics, avionics) offers the possibility to circumvent Pollack's rule. If an application can be partitioned into a set of nearly autonomous concurrent functions, then a nearly linear performance improvement can be achieved. The trick is to assign a dedicated processing element to each of these concurrent functions. Additionally, each dedicated processing element interacts via an Network-on-Chip (NoC) [Wol04] with its communication partners. This architectural approach is followed in numerous SoC architectures [BM06].

To address these challenges we introduce a novel system architecture: the *Time-Triggered System-on-Chip* architecture. This architecture offers a component-based design methodology, which is intended to take on the complexity of SoCs equipped with billions of transistors. The means to achieve this goal is the consequent decoupling of the computational components from the communication infrastructure.

The Time-Triggered System-on-Chip (TTSoC) architecture provides an architectural framework that supports *composability* [KO02, Sif05]. Composability denotes the side-effect-free composition of component services – solely based on interface specifications – to form larger systems-of-systems. For this purpose, the computational components are interconnected through a predictable and deterministic NoC with inherent error containment.

1.1 Motivation

The Time-Triggered Architecture (TTA) [KB03] has established a platform for distributed embedded (hard) real-time system [TS06]. It provides deterministic communication based on a notion of global time with synchronized clocks, endorses composability in application design, and contributes to fault tolerance.

In recent years the TTA has spawned industrially applied products that are based on its concepts.

TTP/C is a time-triggered communication protocol that focuses on safety-critical distributed real-time systems. Its intended domain of application are automotive control systems, aircraft control systems, industrial and power plants, or air traffic control.

TTP/A is a field bus targeted at low-cost distributed embedded systems. Its main usage is in the field of micro controllers, where it is intended to connect sensors and actuators.

Time-Triggered Ethernet (TTE) aims at the same application domains like TTP/C, however it uses the widely spread communication technology *Ethernet* [Gro05] for transmission. Time-Triggered Ethernet (TTE) integrates the time-triggered critical traffic and non-critical, event-driven legacy traffic, so that it is backwards compatible to Ethernet.

With the advent of Deep Submicron (DSM) technology in consumer mass markets, the idea of the SoC has gained vast popularity. Nowadays, we have the technological foundation to miniaturize the main concept of TTA to a "microscopic" dimension. In short, this is what the TTSoC architecture does. It is the fourth member of the TTA family, which consequently follows the time-triggered paradigm.

Certainly, the TTSoC architecture reflects the continuous evolution of the TTA. Even though the TTSoC architecture possesses the same basic capabilities as the original descendants of the TTA, it adds new concepts that give more mightiness to design applications in several domains.

One of the main objectives of the TTSoC architecture is abstraction between application design and underlying communication issues. Such abstraction is the means

to promote composability on an architectural level. In addition to this, encapsulation of communication channels is introduced to provide error containment. Error containment is a strong assumption and required to uphold composability. Finally, the TTSoC architecture is one of the first SoC architectures to include an integrated resource management, so that the system is able to adapt to changing conditions or application requirements in the field during live operation.

Another motivation is a cost argumentation. According to the 2007 semiconductor roadmap [Wil07a], the costs of designing ASICs, e.g., costs of lithographic masks, costs of verification of designs etc., will further increase in the near future. Consequently, it would be desirable to reuse a given design for multiple applications in different target domains. Hence, we need a "generic" system architecture, which fits the requirements of several target domains.

The TTSoC architecture is designed to satisfy the requirements of a batch of target domains from the field of distributed embedded (hard) real-time systems. Moreover, it allows to integrate subsystems of mixed criticality levels, such as control application with tough temporal constraints, or multimedia applications with vast bandwidth and performance requirements.

1.2 Contribution

This thesis concerns the implementation of the TTSoC architecture. It answers the research questions:

- Is it feasible to build such a generic architecture that goes with the requirements of several target applications?
- If so, how would an implementation look like, how would its components work together?
- Finally, what does it cost to equip a given system design with that generic architecture?

Earlier work on the TTSoC architecture has explained the fundamental concepts, design choices, and interface specifications from the theoretical point of view. Contrary, this thesis presents a real implementation of the TTSoC architecture.

The first contribution is that this thesis includes a comprehensive description of the architecture-level interface, through which the *core services* are available. This is the interface, where the abstraction of communication issues takes place.

In this context, the concept of communication in the TTSoC architecture is covered. The thesis describes how we can leverage the mechanisms of the communication subsystem to establish high-level concepts such as determinism, error containment, message ordering, and temporal alignment of communication.

Also, this thesis introduces the design of a NoC, which interconnects participants of communication. This NoC yields characteristics of communication in the TTSoC architecture, thus the realization of that NoC is efficient with respect to area and power. The thesis presents the basics of switching and advanced capabilities such as support for native multi-casting in that NoC.

A major contribution is the answer to the second research question. The thesis explains, how communication is established, which architectural components are in charge of controlling communication, and how these components are implemented. In this context, we find the documentation concerning the implementation of the core services and auxiliary services (e.g., watchdog service, dissemination of diagnosis information) of the TTSoC architecture.

Furthermore, this thesis demonstrates how the TTSoC architecture handles integrated resource management, and how a reconfiguration of system parameters is achieved during live operation.

Finally, this thesis comprises results of working prototypes of the implementation, which allow to draw conclusions about the feasibility and costs of the TTSoC architecture. We state quantitative values about resource usage of the implementation on FPGA technology, and compare these values with other system designs. Also, this thesis includes a cost estimation for production of the implementation on ASIC technology.

1.3 Structure of this Thesis

Chapter 2 gives background information about the TTSoC architecture. It explains the architectural concepts and fundamental properties. Further, it introduces the terminology used by the TTSoC architecture. Additionally, this introduction chapter sketches the tasks of architectural entities and illustrates an overview of the implementation of the TTSoC architecture.

Chapter 3 compares the TTSoC architecture with other approaches that have gained relevance in the research community.

The following chapter 4 is dedicated to the communication service of the TTSoC architecture. It mentions communication primitives and demonstrates, how the notion of a global time is exploited in the TTSoC architecture.

The architectural interface to the core services is covered in chapter 5, while the NoC of the TTSoC architecture is outlined in chapter 6.

Chapter 7 deals with the most important architectural component of the TTSoC architecture. It explains its structure, memories and subcomponents, and how these entities work together to realize the biggest part of the core services.

The integrated resource management and the reconfiguration during live operation are subject of chapter 8.

In chapter 9 the results of the prototypic implementation of the TTSoC architecture are listed. Moreover, this chapter features the hardware set-up, on which the prototypes have been built. Also, a design example of what the TTSoC architecture could look like in a real application scenario is presented.

The last chapter 10 discusses properties of the implementation of the TTSoC architecture, and draws conclusions based on these insights. In this context, it justifies design choices and outlines their optimality. Also, it gives an outlook of future work on the TTSoC architecture.

Chapter 2

Basic Concepts of the TTSoC Architecture

This chapter reveals the contributions and novelties of the TTSoC architecture. Also we show, how these features are organized in conceptual entities, and how these entities realize the TTSoC architecture in a prototypic implementation.

2.1 Features and Key Properties

The TTSoC architecture is equipped with a batch of features and key properties. From these features we can derive a set of core services. Such core services provide an interface to applications built around the TTSoC architecture, which abstracts from the real implementation beneath that interface.

2.1.1 Elevation of the Level of Design Abstractions

We promote the idea that complexity of an evolving design must be managed at a higher level of abstraction [Kop08a]. For this purpose, the aggregate properties of a component must be outlined into an interface specification. We must be able to stick components together by simply examining their interface specifications. Thus, the internal structure and the operations within the component can be neglected. Furthermore, it is possible to change or enhance the implementation of the components, for instance in response to technological improvements. As the implementation issues of the components are hidden behind the interface, no redesign of the overall system is necessary.

For this reason, we introduce the entity of a *micro component* in the TTSoC architecture. By definition, such a micro component is a unit of abstraction. A well-defined message-based interface informs about its functionality [GIJ+03]. The clear distinction of processing within a micro component and the interactions with

its surrounding results into a communication-centric model [BM02, WG02], which is appropriate for many applications.

2.1.2 Determinism & Predictability

The TTSoC architecture entails determinism [Hoe04] with respect to the communication subsystem. Determinism denotes the property of a system (respectively its system model) to reason about the future behaviour (i.e., outputs) and the *state* [MT89] of the overall system based on the knowledge of an initial state and future inputs. We use the following (revised) definition of determinism:

> *A model of a distributed computer system (...) is said to behave deterministically if and only if, given a sparse time-base with an infinite sequence of intervals t_j, the state of the system $\Sigma(t_0)$ at t_0 (now), and a set of future Input Messages $IM_1(t_{i1}), IM_2(t_{i2}), \ldots, IM_n(t_{in})$, then the set of future Output Messages $OM_1(t_{o1}), OM_2(t_{o1}), \ldots, OM_n(t_{on})$ and the state of System $\Sigma(t_x)$ at all future intervals t_x is entailed.* [Kop08b]

The TTSoC architecture supports *a priori* defined on-chip communication expressed in a Time-Triggered Communication Schedule [KB03]. The existence of such a Time-Triggered Communication Schedule fulfils the requirements of determinism, as all future instants, senders and receivers, as well as content of communication activities are known at any time. From a given initial state $\Sigma(t_0)$ we can derive the state and future behaviour of the system at some later instant t' owing to the knowledge contained in the Time-Triggered Communication Schedule.

Such deterministic behaviour makes communication between micro components *predictable*. Besides this, a deterministic behaviour of components is required for the transparent masking of hardware errors by Triple Modular Redundancy (TMR) [Pol94]. Finally, Schütz argues in [Sch93] that determinism improves the testability of a system.

2.1.3 Error Containment through Encapsulation

The TTSoC architecture introduces a strong assumption of *encapsulation* of communication. That is, the communication of each micro component is protected from the communication of other micro components. As a result, communication activities are free of interference and collisions, in general. So, with the non-existence of any conflicts in mind, no micro component will ever disrupt the communication and computation of an opponent. Therefore, we say that communication of micro components is "encapsulated".

One effect of encapsulation is *error containment* [OKSH07]. The communication activities of each micro component are "invisible" to other micro components among the communication subsystem. A misbehaviour of a given micro component, e.g., a

babbling idiot [BB01], will not harm the operation of other micro components. Thus, error containment aids to uphold the composability of a system and increases the robustness against design flaws in subsystems.

Another effect of encapsulation is *complexity reduction*. With the concept of encapsulation (and error containment) we avoid dealing with the behaviour of interfering subsystems. In general, such an unpleasant scenario is more difficult to understand than to reason about the behaviour of clearly encapsulated subsystems. It is essential to avoid all system mechanisms that increase the cognitive load for understanding, if we mean to reduce the cognitive complexity of a design [FCS01]. Also, the test and validation effort for an encapsulated subsystem is smaller than the test effort for interfering subsystems [Owe04].

The TTSoC architecture obtains encapsulation by means of the concept of *encapsulated communication channels*. Firstly, encapsulated communication channels add bandwidth guarantees and bounded latency to the property of encapsulation. Secondly, they entail two guarantees with respect to *message ordering*: *total temporal ordering* and *consistent delivery order*. Particularly the latter is a vital prerequisite to establish system properties such as *replica determinism* [Pol93] or to apply distributed consensus and agreement algorithms [Lyn97].

2.1.4 Global Time Base & Clock Domains

In general, we can not assume that a state-of-the-art SoC comes up with a single clock signal for the entire chip. The reasons why designers introduce multiple clock domains embrace the handling of clock skew, the clocking down of individual Intellectual Property (IP)-blocks as part of power management, or the support for heterogeneous IP-blocks with different speeds (e.g., high-clocked special purpose hardware and a slower general purpose CPU).

The TTSoC architecture extends the approach of multiple clock domains. It does not solely bring in multiple clock domains on behalf of these technological issues, but also establishes a system-wide global time base. This global time base is a *sparse time base* [Kop92] and is generated through internal clock synchronization (i.e., within the SoC) and external clock synchronization (with respect to an external reference time).

The global time base yields in the *temporal coordination* of actions, e.g., communication activities, within the TTSoC architecture on each distributed micro component. Consequently, time stamps that have been noticed at different micro components can be related to each other. Additionally, time stamps are also meaningful outside the micro component where the event has been observed.

So, the TTSoC architecture features different clock domains by design:

- The global time base embodies an independent clock domain. The frequency associated with this clock domain determines the global granularity, to which actions in the system, e.g., communication activities, are synchronized.

- The communication subsystem itself is not driven by the global time base. Even though, the communication activities are synchronized by the global time base, the TTSoC architecture can harness two distinct arrangements of clock domains:
  1. The real communication (i.e., transport of data between micro components) can take place at a frequency with a granularity that is finer than the global granularity of the global time base. As a consequence, there is another system-wide clock domain present, which is independent from the global time base.
  2. Real communication involves different clock domains at each participant, which might also possess finer granularity than the global time base. In this case, the communication subsystem would employ an asynchronous handshake protocol across clock domain boundaries. Even though, there are several local clock domains involved, the communication activities are still synchronized by means of the common global time base.
- Micro components can include an arbitrary number of local clock domains, which are not visible outside the micro component's architectural borders. For instance, a micro component can be assembled by processor cores, memories, and IP-blocks, which run their own frequencies.

The existence of multiple clock domains, particularly of a global time base, entails the decoupling of synchronization of actions within the system and the operation of local entities. The global time base is allowed to maintain a relative slow clock domain compared to the remainder of the system. While the remainder of the system is driven by higher, local clock domains, there is only one slow clock domain that spans the whole system. This design feature helps designer to deal with clock skew and driver capacity issues of clock distribution networks in state-of-the-art chip designs, as the slow, system-wide clock domain is easy to handle, and the higher clock domains are local and independent from each other.

From the conceptual point of view, the decoupling owing to the global time base supports a further reduction of complexity. Application design focuses solely on communication activities, which are specified in the notion of the global time base. Therefore, application design and communication activities are decoupled from the implementation of the TTSoC architecture. Local modifications, for instance tuning frequencies in micro components and the communication subsystem, do not impose changes to the application design, as they are specified with respect to the independent global time base. Conversely, adapting the global time base has impact on the application design of course, but does not impair the implementation of the micro components.

2.1.5 Integrated Resource Management

The TTSoC architecture supports integrated resource management. Integrated resource management addresses requirements with respect to communication resources

(e.g., bandwidth, latency, latency jitter), computational resources (e.g., dynamic allocation of micro components to application subsystems), and power (e.g., power limiter).

Integrated resource management facilitates an efficient utilization of mutually exclusive resource demands, if it is *a priori* known that the worst-case resource consumption in different subsystems cannot occur simultaneously.

Furthermore, in case of a permanent fault affecting only individual micro components, integrated resource management can relocate the application functionality to spare micro components in order to preserve the specified service of the SoC. Additionally, it builds an important cornerstone for the realization of power-aware systems [UK03]. Power management is identified as one of the great challenges in the SIA's semiconductor road map [Wil05].

2.2 Core Services

With reference to the features and key properties mentioned in the section above, we define *core services* of the TTSoC architecture. The core services are:

communication service The TTSoC architecture inherits a time-triggered communication service from the TTA. All participants of communication follow a global, *a priori* defined communication schedule, which synchronizes communication activities by means of a common synchronized global time base. Therefore, the time-triggered communication service entails predictability of communication, and allows to impose bandwidth guarantees and bounded latency.

clock synchronization Communication is initiated by the progression of real time. Therefore, the TTSoC architecture establishes a global time base, which is replicated in each participant of communication. To achieve synchronous communication, this global time base must be kept within a given precision among all participants.

integrated resource management The TTSoC architecture realizes the properties of integrated resource management introduced in section 2.1.5.

diagnostic service The diagnostic service allows to monitor the wealth state of a system, i.e., a TTSoC system. In case of a deviation compared to specified behaviour, the diagnostic service is able to undertake measures to restore a defined operating condition. For this purpose, it takes usage of the communication service (plus clock synchronization service) and the integrated resource management.

The communication service and the clock synchronization service are intrinsically tied to each other. The communication service solely relies on the clock synchronization, while the clock synchronization solely serves the communication service.

Nevertheless, when we talk about the communication service, we implicitly also refer to the clock synchronization service.

The integrated resource management uses the communication service to coordinate the resources of the overall system and to achieve an *on-the-fly reconfiguration*, that is, the reconfiguration of parts of the overall system during run-time. In general, it is an additional feature of the TTSoC architecture. It could also exist on other architectures after adaption. Therefore, we regard it as stand-alone core service.

Note that the diagnostic service[1] is no core service directly, as it requires "real" core services to be realized.

2.3 Architecture Overview

This section introduces the conceptual entities, which make up the TTSoC architecture. Moreover, we sketch their functionality and their contribution to the overall TTSoC architecture.

2.3.1 Entities

Figure 2.1 gives the structure of the TTSoC architecture and illustrates, how the single conceptual entities form an overall system.

The central element of the TTSoC architecture is the *Time-Triggered Network-on-Chip (TTNoC)* [PK08]. It interconnects multiple, possibly heterogeneous IP-blocks called *micro components*.

The TTSoC architecture also introduces a *Trusted Subsystem*. On the one hand, its tasks is to ensure that a fault (e.g., a software fault) within the *host* of a micro component cannot lead to a violation of the micro component's temporal interface specification. Otherwise, the communication between other micro components could be disrupted. For this reason, the Trusted Subsystem (TSS) prevents a faulty micro component from sending messages during the sending slots of any other micro component. On the other hand, the TSS realizes the core services of the TTSoC architecture.

Due to the support of integrated resource management, the TTSoC architecture establishes two special entities: the *Trusted Network Authority* and the *Resource Management Authority*. The objective of their interaction is to accept resource allocation requests from the micro components and reconfigure the SoC. For instance, they could initiate a dynamic update of the Time-Triggered Communication Schedule of the TSS, or switching between power modes (see section 2.3.4).

[1] Yet, the diagnostic service is not covered in this thesis.

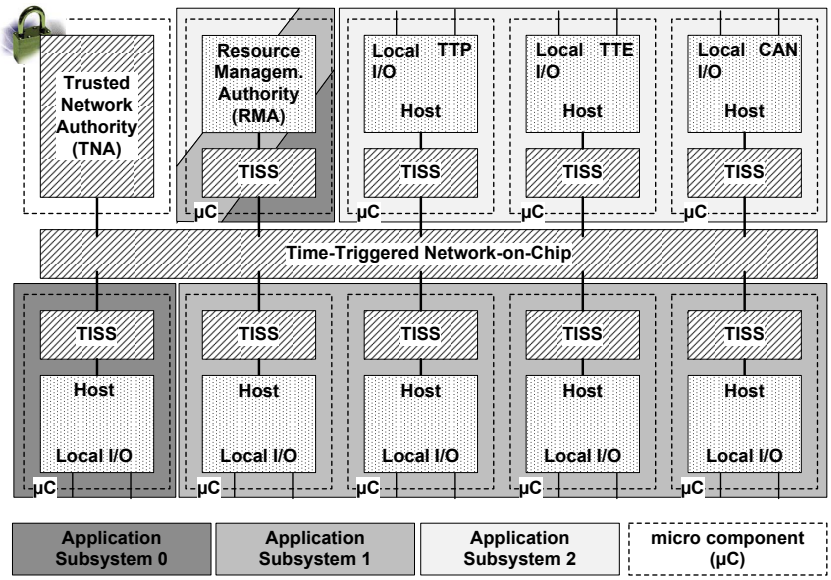


Figure 2.1: Structure of the TTSoC architecture

2.3.2 Micro Component

A TTSoC system can host multiple Distributed Application Subsystems (DASs) [OPHS06] (possibly of different criticality levels), which provide a part of the service of the overall system each. An example of a Distributed Application Subsystem (DAS) in the automotive domain is a braking subsystem. We call a nearly autonomous and possibly heterogeneous IP-block, which is used by a particular DAS, a *micro component*. A micro component is a self-contained computing element. For example, it can be implemented as a general purpose processor or as special purpose hardware. A DAS can be realized on a single micro component or by using a group of possibly heterogeneous micro components (either on one or multiple interconnected SoCs).

The interaction between the micro components of a DAS manifests solely in the exchange of messages through the Time-Triggered Network-on-Chip (TTNoC). Each micro component is **encapsulated**, i.e., the behaviour of a micro component can neither disrupt the computations nor the communication performed by other micro components. By design, encapsulation prevents temporal interference (e.g., delaying messages or computations in another micro component) and spatial interference (e.g., overwriting a message produced by another micro component). The only chance, when a faulty micro component can affect other micro components, is when a micro component gives faulty input (in the value domain) to other micro components via the sent messages.

With encapsulation we have the appropriate means to install TMR. TMR aids the detection and masking of failure of a micro component. Encapsulation is necessary to guarantee the independence of replicas. Otherwise, a faulty micro component

could disrupt communication or communication of the replicas. This would end in common mode failures.

Moreover, we use encapsulation to achieve the property of *composability* [Sif05]. Composability denotes the capability of a system to incrementally add new micro components, without overriding prior services of the already existing micro components. This is the requirement, if we want to seamlessly integrate independently developed DASs and micro components in the TTSoC architecture. The same applies to SoCs that encompass the application of subsystems of different criticality levels. In such a mixed criticality system, a failure of micro components of a non safety-critical application subsystem must not cause the failure of application subsystems of higher criticality.

We also pay tribute to encapsulation with the structure of a micro component. It comprises two parts: a *host* and a so-called *Trusted Interface Subsystem*, which is part of the TSS. The host implements the application services. By introducing the Trusted Interface Subsystem (TISS), the TTSoC architecture provides a dedicated architectural element that protects the access to the TTNoC. Each TISS contains a data structure, which stores *a priori* knowledge concerning the global instants in time of all message receptions and transmissions of the respective micro components – the Time-Triggered Communication Schedule. Since this Time-Triggered Communication Schedule cannot be modified by the host, a design fault or a hardware fault in the host cannot affect the exchange of messages in the TSS.

2.3.3 Requirements for the Trusted Subsystem

The TTNoC, which is a part of the TSS, interconnects the micro components of a TTSoC. Moreover, the scope of the TSS encompasses clock synchronization for the establishment of a global time base, as well as the predictable transport of periodic and sporadic messages.

Clock Synchronization

The TSS performs clock synchronization. The goal of clock synchronization is to provide a global time base for all micro components, even though there might be multiple clock domains in a TTSoC. The global time base is based on a uniform time format, which has been standardized by the OMG in the smart transducer interface standard [AC03].

A digital time format can be characterized by the three parameters: granularity, horizon and epoch. The granularity determines the minimum interval between two adjacent ticks of a clock. In other words, this is the smallest interval that can be measured with this time format. The reasonable granularity can be derived from the achieved precision of the clock synchronization [KO87]. The horizon determines the instant when the time is going to wrap around. The epoch denotes the instant when the measuring of the time begins.

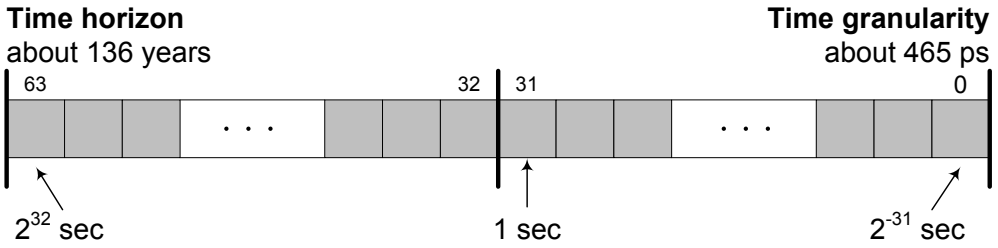


Figure 2.2: Time format used in the TSS

The time format of the global time base in the TSS (see Figure 2.2) is a binary time format that is based on the physical second. Fractions of a second are represented as 32 negative powers of two (down to about 465 picoseconds), and full seconds are presented in 32 positive powers of two (up to about 136 years). This time format is closely related to the time format of the GPS (General Positioning System) time and takes the epoch from GPS. In case there is no external synchronization [KO87], the epoch starts with the power-up instant.

Predictable Transport of Messages

In the TTSoC architecture we divide the available bandwidth of the TTNoC into periodic, conflict-free sending slots by means of Time Division Multiple Access (TDMA). Such a TDMA slot can be used in two different ways. Either it transports data of *periodic messages*, or *sporadic transmission of messages*. In the latter case, the sender issues a messages, whenever it intends to notify a receiver about a new event.

In addition to TDMA slots, we introduce the communication primitive of *pulsed data stream* [Kop06]. The purpose of a pulsed data stream is to model the allocation of TDMA slots. A pulsed data stream is a time-triggered, periodic, uni-directional data stream. It transports data in pulses, which possess a defined length and a defined topology. The pulsed data stream originates and *exactly one sender* and propagates to *at least one receiver*, which are identified *a priori*. Also, the period and phase in a periodic control system are given *a priori*.

Such pulsed behaviour is predestined for efficient transmission of large data in applications, which demand a temporal alignment of sender and receiver. This is particularly the case for those applications, which require a short latency between sender and receiver, for instance most real-time systems. For example, consider a fictitious control loop realized by three micro components. Micro component (A) performs sensor data acquisition, micro component (C) processes the control algorithm (C), and micro component (E) operates the actuator, as it is depicted in Figure 2.3[2]. In this application temporal alignment is vital to reduce the end-to-end latency of the control loop, which is an important quality characteristics. For instance, this requirement applies to the transmission of the control value (D) and the start of

[2]In this cyclic model of time, the perimeter represents the period of the control application.

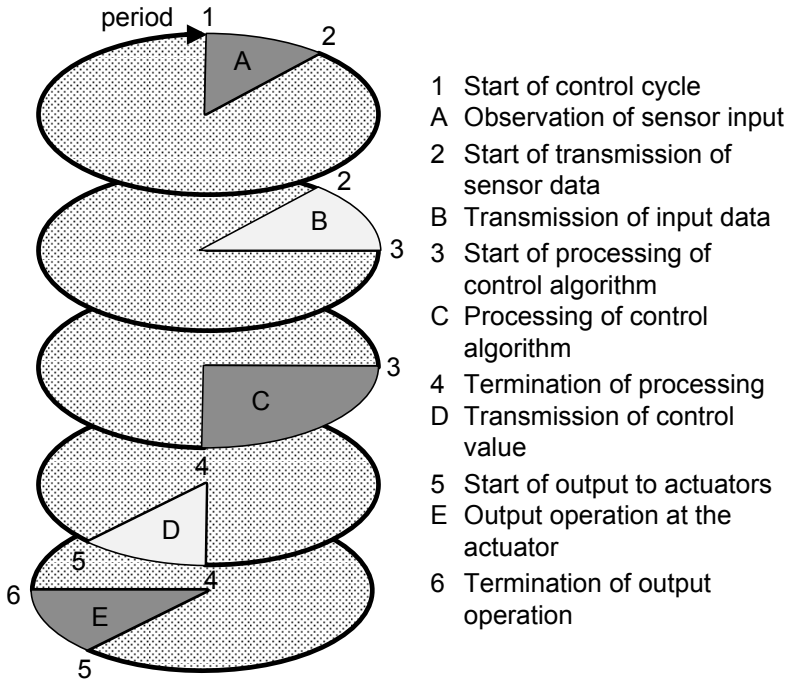


Figure 2.3: Temporal Alignment in Control Loops

actuator output (see instant 5 in Figure 2.3), as well as the sensor data transmission (B) and the start of the processing of the control algorithm (see instant 3 in Figure 2.3). With pulsed data streams we can achieve an efficient, temporal alignment of data transmissions, which refer to (B) and (D) in Figure 2.3.

Contrary to the communication service of the TTSoC architecture, many existing NoCs only provide the feature of guaranteed bandwidth to individual senders. The lack of temporal alignment results in the following consequences [SOHK08]:

- either the short latency can not be guaranteed,
- a high bandwidth has to be granted to the sender throughout the entire period of the control cycle, although it is only required for a short interval, or
- the communication system has to be periodically reconfigured in order to free and re-allocate the non-used communication resources.

Similarly, in a fault-tolerant system that masks failures by TMR, a high bandwidth communication service is required for short intervals to exchange and vote on the state data of the replicated channels.

To sum up, a real-time communication network should consider these pulsed communication requirements and provide appropriate services. The TTSoC architecture addresses this issue, eventually.

2.3.4 Architectural Elements of Resource Management

Integrated resource management in the TTSoC architecture serves the purpose to dynamically assign computational resources (i.e., micro components) to application subsystems and to grant communication resources and power to the individual micro components.

We distinguish between two fundamentally different types of application subsystems [SOHK08].

Safety-critical application subsystems need to be certified to the highest criticality classes (e.g., class A according to DO-178B).

Non safety-critical applications subsystems do not require certification to the highest criticality classes.

In general, these two types of application subsystems desire fundamentally different (also conflicting) design paradigms. Safety-critical applications focus on simplicity and determinism. Therefore, it is straightforward to facilitate thorough verification and validation. In contrast, non safety-critical applications can provide more complex application services, whereas we do not have sufficient *a priori* knowledge about the environment. Moreover, the dynamics to handle evolving application scenarios and changing environments can be challenging.

In the TTSoC architecture, two distinct architectural elements deal with integrated resource management, namely the Trusted Network Authority (TNA) and the Resource Management Authority (RMA). The RMA computes new resource allocations for the non safety-critical application subsystems, while the TNA ensures that the new resource allocations have no adverse effect on the behaviour of the safety-critical application subsystems. As depicted in Figure 2.1, the TNA is part of the TSS of the TTSoC, whereas the RMA is not. A consequence of splitting the entire resource management into two distinct parts (whereas only one is part of the TSS) the certification of the TTSoC is significantly simplified. We explain this with the fact that checking the correctness of a resource allocation through the TNA is significantly simpler than its generation through the RMA.

Resource Management Authority

The RMA is in charge of scheduling of available resources, which are allocated among the micro components. For this purpose the RMA leverages application-specific (e.g., communication topology) and system knowledge (e.g., temporal properties of the TTNoC). Nevertheless, the RMA is not allowed to change the configuration of the TSS directly, i.e., to update the affected TISSs.

Trusted Network Authority

The TNA is some kind of guardian for the reconfiguration activities performed by the RMA. Therefore, the TNA goes through the resource reservations from the RMA and looks out for potential collisions on the TTNoC or any violations of resource reservations.

If an erroneous resource schedule is detected, the TNA turns away this new resource reservations from the RMA. Then, the current configuration remains unchanged.

In case of the TNA regards the reservations as correct, it updates the configuration of the micro components. Since the TNA is part of the TSS it possesses the privilege to reconfigure the micro components via the TISS.

2.3.5 Gateways

In ultra-dependable systems, a maximum failure rate of 10^{-9} critical failures per hour is demanded [SWH95]. Today's technology does not support the manufacturing of chips with failure rates, which are low enough to meet these reliability requirements. Actually, component failure rates are usually in the magnitude of 10^{-5} to 10^{-6} [PMH98]. Unfortunately, ultra-dependable applications require the system as a whole to be more reliable than any one of its components. With this technological limits in mind, we can only achieve the required reliability by means of fault-tolerant strategies. Such strategies aim at a continuous operation despite the presence of component failures.

The TTSoC architecture supports gateways for accessing chip-external networks (e.g., TTP [KG94], TTE [KAGS05] and CAN [Bos91]). With such a gateway we can realize interoperability with public networks, for example the Internet, and the ability to interconnect multiple SoCs to a distributed system. As a result, we can build distributed systems that house applications for ultra-dependable systems based on the TTSoC architecture.

If the chip-external network is also time-triggered (e.g., TTP, TTE), we can leverage the TDMA schemes of both (the internal and the chip-external network) to synchronize communication across device borders. Consequently, a message that is sent on the chip-external network arrives at the micro components within a bounded delay with minimum jitter (solely depending on the granularity of the global time base). The alignment between messages on time-triggered networks ensures that replicated TTSoCs perceive a message at the same time, i.e., within the same inactivity interval of the global (sparse) time base [Kop92]. This property is significant for achieving replica determinism [Pol94] as required for active redundancy based on exact voting. Without synchronization between the TTNoC and the chip-external network, there could always occur a scenario in which one TTSoC forwards the message to the micro components in one period of the pulsed data stream, while another TTSoC would forward the message in the next period.

Furthermore, the introduced gateways can provide the TTSoC with an externally synchronized time base. For example, the global time base of the TTSoC can be synchronized to GPS. As a consequence, a time stamp assigned to an event is also meaningful outside the TTSoC. Last but not least, the global time base enables a global coordination of activities spanning multiple TTSoCs (e.g., output to actuators at the same global point in time).

2.3.6 Support for Diagnosis

The TTSoC architecture incorporates a dedicated architectural element for the purpose of diagnosis – the so-called *Diagnostic Unit* [SOH+07, POS+07]. Diagnosis is executed in three phases:

1. failure detection
2. dissemination
3. analysis

Failure Detection

Most architectural elements in the TTSoC architecture, for instance the TISSs, the TNA etc., perform failure detection in order to indicate faulty and abnormal behaviour of micro components.

The TISS is equipped with mechanisms to observe the behaviour of the attached host. For instance, a *watchdog* monitors the host, whether it has crashed or it is still alive. Other mechanisms recognize violations of message arrivals. The host performs an application-specific failure detection. These are not defined by the TTSoC architecture. The RMA checks the resource requests of the micro components against predefined constraints. Each invalid request is recorded since it might indicate a failure in the requesting component. The Diagnostic Unit (DU) uses message classification to detect failures in the overall system. For this purpose, the messages of all micro components are routed to the DU , which controls syntactic, temporal, and semantic correctness [POS+07].

Dissemination

Detected failures in the single entities are reported to the DU via the *dissemination service*. This special service produces *failure indication messages*. A failure indication message includes information concerning *type* of the occurred failure (e.g., crash failure of a host, invalid resource requests), the *instant* of detection, and the *location* within the TTSoC (i.e., the micro component). The dissemination service is associated with a dedicated time-triggered message for each micro component that is capable of delivering failure indication messages.

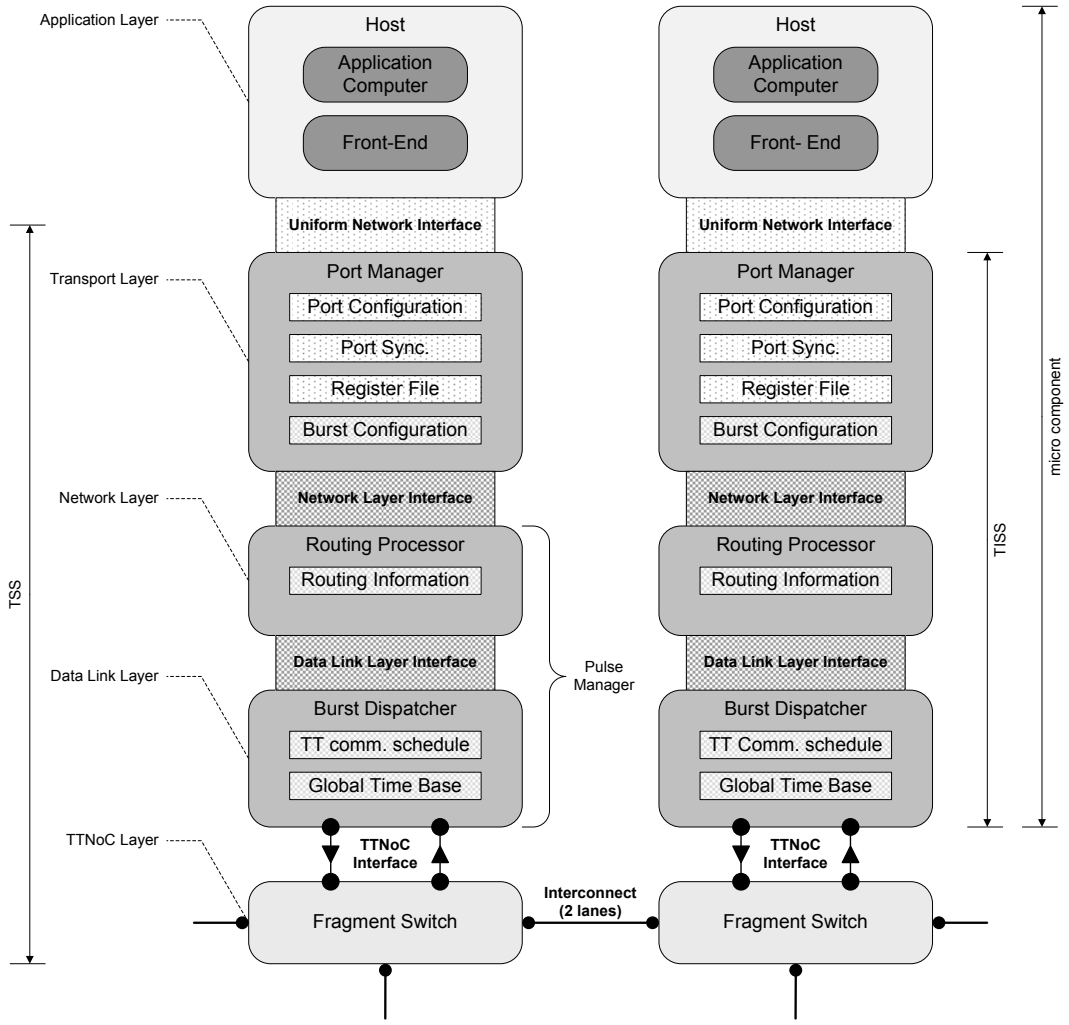


Figure 2.4: Schematics of the TTSoC prototype implementation

Analysis

Based on the collected failure indication messages, the DU establishes a holistic view onto the system. Furthermore, the DU monitors the repeated occurrence of failure indications in order to distinguish between permanent and transient failures.

2.4 Implementation Overview

This section sketches the prototypic implementation of the TTSoC architecture. It introduces conceptual layers and the modules, which reside at these layers. Each module fulfils a functions that has been introduced earlier in this chapters.

Figure 2.4 shows a vertical schematics of the implementation of the TTSoC architecture. We introduce 5 layers, which relate to the layers of the TCP/IP stack [Car96] or the ISO OSI reference model [Ros90] in naming and purpose.

2.4.1 The Five Layers

At the bottom the TTSoC architecture defines the TTNoC layer. This is the conceptual layer, where the realization of the TTNoC resides. Figure 2.4 mentions Fragment Switches that build the TTNoC. We dedicate chapter 6 to the implementation of the TTNoC.

The interface between the bottom TTNoC layer and the Data Link layer is the TTNoC interface. It is the physical connection between the Fragment Switch, which is closest to the TISS of a micro component and the bottom interface of the TISS.

The TISS spans over 3 layers with an own entity at each layer. Details about the TISS can be found in chapter 7.

At the bottom at the Data Link layer there resides the Burst Dispatcher. Its purpose is to handle the arbitration of the TTNoC. Therefore, the Burst Dispatcher includes the global time base, which synchronizes communication activities among micro components. The Time-Triggered Communication Schedule gives the necessary information, how this arbitration is managed.

Then, the Network Layer is in charge of building up communication channels between micro components. The routing information, which is used in the Routing Processor, determines the route of a communication channel along the TTNoC.

In the Transport Layer there resides the Port Manager. It relies on the services of the lower layers to realize major part of the communication service. It cares for access and synchronization of "ports", which are the end-points of communication channels. It controls the flow of fragments of messages for send and receive operations. Moreover, the Port Manager includes the Register File of the TISS, which houses status information and user settings. To provide its service the Port Manager is equipped with several memory entities: the Burst Configuration Memory, the Port Synchronization Memory, and the Port Configuration Memory.

The Uniform Network Interface (UNI) is the interface on top of the TISS. Physically, it is the interface between the TISS and the host. Logically, it provides access to the architectural core services implemented by the TSS. Chapter 5 contains a detailed description of the UNI.

The UNI is intended to be a generic interface to the Application Layer, where application-specific hosts can be attached. A host is made up of the Front-End and an application computer, which can be an own IP-block, a soft-core CPU such as the Altera Nios II^{TM} Embedded Processor[3], or an off-chip micro controller. To provide maximum portability, a Front-End is introduced to wrap the characteristics of the

[3] `http://www.altera.com/products/ip/processors/nios2/ni2-index.html`

UNI to match the actual application computer. Besides this, the Front-End contains the *Port Memory*.

The Port Memory is a (true) dual-ported memory that is shared between the application computer of the host and the TISS. It houses the "ports", that is, "ports" contain the application data associated with messages of communication channels. The role of the Port Memory can take any kind of (true) dual-ported memory outside the TISS that is attached to the *Port Interface* (see section 5.1). Another possible implementation is a scratch pad or L1 cache of a microprocessor [HP06], whereas the CPU is attached at one side and the Port Interface at the other. Apparently, the implementation of the Port Memory is subject of the target hardware, in which the TTSoC architecture is hosted. The layout of the ports residing in the Port Memory is covered within the scope of the UNI, and therefore explained in chapter 5.

Figure 2.4 also illustrates the scope of each layer, that is, to which architectural component a layer belongs to as introduced in chapter 2.3. The TTNoC layer embodies an own architectural component, obviously this is the TTNoC. Everything else is part of the micro component. However, we also learn from Figure 2.4 that each micro component contains parts of the TSS, i.e., the TISS. By definition the TTNoC is also part of the TSS.

2.4.2 System Operation Frequency

Note that the prototypic implementation uses the first arrangement of clock domains for the communication subsystem. That is, the TSS is equipped with a single, synchronous clock domain. Like the global time base, it spans across the whole system, and drives each Fragment Switch of the TTNoC as well as all TISSs. This particular clock domain is named the *system operation frequency*. This system operation frequency is a multiple of the frequency associated with the global time base.

Chapter 3

Related Work

This chapter examines related work. Even though, the research area of SoC designs has plenty of contributions, we focus on those alternatives, which reveal some relevance for the TTSoC architecture. We also compare the characteristics of the related work with the TTSoC architecture.

In general, we can distinguish between two distinct approaches:

interconnect fabrics concern the "local" scope of an IP-core. The abstraction level of interconnect fabrics focuses the linkage of peripheral components such as memory controllers, Ethernet controllers, or DMA controllers with one or more CPU cores. In terms of the TTSoC architecture, an interconnect fabric addresses the realization of a micro component.

system architectures aim at the connection of several IP-cores. Usually, they include an NoC architecture. System architectures define interfaces to IP-cores, but their realization is not covered. In terms of the TTSoC architecture, a system architecture deals with the connection of several micro components.

In the following we introduce prominent examples of interconnect fabrics as well as system architectures.

3.1 The AMBA Family

The Advanced Microcontroller Bus Architecture (AMBA) gathers a series of standards intended for on-chip communication in high-performance microcontrollers. According to the definition above, it can be regarded as an interconnect fabric. AMBA has established as de-facto standard in the field of microcontroller design. During the past decade, it has undergone several evolutionary steps, whereas each version extended the feature set and increased performance.

3.1.1 Advanced High-Performance Bus

Advanced High-Performance Bus (AHB) [ARM99] is the most long-lasting version of the AMBA family. It serves as a high-performance system *backbone* bus. It has been designed to interconnect high clocked system modules like CPU cores and on-chip memories, whereas performance and throughput are of utmost importance. Figure 3.1 shows an example of a microcontroller design that is equipped with a typical set of modules, e.g., the AHB Backbone, a CPU, a memory controller, a Direct Memory Access (DMA) controller.

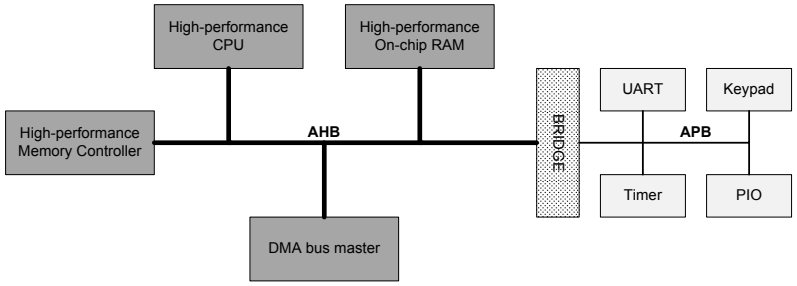


Figure 3.1: Example of an AHB-based microcontroller design

Moreover, we see in Figure 3.1 that a bridge builds the link to an Advanced Peripheral Bus (APB), which mainly interconnects low-power, low-performance peripherals like I/O modules and timers. Other than the AHB, which features high throughput, high performance by means of pipelined operation, burst transfers, as well as multiple bus masters, the APB provides a reduced functionality. Instead of maximum performance, APB focuses on minimal power consumption.

AHB defines four different roles, whereas each participant of AMBA communication takes one role.

AHB Master is the initiator of read and write operations. AHB is enabled to contain multiple masters, however only one is allowed to occupy the backbone at a given instant of time.

AHB Slave responds to requests from AHB Masters. Each slave defines a specific address range, where it offers its services or data to the masters. AHB Slaves also indicate the status of the previous transfer (e.g., "success" of "failure") to the active master.

AHB Arbiter ensures that exactly one AHB Master occupies the backbone at any time. In other words, the AHB Arbiter grants access to the shared bus. AMBA supports different arbitration protocols (e.g., priority-based arbitration), which can be chosen according to the target application's requirements.

AHB Decoder operates as an address decoder. According to the address, which the currently active master requests, the AHB Decoder selects the proper AHB

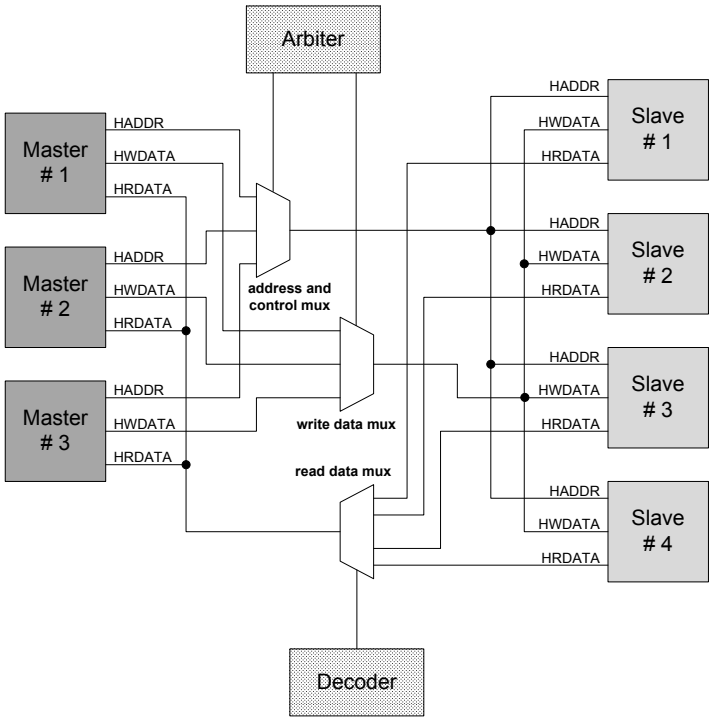


Figure 3.2: Interconnection scheme of AHB

Slave, which has the requested address in its address range. The AHB Decoder is a single, centralized module in the AMBA interconnect fabric.

According to its specification, AMBA AHB is based on the master/slave paradigm. The backbone is realized as a central multiplexer. When an AHB Master begins a read or write transfer, it issues some address, which refers to a given AHB Slave. The AHB Arbiter selects the address, control, and write data signals from the currently active AHB Master by means of multiplexers. The multiplexed signals are routed to each slave, then. Contrary, the AHB Decoder multiplexes all read data signals in order to select the output signals of that slave, which services the previous request by a AHB Master. The multiplexed read data vector is shared among all masters, whereas the most recent active master fetches the response from the according slave. Figure 3.2 illustrates this pattern of interconnects of AHB.

An AHB Master must be granted access to the backbone before it starts a transfer. For this purpose, it asserts a request signal (`HBUSREQx` for master no. x) to the AHB Arbiter. The master is stalled, until the arbiter has granted access to the master (`HGRANTx` for master no. x). Afterwards, the AHB Master starts the transfer by driving the control and address signals, which transport information concerning address, read / write, and length of the operation. AHB provides different types of operations such as basic transfer that concern single data words as well as sequential burst operations beginning at a given start address.

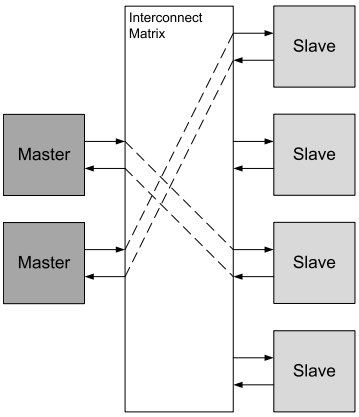


Figure 3.3: Structure of Multi-Layer AHB

3.1.2 Multi-Layer AHB

AHB in its earliest version suffers from the restriction that there is only one AHB Master allowed to occupy the backbone. Multi-Layer AHB [ARM01], which is backward compatible to AHB, has been ratified to circumvent this architectural limit. An *interconnect matrix* provides parallel access paths between multiple masters and slaves at the same time. The interconnect matrix illustrated in Figure 3.3 connects each AHB Master with each AHB Slave. The connection from each master to every slave is called an AHB *layer*.

Within the interconnect matrix, each layer has a dedicated AHB Decoder that serves the request for read and write transfers of one AHB Master, as shown in Figure 3.4. If two or more layers demand for access to the same slave, arbitration must be executed. Each slave is equipped with dedicated arbitration logic (i.e., an AHB Arbiter), therefore an individual arbitration scheme (e.g., round robin or fixed priority) can be applied for each slave.

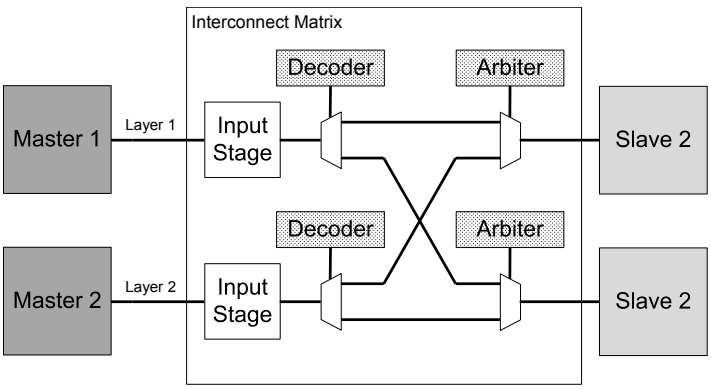
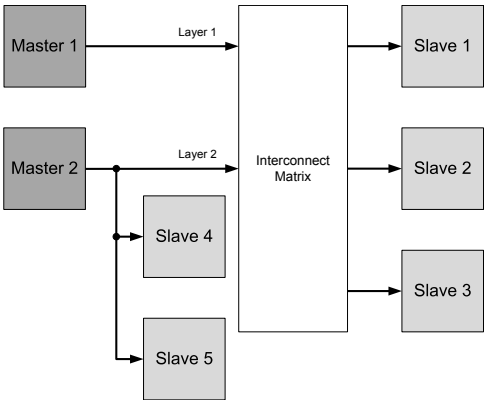
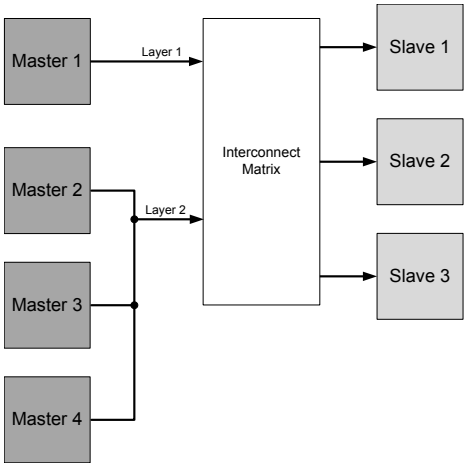


Figure 3.4: Internal view on the Multi-Layer AHB interconnect matrix

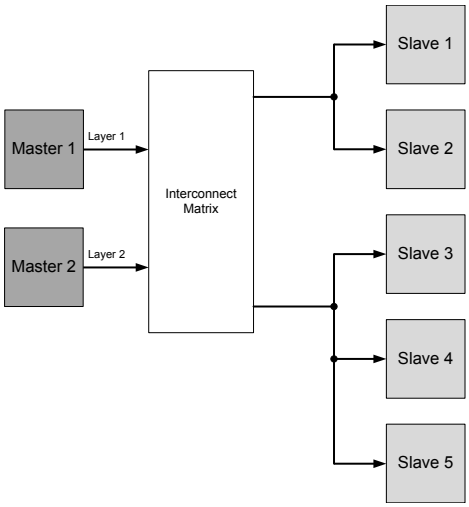
Multi-Layer AHB allows to reduce the complexity of the interconnect matrix. It is possible to share ports of the interconnect matrix by multiple masters or slaves so that not every master and slave demand for own ports. Three different scenarios depicted in Figure 3.5 exist in the standard, which can be combined.



(a) Local slaves



(b) Multiple masters on a single layer



(c) Multiple slaves on a single port

Figure 3.5: Possible groupings in Multi-Layer AHB

local slaves An AHB Master can keep a subset of AHB Slaves "private" so that these slaves are only attached to its own layer and are positioned before the interconnect matrix (see Figure 3.5a). Thus, the interconnect matrix becomes slimmer.

multiple slaves on a single slave port Multiple slaves can be grouped together and be accessible at a single port of the interconnect matrix (see Figure 3.5b). In this case, the bandwidth of that layer is shared among the slaves. Such a scenario might be convenient for debugging, when a set of slaves needs to be accessed by more than one master.

multiple masters on a single layer Like AHB Slaves, AHB Masters can be grouped to share a single port of the interconnect matrix (see Figure 3.5c). It is reasonable to organize masters with low bandwidth requirements and no need for parallel channels in such a way.

3.1.3 AXI

AMBA AXI [ARM04] – also called "AMBA 3.0" – is the most recent and most advanced member of the AMBA family. It facilitates maximum resource efficiency and throughput, as it introduces additional features like multiple outstanding transactions, out-of-order transaction completion, and efficient burst transactions (with solely the start address issued).

In traditional AHB, each transfer is structured into an address and a subsequent data phase. This organisation inherently entails pipelining. While the $(i + 1)^{th}$ address is issued, the i^{th} transaction is processed. However, the restriction that the data phase must immediately follow the address phase of the same transaction prevents multiple outstanding transactions.

In contrast to AHB, AXI introduces *ID tags* to transactions. All transactions that belong to the same ID tag have to be completed *in order*, while transactions with different ID tags can be interleaved and completed *out of order*. This property enables multiple outstanding transactions and out-of-order transaction completion. The latter contributes to better performance, as it allows to complete fast-responding slaves ahead of slower slaves, although the transaction towards the slower slaves has been initiated first. In other words, fast transactions may "overtake" slower transactions.

AMBA AXI introduces the notion of *channels*. Each of five independent channels fulfils one specific functions for communication.

Read address channel carries address and control information for all read transactions.

Write address channel is similar to the read address channel, but it transport address and control information for all write transactions.

Read data channel transports the read data and additional response information (e.g., the completion status) from the slave to the master, which has initiated the transaction.

Write data channel conveys write data to a slave. Data spread through the write data channel is buffered, so that the master need not wait for commitment and acknowledgement of previous write transactions.

Write response channel provides feedback like completion signals from the slave to masters. Completion signals occur only once for each burst, but not for each transfer within a burst.

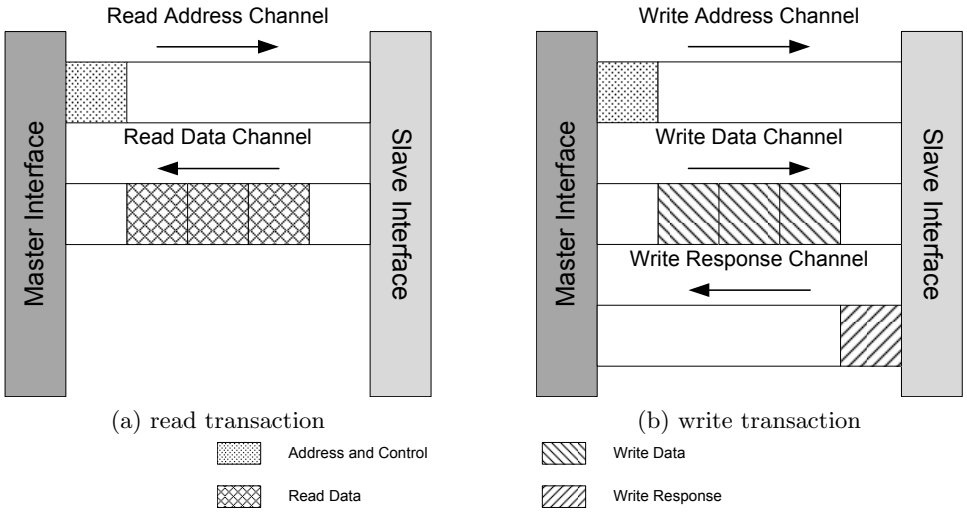


Figure 3.6: Transactions in AHB AXI

Figure 3.6a depicts how the read address channel and the read data channels are operated during a read transactions. Figure 3.6b illustrates the usage of write address, write data, and write response channel for write transactions.

Each channel executes a two-way handshake mechanism. The signal `VALID` indicates that new data or control information is available on that channel, while the destination signals via `READY` to acknowledge the receipt of that information. The read and write data channels include an additional signal `LAST` to pinpoint the very last data item of a transaction. This two-way handshake mechanism enables the master as well as the slave to control the rate at which data or control information is transferred. Hence, it includes a flow control mechanism.

AMBA AXI support a flexible arrangement of channels between masters and slaves. We can identify three different approaches.

Shared Address and Single Data buses (SASD) Only a single master can be active per channel. This concept equals the capabilities of AHB.

Shared Address and Multiple Data buses (SAMD) A single master can be active per channel, but several channels can be driven by distinct masters.

Multiple Address and Multiple Data buses (MAMD) For each channel several masters are allowed to use that channel. This is the most performant and also the most complex scenario, as the verification of the system has to consider access conflict by subsets of master at a given channel.

3.2 Open Core Protocol

The Open Core Protocol (OCP) [Ass05] specifies a non-bus, *point-to-point interface* between pairs of communication IP-cores. IP-cores comprise simple peripheral components (e.g., UART, timer modules), on-chip subsystems, or whole microprocessor cores. OCP follows the *master / slave* paradigm, that is, for each point-to-point connection one IP-core takes the role of the master, while the opposite core plays the slave. The master has exclusive control in the point-to-point connection. It initiates all transfers, while the slave responds to the commands, either by accepting data from the master or by presenting requested data to the master. So, a point-to-point connection is uni-directional. The design decision, which IP-core takes what role is dependent on the targeted communication pattern between the two IP-cores.

In order to set up a peer-to-peer communication, two instances of point-to-point OCP connections must be installed: one connection, where the first core is the master and the second core the slave, and another connection, where the roles of first and second core are reverse. Figure 3.7 visualizes the arrangement of OCP connections to peer-to-peer communication channels between IP-cores.

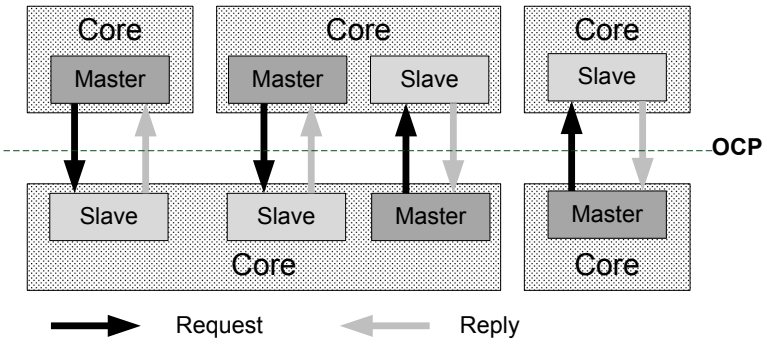


Figure 3.7: Arrangement of OCP connections

OCP supports simple write and read transactions, which enable the master to write or read specific data words at a given address to or from the slave. Additionally, master and slave can exchange *side-band information* via dedicated signals. For example, such additional information are interrupts, error indications, or reset requests. Side-band signals can also be used to establish a flow control between master and slave. Moreover, OCP involves a basic mechanism of burst transactions.

OCP defines a strictly synchronous interface, whereas all signals across a point-to-point connection are driven by a single clock, i.e., belong to a single clock domain. Besides this, each signal is uni-directional, while the driver is either the master or the slave.

OCP does not incorporate interconnections between multiple masters and slaves. Each topology must be reduced to uni-directional point-to-point connections. As a consequence, there is no need to consider arbitration or address decoding. From the architectural point of view, a designer can regard two IP-cores as they would

be directly connected, and OCP dictates a naming convention and semantics of the interconnecting signals.

3.3 Æthereal

The Æthereal [GDR05] architecture serves as a NoC for SoC designs. It provides two distinct traffic classes of communication [GvMPW02]:

guaranteed services For guaranteed services, Æthereal assures a specific amount of bandwidth that can be used by a given communication channel. Additionally, the latency of transmission is bounded for guaranteed services.

best effort services A best effort service occupies the bandwidth, which has been left over by the communication channels of guaranteed services.

The purpose of guaranteed services is to aid a compositional design of robust SoCs, while best effort services increase the resource efficiency, as they claim the NoCs capacity that is unused by guaranteed services. Both service classes provide data integrity, lossless data delivery and in-order data delivery [RGR+08].

An Æthereal NoC is constructed of *Network Interfaces* (NIs) [RDP+05] and *routers* [RGR+08], which are interconnected by *links*. There exists a specific type of router for each service class: Guaranteed Throughput (GT) routers and Best Effort (BE) routers.

Within the NoC, Æthereal establishes a packet-based protocol. Generally, IP-cores that are attached to a Network Interface (NI) each can be free to realize any protocol. Therefore, the NI has to translate the local protocol of the IP-core to the internal packet-based protocol. The routers transport such packets – so-called *messages* – among the NoC, while NIs are the source and termination points of transmissions.

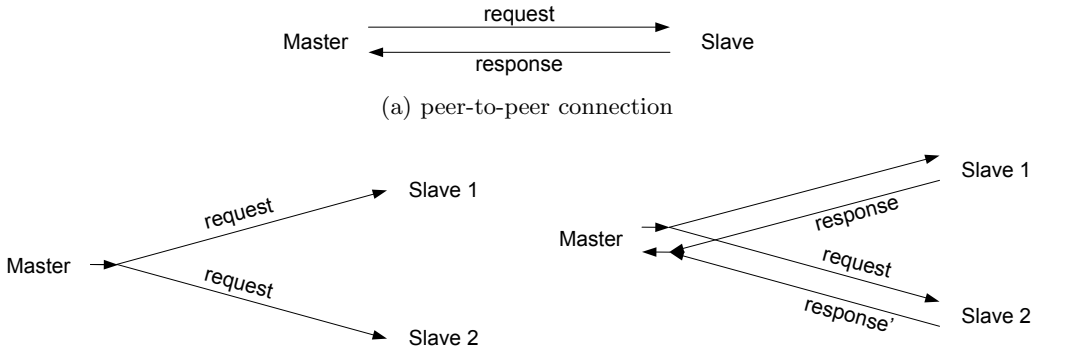
Æthereal presents a *shared-memory abstraction* [RDG+04] to IP-cores. The protocol employed at the NIs is a transaction based master/slave model, which has been chosen to ensure backward compatibility to existing on-chip interconnect fabrics like AXI [ARM04] or OCP [Ass05].

The signals of an IP-core towards an NI are designed according to a given specification, e.g., AXI or OCP. The signals are wrapped into *request messages* and *response messages*. Then, these messages are transported by the Æthereal NoC contained in *packets*.

3.3.1 Connections in Æthereal

The communication service introduces the concept of so-called *connections*, which in turn are composed of several uni-directional peer-to-peer *channels* [RDP+05]. With

these entities, Æthereal models, e.g., OCP peer-to-peer connections, which consist of a (uni-directional) request channel and a corresponding (uni-directional) response channel (see Figure 3.8a). Another example, a multi-cast (see Figure 3.8b) or narrow-cast (see Figure 3.8c) connection could be implemented by a collection of channels, whereas each channel flows from a master to a given slave.



(a) peer-to-peer connection

(b) multi-cast

(c) narrow-cast (transaction processed by slave 2)

Figure 3.8: Connection types in Æthereal

Each channel involves two queues, whereas one queue resides in the NI of the sender, and the other queue is situated in the NI of the receiver. Figure 3.9 illustrates both queues and their position in the Æthereal architecture. The queues are protected by an *end-to-end credit-based flow control mechanism* [MB06], which prevents queue overflows in the NIs. With this mechanism it is possible to guarantee message delivery and avoid message losses due to overflow.

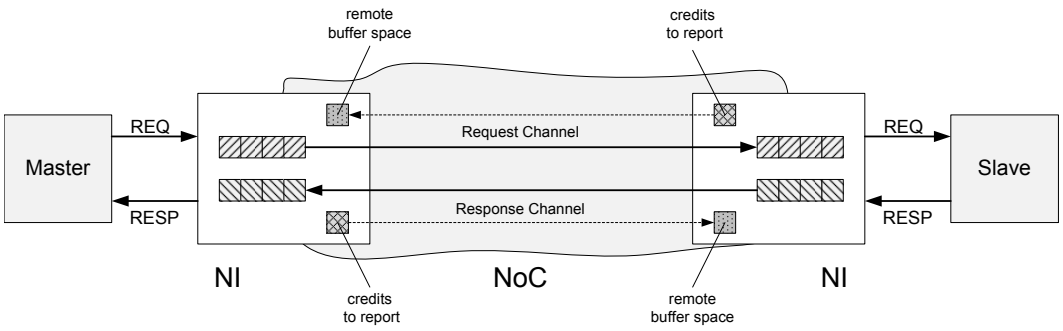


Figure 3.9: Æthereal credit-based flow control

At the sender side, a channel incorporates a counter, which keeps track of the available buffer space at the remote destination queue [RDG+04]. The initial value of this counter is the queue size of the remote destination queue. The counter is decremented, whenever data is sent from the source queue to the remote destination queue. When the receiver, i.e., the IP-core attached at the NI, consumes data from its local destination queue of that channel, *credits* are generated. Such credits indicate that there is more space available in the destination queue now. The credits are

delivered to the sender of the channel, and the credits are added to the counter, which keeps track of the buffer space in the remote destination queue.

Credits can be transmitted in dedicated credit packets, or they are "piggybacked" in the header of packets of the channel into reverse direction, which improves the bandwidth efficiency of that credit-flow-control protocol.

Considering the service classes mentioned above, channels offer two types of transmission [RGR+08]: *GT* is associated with the "guaranteed services", whereas *BE* correlates to "best effort services".

The first, GT channels, which are routed through GT routers only, assure a minimum bandwidth and a maximum latency. The means to realize such guarantees is a TDMA scheme combined with *pipelined virtual circuit switching*. The TDMA scheme manifests in a table. The size S of that table is pre-defined, for instance $S = 128$ slots. It lists, which operation is executed by which NI during a given TDMA slot. The global table is split up into local replicas for each NI, which contain the local view on communication. The granularity of bandwidth that can be reserved equals $\frac{1}{S}$ of a channel's maximum bandwidth. In each slot, a NI is allowed either to read or write (respectively send and receive) at most one block of data. Depending on the duration of a slot and the size of a block processed during a slot, we can determine the bandwidth B for one slot. Consequently, if a channel is assigned N slots for transmission, it possess a bandwidth of $N \cdot B$.

In contrast to GT, BE channels are solely transported by BE routers. They offer different *routing modes* or (also called) *network flow control* such as packet switching, virtual cut-through or wormhole routing. The queues in BE channels can be operated according to input queuing or virtual-output queuing techniques.

A single channel has the ability to guarantee a temporal ordering of messages. That is, all messages appear in the same order at a receiver as they have been sent at the sender. However, the NI manages different channels independently. As a consequence, in the Æthereal architecture we can not reason about the ordering of messages across several channels. Furthermore, a reordering of message (across several channels) is possible to happen.

3.3.2 Structure of NIs

Figure 3.10 depicts the internal structure of an Æthereal NI. The NI consists of one *kernel* and some optional *shells*. The shells are those components, which translate the semantics of a given protocol, e.g., AXI or OCP, that is applied at the interface to the IP-core into generic messages. NI ports provide the physical access from the IP-core to the NI, for example ports implementing AXI or the Device Transaction Level (DTL) [Phi02].

The kernel wraps the generic messages into GT or BE packets, dispatches them according to the TDMA slot scheme, and realizes the end-to-end credit-based flow control. From the point of view of the NI kernel, the semantics as well as structure of the internal generic messages is irrelevant, as the NI kernel solely handles packets.

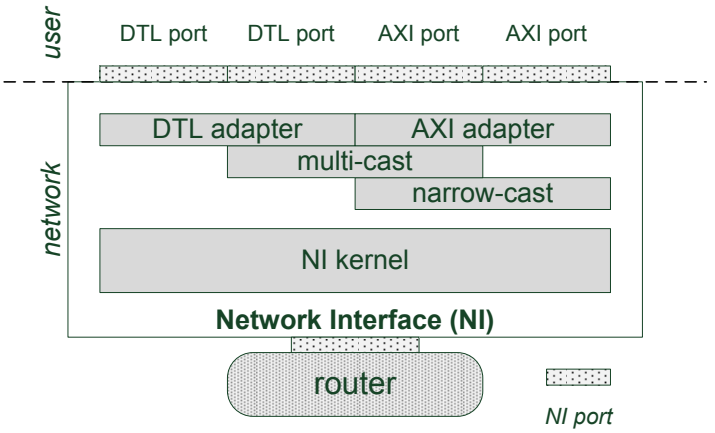


Figure 3.10: Æthereal NI kernel and shells

As the shells realize a higher abstraction than the packets of the NI kernel, several shells can be cascaded or combined in order to implement more advanced protocols. As shown in Figure 3.10, an NI can be assembled with shells that control multi-casting or narrow-casting connections.

3.3.3 Reconfiguration in Æthereal

Æthereal includes a concept of (static) run-time reconfiguration [HCG07]. A system contains pre-defined configurations, which describe channels and connections between IP-cores, that are associated with a given "user mode". A user mode reflects the communication requirements of the current functionality of the system, e.g., an active phone call, MP3 playback in a cell phone, as it is operated by a user. These configurations are deployed with the system, and can be loaded from a separate memory and distributed to NIs at run-time.

The design flow of reconfiguration begins with the specification of the application requirements of user modes [GDG+05]. Æthereal's concept of reconfiguration defines a high-level description of communication between IP (cores). A second specification lists the connections between IP, which are attached to the logical ports of IPs (core ports). With tool support, the specifications are verified off-line and prepared to run a *mapping and routing algorithm* [HGR07]. The purpose of this graph-based heuristic is to determine routes of channels and connections through the NoC, to arrange bandwidth by calculating the TDMA slot schemes, and to map core ports to ports of NIs according to the application requirements. The result of the algorithm is converted into a memory image that can be deployed with the system.

The process of reconfiguration [HG07] in Æthereal incorporates the NIs, for which the configuration data is intended and at least one dedicated architectural component – the *configuration master* – that is in charge of distributing new configuration data

to NIs. The operation of the configuration master is hidden from the application programmer, thus, reconfiguration is an autonomous subsystem of Æthereal.

The configuration data is carried over the same wiring of the NoC like the application data, whereas no additional control network is necessary. Therefore, the existing infrastructure is reused, and the distribution of configuration data can be modelled as connections and channels just like user-generated traffic.

The configuration master sets up a *configuration connection* to those NIs, which communication pattern deviates in the new configuration compared to the current configuration. The configuration master supports the following operations in order to manipulate channels and connections between pairs of masters and slaves:

- open a new connection between NIs
- modify an existing connection
- close an existing connection

During reconfiguration the configuration master iterates through the description image of the new configuration and identifies connections, which must be installed, altered, or destroyed. Depending on the type of operation, the configuration master opens a *configuration request channel* to each NI, which is affected by the operation. Open and destroy operations involve the NIs of a master and a corresponding slave, while for modify operations a single NI might be sufficient. After the configuration request channel is established, that NI gives back a *configuration response channel*. Finally, the configuration master ships the configuration data for that NI, whereas *configuration registers* accessible through the logical *configuration port* in the NI (i.e., kernel and shells) are read and written. There exist registers that hold configuration of paths, queue, channel type (GT, BE), space, credits etc.

Æthereal supports parallelism in the reconfiguration infrastructure. First, it is feasible to have several configuration masters in the system, which might build a hierarchy, whereas each level in the hierarchy is in charge of a given subsystem or set of subsystems. Secondly, the configuration master might handle several configuration connections to multiple NIs. Currently, Æthereal remains with a single configuration master with a single port for configuration connections.

3.4 Nostrum

The Nostrum [MNT02] architecture aims at the provision of different Quality of Service (QoS) guarantees. In fact, Nostrum supports *Guaranteed Bandwidth (GB)* as well as *Best Effort (BE)* services. Both service classes offer *packet switched communication*.

For BE, Nostrum proposes a load distribution algorithm called Proximity Congestion Awareness (PCA) [NMÖJ03], whereas switches in the NoC use load information

of neighbouring switches (called *stress values*) for their own switching decisions. This helps to avoid congestions.

For Guaranteed Bandwidth (GB) services, Nostrum introduces *Virtual Circuits*, which are implemented by a combination of two concepts [MNTJ04]: *Temporally Disjoint Networks (TDNs)* and *Looped Containers (LCs)*. Guaranteed bandwidth is achieved by transporting stream oriented data in special packets, so-called *containers*, which are routed by means of *deflective routing*[1] [FR92]. This routing strategy in combination with PCA allows to diminish — also to nullify depending on the communication patterns of the target application — the need for packet buffers in Nostrum switches, thus, increasing the tolerance against network disturbances, e.g., congestions, and contributing to energy efficiency.

The Nostrum backbone [MNT+04] defines a layered protocol stack, which enables reliable communication to the application level. The basis of the layering is the NoC with its switches. A Network Interface (NI) above the NoC offers a basic set of services, such as BE and GB. A Resource Network Interface (RNI) acts as glue logic between IP-core at application level and the NI.

3.4.1 Topology Decision

Generally, Nostrum is able to arrange its switches in any topology such as folded torus, fat-trees [Lei85] etc. However, Kumar et al. argue in [KJS+02], why they prefer *mesh topologies* for Nostrum.

Firstly, a bus topology has shown to suffer from limited scalability and poor performance for larger systems [GG00].

Secondly, higher order dimensional topologies are hard to implement. Culler and Singh analyse in [CSG98] that low dimensional topologies (i.e., two dimensional) are superior, when wiring and interconnects carry significant costs, high bandwidth occurs between switches, and delay in switches is in the magnitude of propagation delays along wiring. In this context, the torus topology was rejected for Nostrum, as a folded torus has twice as long inter-switch connections than a mesh.

Additionally, Kumar et al. are convinced that there is no need for higher dimensional topologies, since two dimensional meshes are quite efficient in a large number of target applications in the field of signal and multimedia processing. Besides this, a mesh topology entails some convenient properties, such as a simple addressing scheme, and efficient realization of multiple routes.

Finally, we see in the next section that Nostrum heavily relies on the geometry and implementation of mesh topologies to find efficient solutions to realize the concepts of the Temporally Disjoint Network (TDN) and the Looped Container (LC).

[1] also called *hot-potato routing*

3.4.2 Theory of Operation

The deflective routing in Nostrum implies that there are no buffer queues involved, which could reorder packets in a switch, i.e., packets leave a switch in the same order as they have entered. The packet duration is one clock cycle, in other words, the length of a packet is one flit. Packets entering a switch at the same clock cycle are exposed to the same processing delay through the switch, and therefore also leave simultaneously. However, due to different possible routes and corresponding path lengths for each packet, a reordering across the overall NoC is possible. The reason for packets taking different routes is that switching decision are made locally on a dynamic basis (i.e., PCA).

Temporally Disjoint Networks

Considering deflective routing, the property of non-reordering of packets (at a given switch) creates an implicit TDMA schema in the NoC. This is called TDN. A TDN results from the geometric layout of the NoC, whereas two factors are the *topology* and the *number of buffer stages* in the switches.

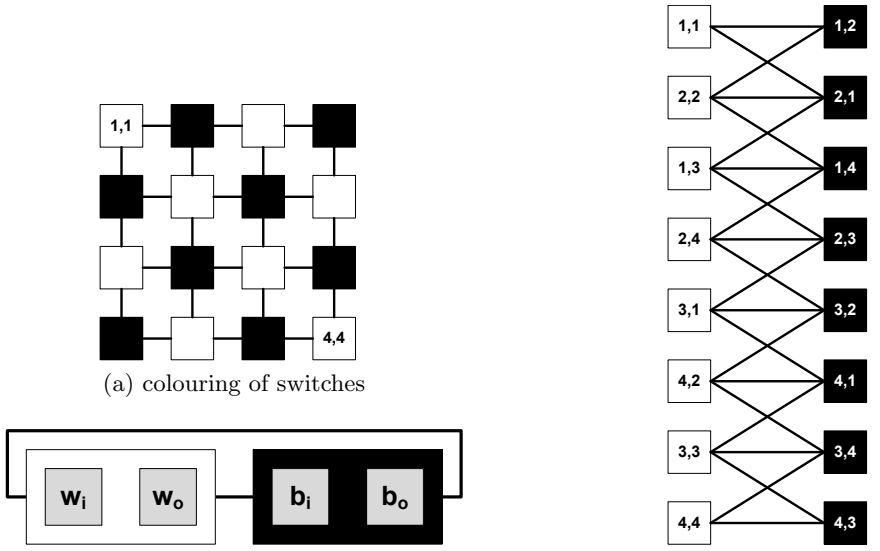


Figure 3.11: Temporally Disjoint Network

Topology Packets that are emitted on the same clock cycle can only collide — say, "will only be on the same net" — if they are on a multiple distance of the smallest round-trip delay.

For an explanation, Millberg et al. [MNTJ04] apply a colouring of the switches in the mesh topology, as shown in Figure 3.11a. Since white nodes are only connected

to black ones, and black nodes are only linked to white ones, every packet passes by a white and black node alternatingly along its path through the NoC.

Figure 3.11b restructures the mesh topology into a bipartite graph (with bidirectional edges) to point out this characteristics. As a result, at any point in time, two packets residing in nodes of different colour will never meet at a given switch. Consequently, any two packets can not affect each others switching decisions. If we collapse the bipartite graph into one virtual node for white and one for black, and further split up the bidirectional edges, we obtain an arrangement as shown in Figure 3.11c.

The colours of the nodes embody neighbouring *time/space slots*. A packet currently sitting on a given node could be seen in another network (i.e., a TDN) than its neighbour on the opposite node, therefore, any movement of two packets, which happens synchronously, involves a hop into another virtual network without the chance of collisions.

The contribution to the number of TDNs that stems from the topology is called the *topology factor*.

Number of Buffer Stages Buffers within the switches also create a set of TDNs. Figure 3.11c shows the collapsed topology graph with a stage for input (x_i) and output (x_o) buffers for white (w_y) and black (b_y) nodes. Each packet travelling trough the NoC visits buffers in the following cyclic order: $w_i \rightarrow w_o \rightarrow b_i \rightarrow b_o$.

Total Number As we have a set of colours (i.e., white and black) and buffer stages (i.e., input and output buffer), we can conclude the total number of TDNs in the following formula:

$$\text{TDN} = \text{topology factor} \times \text{number of buffer stages}$$

So, in the example above we have a total number of $2 \times 2 = 4$ TDNs, which also embodies the smallest round-trip delay (measured in clock cycles).

It is a clever policy to utilize each TDN for a specific type of communication, for instance different priority classes, traffic types, load conditions etc.

Looped Containers

In Nostrum, the LC is this concept that transports a Virtual Circuit (VC) and brings in bandwidth guarantees and latency bounds. The idea is that GB is achieved by having data loaded into container packets, which are *looped* between source and destination of communication. The principle is illustrated in Figure 3.12.

Figure 3.12 shows a round-trip delay of four clock cycles, whereas the switch attached to the source, an intermediate switch, and the switch attached to the destination are involved. In the first clock cycle, the container that is intended to

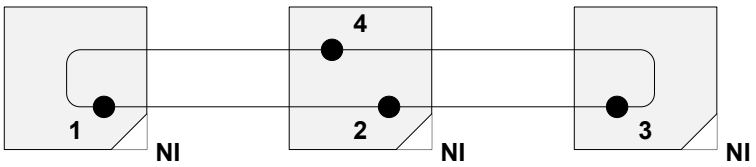


Figure 3.12: Looped Container

transport data arrives at the source switch. The source packs information into the container packet via its NI. In the second clock cycle, this container has moved on to the intermediate switch. Then, at cycle number three the container arrives at the destination switch, where the data is discharged. The container is not consumed at the destination, but passed on via the intermediate switch at clock cycle number four. Finally, the container completes the loop in the next clock cycle, which is regarded as "first" cycle again.

Combining TDNs and LCs

Nostrum combines LCs and TDNs to establish multiple VCs on a given switch. Each container of a LC occupies one TDN. Due to the movement of containers in each clock cycle, a given container always remains in the same TDN. Conversely, on each clock cycle, a container belonging to another TDN passes by at a given link of a switch.

A switch is able to process as many VCs at the same time as the number of TDNs, whereas each VC's containers reserve an own TDN. The TDN, which a VC is assigned to, remains constant during the routing through the NoC.

The route of a VC, i.e., LC and the TDN involved, is decided at design time, however the number of containers used by a VC is variable at run-time. There exists a possibility to increase or decrease the bandwidth of a VC by creating or destroying new containers in the LCs.

Multi-casting of VCs can easily be realized, if multiple destinations along the route of the VC grab the containers associated with that VC.

3.4.3 Power Management in Nostrum

Nostrum's main contribution to power management is the application of deflective routing, which enables the absence of internal buffer queues throughout the NoC. As switches do not include buffer memory, their power consumption is in the same magnitude as the power consumption of links between switches [PJ06].

Besides this, Nostrum features an *adaptive power management* [LJ06]. This mechanism is able to adjust frequency within a discrete range of frequency and voltage for the Nostrum system during live operation. This control mechanism, which is an autonomous subsystem of Nostrum, considers the difference between prediction

of network load and the saturation point (maximum load). If the network seems to become idle, frequency is scaled down immediately. If the network load seems to increase beyond a predefined threshold, the frequency is increased accordingly. Contrary, if the network load is predicted to drop beneath a predefined threshold, frequency is reduced in discrete steps. As a result, the network load justifies the current frequency, thus yielding into an optimal energy efficiency for all traffic patterns.

3.5 MANGO

The MANGO[2] [Bje05] architecture realizes a clockless NoC. It follows the Globally-Asynchronous Locally Synchronous (GALS) [MVK+99] approach, whereas IP-cores are locally synchronous (i.e., belong to some clock domain), but communication takes place asynchronously. Figure 3.13 illustrates a conceptual view on MANGO.

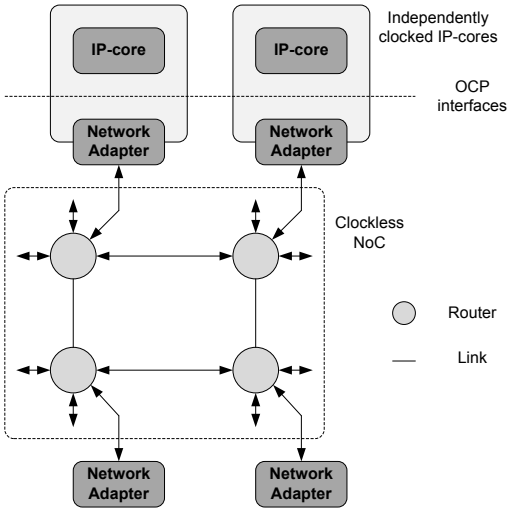


Figure 3.13: Overview of MANGO

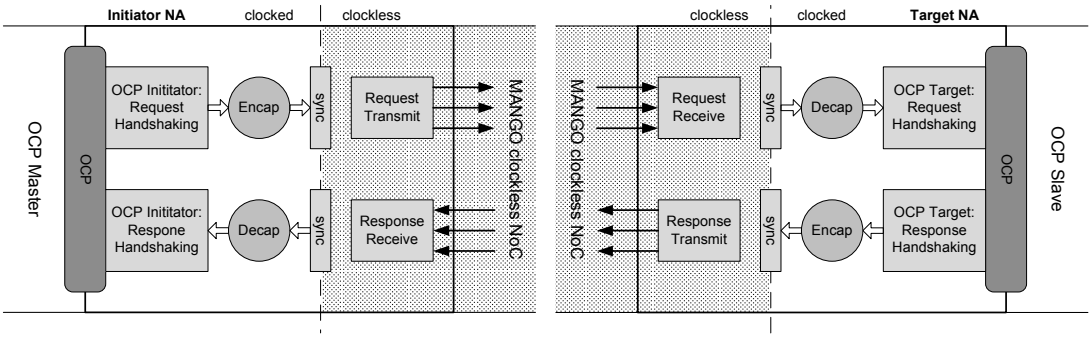
As we can see in Figure 3.13, the entities defined in MANGO comprise the Network Adapter (NA), the router, and the link.

3.5.1 Network Adapter

The NA [BMOS05] implements synchronization between the clocked IP-core and the asynchronous, clockless MANGO NoC. Furthermore, the NA offers a primitive message-passing functionality, which is realized in the NoC, by means of OCP transactions. That is, on the IP-core's side, the NA establishes a socket-based OCP interface, i.e., the Core Interface (CI), while at the NoC-side it realizes the Network

[2]Message-passing Asynchronous Network-on-Chip providing Guaranteed services through OCP interfaces

Interface (NI). The IP-core utilizes transaction handshaking according to OCP. The NI conducts *packetization* of the OCP transactions [BS06b] into packets that are switched through the NoC.



(a) Master-side Initiator (b) Slave-side Target

Figure 3.14: Types of NAs

As OCP follows the master/slave paradigm, the NAs and IP-cores must be design according to these roles. An IP-core operating as an OCP Master serves as an *initiator* for communication, while the OCP Slave IP-core is the *target*. Consequently, the NA of the initiator and the target include different modules. As shown in Figure 3.14, the request/response OCP channels are directly mapped to *request and response paths* through the NoC. The `Handshaking` and `Encap/Decap` modules in Figure 3.14a and Figure 3.14b convert the OCP transactions into packets and vice versa. `Transmit/Receive` modules handle the uni-directional, asynchronous communication channel through the NoC, after the `sync` modules have bridged the data between clocked and clockless zone.

3.5.2 Routers in MANGO

The routers [BS05a] in MANGO feature connection-oriented Guaranteed Service (GS) [BS06a] as well as connection-less Best Effort (BE) communication. So, there exists a dedicated GS router and BE router type.

The basic concept of communication is the Virtual Channel (Vc) [Dal92]. A Vc allows to maintain logically independent channels sharing one physical link. For GS, the Vcs apply *share-based* Vc control [BS04], whereas for BE a credit-based flow control is used.

MANGO routers provide P input and P output ports. Each output port includes a *buffer* for each Vc, hence each router is equipped with $P \times V$ buffers of length L, whereas V denotes the number of Vcs and L the length of each buffer. Usually, L has the length of exactly one flit, thus enabling to buffer exactly one flit before forwarding.

For GS, switching decisions are made on behalf of *steering bits*, which are locally *pre-programmed* for each Vc of each output port. The steering bits are distributed

via BE packets. A Vc can be understood as a segment of the overall route of a communication channel between NAs. The concatenation of Vcs, which is controlled by appropriate assignment of steering bits, manifests in the route of the communication channel. Contrary to pre-programmed steering bits, BE uses source routing.

In order to prevent that flits in the Vc buffers of a given output port collide, or stall and block each other, a *Vc control mechanism* must be applied. This ensures that a flit is only transmitted on the link attached to the output port, if and only if there is a free buffer space in the target Vc buffer in the next router. Therefore, the Vc control mechanism introduces *control channel bits* back to the previous router for each Vc buffer. When a flit is admitted to occupy the link towards the next router, the control channel bits indicate to lock a *sharebox* at the output port so that flits in other Vc buffers can not be forwarded. After the flit has passed by an *unsharebox* at the neighbouring router, that unsharebox triggers an "unlock" signal, which also belongs to the control channel bits. Consequently, the sharebox at the sending router is released, and the link is regarded as free. This mechanism not only results in collision avoidance between different Vc buffers of an output port, but also entails a *back pressure flow control*.

A *link arbiter* assigns the physical links of each output port to Vc buffers so that the total bandwidth is fairly shared between Vcs. For that purpose, the link arbiter implements the Asynchronous Latency Guarantee (ALG) algorithm [BS05b], which provides per-connection latency as well as bandwidth guarantees.

3.6 ×pipes

×pipes [BB04] [SAC+05] [MB06, section 6.10, pages 271 – 281] is a System C[3] library of parameterizable, synthesizable NoC components – so-called *soft macros* – optimized for low-latency and high-frequency operation. A soft macro is a template in the ×pipes architecture. Such a template can take a particular role in an ×pipes design. ×pipes features 3 different types of templates: switches, Network Interfaces (NIs), and links. Each of these entities is configurable and tunable at design time. Furthermore, templates can be arranged in arbitrary order, thus enabling a wide design space of possible topologies. Generally, ×pipes allows to apply configuration on a global scope, which are "network-specific", i.e., valid for the whole design, as well as on a local scope, which are "block-specific", i.e., valid for one or more templates from the library.

The basic communication protocol of ×pipes's NIs is OCP, which entails point-to-point communication. One of the main purposes of NIs is to adapt the semantics of the OCP interfaces to network level (internal) protocols. This network level protocol focuses on *packets*, which convey the data of OCP transactions. Therefore, NIs handle the *packetization* of OCP transactions. Also, a NI distinguishes between *header* and *payload*. Similar to MANGO, ×pipes takes the master/slave paradigm of OCP into

[3] `http://www.systemc.org`

account. That is, there exist NIs operating as *initiators*, and other NIs playing the role of *targets*, depending whether the attached IP-core is an OCP Master or OCP Slave.

×pipes applies source routing as the basic routing mechanism. Routes are statically stored in look-up tables. Routing information is included in the header before user data in the payload of a packet. The routing information consists of bits that indicate the direction, to which output port some incoming data at a given switch has to be forwarded.

Another aspect of ×pipes is its focus on communication reliability. An error detection mechanism, which is distributed among entities of the NoC, is performed to monitor the correctness of transmissions in the value domain. In case of errors, an error recovery technique becomes active. ×pipes uses a link level retransmission protocol such as GO-BACK-N to initiate retries of the faulty transmission. The distributed error detection has been preferred, as it allows to restrain the effects of error propagation. For instance, it would be possible to prevent a packet with corrupted header to be directed the wrong way in the NoC. On the one hand, such distributed error detection imposes penalty with respect to area overhead. On the other hand, error detection with retransmission is more convenient with respect to energy-per-bit efficiency than on-the-fly recovery. In the latter case, the error recovery circuitry must be active even during correct operation and therefore consumes energy all the time, while error detection with retransmission claims additional energy in case of errors.

The links between switches in ×pipes are pipelined. The links are subdivided into basic segments, whose length can be tailored to the system parameters such as desired clock frequency or driver strength. As a consequence, the input rate of data at links is no longer limited due to long propagation delays. In this context, the throughput of links is also increased owing to pipelining.

3.7 Comparison

In this section we compare the characteristics of the system architectures, which have been introduced above, to our best knowledge. The essence is summarized in Table 3.1. The interconnection fabrics, i.e., AMBA and OCP, possess a different scope than system architectures, and therefore can not fairly be compared with Æthereal, Nostrum, MANGO, ×pipes, and the TTSoC architecture. In the following we examine some characteristics with reference to the TTSoC architecture.

Multi-casting All system architectures except MANGO have identified the need for multi-casting. In fact, Æthereal does not support multi-casting natively. The shells in the NIs embody extensions to the basic services of the kernel, whereas a multi-casting shell groups several channels into one logical multi-cast channel. Nostrum benefits from the characteristics of LCs to route LCs to all receivers. ×pipes

Characteristic	Æthereal	Nostrum	MANGO	×pipes	TTSoC architecture
service classes	GT + BE	GB + BE	GS + BE	n/a	periodic and sporadic transportation
abstraction level	transactions	packets	OCP transactions		application messages
communication paradigm	shared memory	message passing	shared memory		message passing
guarantees	bandwidth + latency (for GT, GB, GS)			none	bandwidth + latency (for all service classes)
determinism	partially (with respect to GT, GB, GS)			n/a	by design
predictability	for GT	for GB	for GS	n/a	due to determinism
encapsulation	not explicitly by design				by design
error containment	not explicitly by design				due to encapsulation
message ordering:					
• single channel	temporal order	reordering	temporal order		temporal order
• multiple ch.	none				consistent delivery order
clock domains	one (highly synchronous)		clockless NoC	one	multiple + global time base
reconfigurability	static res. man.	+/- containers	programming Vcs	at design time	integrated resource management
reconfiguration	"soft cut"	unknown	by BE	n/a	"hard cut"
power management	unknown	adaptive	no dynamic power	n/a	aspect of resource management
buffers	required	avoided	required	required	conceptually none
transmission strategy	packet switching	LC	packet switching	packet switching	bursts of fragments of messages
arbitration mechanism	TDMA	TDN	ALG	n/a	time-triggered, periodic control system
flow control	credit-based	implicit	back pressure	GO-BACK-N	cooperating participants
routing strategy	selectable	deflective routing	steering bits (GS) source routing (BE)	source routing	source routing
topology	any	any	mesh preferred	any	any
multi-casting	as extension	arrangement of LC	n/a	supported	split-point multi-casting

Table 3.1: Comparison of characteristics

supports multi-casting be means of source routing. The TTSoC architecture also provides native multi-casting, as introduced in chapter 6, which is called *split point multi-casting*.

Arbitration Mechanism Considering the TDMA slot scheme and its *a priori* allocation, the Æthereal architecture is close to the TTSoC architecture, while Nostrum and MANGO facilitate more advanced arbitration mechanisms. While Æthereal incorporates a single global period with a fixed number of TDMA slots, the TTSoC architecture entails a periodic control system that features several concurrent periods with phase alignment within each period. The supported periods are in the range from a few nanoseconds up to seconds and can be adapted to special needs of a target application. Thus, the TTSoC architecture enables the temporal alignment of communication activities of periodic application jobs in order to reduce the end-to-end latency of the overall application. Such a feature is vital for many types of real-time systems. Additionally, the periodic control system is based on physical time, therefore periodicity of communication activities can be expressed in an intuitive way. This contributes to complexity reduction.

Buffers Nostrum and the TTSoC architecture are system architectures that do without buffers in switches, routers, i.e., in the whole NoC. Nostrum yields the properties of deflective routing, and the remarkably smart concepts of TDN and LC. By design, the TTSoC architecture relies on its *a priori* knowledge of communication patterns, which is leveraged to avoid congestions, and therefore relinquishes buffers.

Power Management With its clockless, asynchronous NoC, MANGO could be the most effective system architecture with respect to power management, as it cuts down on dynamic power consumption. However, the buffers in MANGO waste this conceptual advantage. Besides MANGO, Nostrum has presented many effective contributions to power management, as it includes voltage and frequency scaling in an autonomous subsystem of the NoC. Nevertheless, such adaptive power management is highly problematic, when it comes to predictability. Higher frequencies let communication arrive earlier at destinations, while slower frequencies increase the latency with respect to a reference time.

The TTSoC architecture features (active) power management as an aspect of the integrated resource management. Therefore, power management is an issue at architectural level. That is, power management can be established regarding the global view on the system, and is not solely restricted to components of the NoC. In this context, it also considers to uphold predictability and determinism of the system. Besides active power management, the most entities of the TTSoC architecture have been designed with power efficiency of realization in mind, as summarized in section 10.1.

Reconfigurability & Reconfiguration All system architecture address reconfigurability to a different extend. ×pipes is configurable and tunable at design time, however (to our knowledge) it does not deal with reconfiguration during live operation. The reprogramming of Vcs in MANGO as well as the creation or destruction of containers for LCs in Nostrum can be considered as reconfigurability. From the architectural point of view, Æthereal proposes a mechanism of on-the-fly reconfiguration of pre-defined channel configurations. We call the on-the-fly distribution of pre-defined configurations a "static resource management".

Channels can be installed, closed or existing channels can be modified during live operation. We call this a "soft cut", as the switch from one configuration to another is executed simultaneously, while the previous configuration is still operating. As a consequence, for some time there are two configuration coexisting in the system, which leaves an inconsistent system state during the process of reconfiguration. For instance, there might by application subsystem with a subset of communication channels already established, while the remainder of required communication channels is to be installed, yet. Consequently, that application subsystem is in an intermediate state, which has to be considered in the application design.

Contrary to Æthereal, the TTSoC architecture enforces "hard cuts" for the switch from one configuration to another one. As explained in chapter 8, a new configuration is first loaded into participants of communication, but it does not become active immediately. The switch happens synchronously among the whole system, i.e., synchronized by means of the Time-Triggered Communication Schedule. As a result, parts of different configurations do not coexist in the system during the reconfiguration process, and the system state remains consistent. We are convinced that the approach of "hard cuts" contributes to complexity reduction and minimizes the probability of design errors considering the reconfiguration.

Finally, in addition to static resource management the TTSoC architecture features a dynamic resource management, which is able to calculate new configurations at run-time, while it still upholds the consistency of the system state as well as predictability and determinism.

Clock Domains Unlike the TTSoC architecture, Æthereal and Nostrum support one single clock domain to dispatch and transmit data. As a consequence, all participants of communications (the NIs) must be synchronized to the same clock domain, which tends to by quite high to achieve throughput and performance. In contrast to this approach, the TTSoC architecture manages several clock domains by design. The global time base embodies an own clock domain, which is used to synchronize communication activities. The real communication takes place at another local system operation frequency. Thus, dispatching and communication are decoupled with respect to clock domains. There is no need to maintain one global clock signal of relatively high frequency in order to drive the whole system. Instead of this, the clock domain of the global time base that spans across the whole system design can be lower. This relaxes the requirements of synchronization within the system, and also aids to cope with capacity and driver strength issues.

A result of the introduction of multiple clock domains is the independence of communication from application design. We specify communication activities in a slower clock domain, which is easier to handle, while the participants of communication (i.e., the TISSs) are allowed to run at higher frequencies. As a result, tuning the communication subsystem, e.g., increasing the frequency associated with the clock domain of the global time base, does not produce any feedback to the application design. Conversely, if an application requires a lower granularity for the synchronization of communication activities, the global time base is modified, but application design is not affected.

Message Ordering All system architectures are able to handle temporal ordering of messages within a single communication channel. Nostrum might violate temporal ordering in case of packets taking different routes, as switching decision are made locally and (for BE) are affected by the current load distribution in the NoC. In fact, the TTSoC architecture might also reorder fragments of whole messages due to different routes. However, the TTSoC architecture abstracts from the realization of communication and solely presents completed messages to IP-cores, thus making the fact of reordering of fragments irrelevant.

The TTSoC architecture is the only one (to our best knowledge), which can handle a consistent delivery ordering of messages across different communication channels (see section 4.7).

Determinism & Predictability Æthereal, Nostrum, and MANGO provide guaranteed bandwidth and bounded latency, thus allowing a sense of predictability for this service class. However, none of them addresses determinism — according to the definition given in section 2.1.2 — like the TTSoC architecture. The TTSoC architecture derives predictability not from a special service class, but from the explicit support for determinism in its design.

Abstraction Level One aspect, where the TTSoC architecture definitely outperforms the other system architectures, is the level of design abstraction. While Æthereal, MANGO, and ×pipes deal with low-level packets and transactions, the TTSoC architecture abstracts from such communication issues and presents application-level messages to hosts. Nostrum also includes a notion of "messages" in the NoC, but these messages embody packetized chunks of application messages, and not completed application messages.

The high abstraction levels entails practical assumptions such as encapsulation, error containment, and message ordering. These features allow a system designer to focus on the application design and rely on the TTSoC's core services.

Service Classes The TTSoC architecture is a system architecture that does not include a notion of GT/GB/GS and BE. Instead of these service classes, it considers

periodic and sporadic application-level messages, which are transported via encapsulated communication channel. Each encapsulated communication channel always possesses quasi full bandwidth and exclusive access to the NoC, whereas the latency of transmission is also predictable. Consequently, for the application designer there is no need to care how to achieve a given bandwidth and latency, because this has already been guaranteed by the TTSoC architecture's core services. On the one hand, this relieves the application designer, as he/she has not to cope with such communication issues any longer. On the other hand, this is a totally novel design style to model on-chip communication in system architectures. While other system architectures establish evolution of well-known design styles, the TTSoC architecture facilitates a revolution in SoC design.

Other Features At its current state of development, the TTSoC architecture does not include error detection and retransmission like ×pipes. However, a service of reliable communication can be realized at a different level. Combining the diagnostic service and the on-the-fly reconfiguration, the TTSoC architecture is able to determine broken links or faulty switches in the TTNoC, and then compensates for an error by altering the routes and avoiding faulty switches. We thing that elevating such a service to a higher level than handling it in the NoC is more efficient and more flexible.

Chapter 4

Communication Service

We have learned from the previous sections that the TTSoC architecture provides a set of architectural *core services*. These are:

- an encapsulated, time-triggered communication service
- a diagnostic service
- integrated resource management with on-the-fly reconfiguration of system parameters, e.g., the Time-Triggered Communication Schedule
- clock synchronization to establish a consistent global time base that synchronizes communication activities

This chapter is dedicated to the core service of the encapsulated, time-triggered communication service. We introduce the architectural concepts that are used in the TTSoC architecture to model the communication between micro components.

4.1 Encapsulated Communication Channels

The basic architectural concept that is associated with the communication service is the *encapsulated communication channel*. The term "encapsulated communication channel" denotes an uni-directional connection. It transports *messages* at predefined points in time. Additionally, it supports exactly one source (sender) and at least one destination (receiver).

The endpoints of an encapsulated communication channel are called *ports*. We distinguish between output ports, which are located at the source (where the messages are produced), and input ports located at the destinations (where the messages are consumed).

4.1.1 Establishing Encapsulation

Encapsulated communication channels introduce *encapsulation* among micro components. Such encapsulation prevents interferences between communication activities of different micro components in two distinct domains:

temporal domain e.g., delaying a message or computation in another micro component

spatial domain e.g., overwriting a message produced by another micro component

However, encapsulation does not cover the value domain, for instance faulty input to other micro components via the sent messages.

In order to prevent any unintended interference between subsystems, the TTSoC architecture ensures such a *temporal and spatial partitioning* with respect to encapsulated communication channels. That is, communication activities in a given encapsulated communication channel are neither visible nor have any effect (e.g., performance penalty) on the exchange of messages in any other encapsulated communication channel. It is guaranteed that the micro component that is permitted to send messages over a given encapsulated communication channel is the only micro component, which is defined as the source of that encapsulated communication channel (i.e., the micro component where the output port of the encapsulated communication channel is located).

The TISS, which is located between the host of a micro component and the TTNoC, establishes encapsulated communication channels. Firstly, it enforces the temporal and spatial partitioning. Secondly, the TISS acts as a guardian for the shared TTNoC, where the encapsulated communication channels run through, by accessing the TTNoC exclusively at *a priori* known points in time according to the TDMA scheme. The allocation of TDMA slots manifests in the Time-Triggered Communication Schedule distributed among the TISSs. This allocation is explained in section 4.6.

In order to guarantee that the encapsulation properties of the communication service are not violated during the presence of a design fault or a hardware fault within the host, the implementation of the TISS ensures that the host cannot alter the Time-Triggered Communication Schedule in the TISS. As already mentioned in section 2.3.2, the TISS itself is a part of the TSS and is considered to by free of design faults.

4.1.2 Topology

A single micro component can be attached to several different encapsulated communication channels. But it is exactly once connected to a given encapsulated communication channel either in the role of the source or any destination. Consequently, a micro component might possess multiple input and output ports. However, by

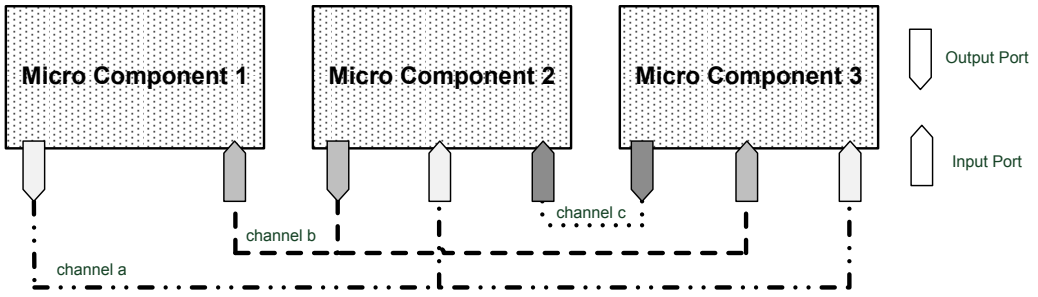


Figure 4.1: Example topology of encapsulated communication channels

design restriction, it is allowed to receive from or send to at most one encapsulated communication channel at a given instant of time.

The arrangement of destinations (among micro components) and the assignment of the source define the topology of an encapsulated communication channel. Since the number of destinations of an encapsulated communication channel is variable, *single-cast*,*multi-cast* and *broad-cast* modes are supported. Figure 4.1 illustrates encapsulated communication channels that spread across micro components.

4.2 Interface to the Communication Service

From the point of view of the hosts, communication is transparent, because the TTNoC is not directly accessible to the hosts in each micro component. Instead of this, the TISS provides an interface to hosts that includes this architectural core service – the UNI (see chapter 5). A host accesses the communication service via the ports of the encapsulated communication channels, which are visible within the scope of the UNI.

As mentioned above, we distinguish between input and output ports. In addition to their direction, we classify ports with respect to their access paradigm, which defines the way, how a host interacts with the corresponding encapsulated communication channel. In order to satisfy the need of a wide range of application domains, two basic types of ports are provided: *state ports* and *event ports*.

4.2.1 State Ports

State ports are used for the periodic transmission of messages with **state semantics**. A state port holds only a single state message at a time. Since state messages always contain the current state of a variable they are inherently idempotent. Old messages can be simply overwritten by new messages and – in contrast to event semantics of messages – exactly-once processing of the message is not required. In order to ensure that only consistent data are transmitted and received over the TTNoC, **explicit synchronization mechanisms** are provided for both (input state ports and output state ports) that coordinate update operations by the host and the TISS.

For input state ports, we apply the *Non-Blocking Write (NBW) protocol* [Kop97]. The Non-Blocking Write (NBW) protocol facilitates to detect any inconsistent read access by the host. It detects situations, where an input state port was updated by the TISS (due to the reception of a message), while the host was performing a read access on that port. In such a case, the host knows that it has acquired inconsistent data and retries the read access.

Hosts that are synchronized to the global time can use *implicit synchronization* to access input state ports as an alternative to the NBW protocol. Based on the *a priori* known points in time when the input state port is updated by the TISS, the host can temporally interleave its read accesses with the updates of the host in such a way that conflicts are avoided.

The NBW protocol can not be used for output state ports, since the TISS has to transmit messages over the TTNoC with minimal jitter. As a consequence, the TISS can not afford to retry a read access to a state output port, when the read access has resulted in inconsistent data due to a concurrent update by the host. Therefore, output state ports are synchronized by the use of a **double buffer**, where the host alternately updates one of the buffers, while the TISS accesses the other buffer.

4.2.2 Event Ports

Event ports are used for the sporadic transmission of messages with **event semantics**. Contrary to state ports, event ports have to support exactly-once semantics — each event message that has been sent to a set of receivers has to be received and processed exactly once by each receiver. In order to support exactly-once semantics, event ports are realized as **queues** that can hold multiple event messages at a time. The length of a queue is variable.

Event ports are synchronized by means of pointers for the write position and the read position of a given port. This is an explicit synchronization, implicit synchronization does not exist in this case.

The TISS indicates the presence of a new event message in an input event port by increasing the write position of the port *after* it has written the message into the corresponding position of the port. The host indicates the consumption of a message by increasing the read position of the port after it has consumed the message from the port. The reverse principle is used for output event ports.

4.3 Pulsed Data Streams

Encapsulated communication channels present a high-level abstraction of communication activities at a well-specified interface (the UNI) between host and TISS. Nevertheless, the real communication activity corresponds to the concept of *pulsed data streams* [Kop06]. The concept of pulsed data streams enables the temporal alignment of computation and communication (see section 2.3.3). Furthermore, it

established bandwidth guarantees for communication and deterministic end-to-end latencies.

A pulsed data stream is a time-triggered, periodic, uni-directional data stream that transports data in *pulses*. A pulse possesses a defined length (the so-called *duration*) and goes from exactly 1 sender to n according to *a priori* identified receivers at a specified *phase* of every cycle of a *periodic control system*.

The existence of a pulsed data stream as the communication primitive within encapsulated communication channels is abstracted, hence not visible to a host. In other words, an encapsulated communication channel houses exactly one pulsed data stream. Additionally, the topology of the encapsulated communication channel manifests in the arrangement of sender and receiver of the pulsed data stream. Besides this, the message of an encapsulated communication channel matches the pulse of an pulsed data stream. The length of the message conforms with the duration of the pulse.

4.3.1 Periodic Control System

In the TTSoC architecture the time format of the global time base is derived from the physical second. As illustrated in Figure 2.2, we implement the time format of the global time base as a binary counter with a fixed vector width of 64 bit. The upper 32 bit concern full seconds, while the lower 32 bit measure fractions of a second. As a result, the global time base has a horizon of about 2^{32} *sec* $\approx$ 136 years, whereas the maximum granularity goes down to 2^{-31} *sec* $\approx$ 465 *ps*.

From this time format we derive the periodic control system, which defines periods and phases of the pulses in a pulsed data stream. As the pulses in pulsed data streams realize the message of encapsulated communication channels, all communication activities within encapsulated communication channels are also aligned according to this particular interpretation of the time format.

Periods

The time format is used to measure the pulse periods and pulse phases of a pulsed data stream. For this purpose, we choose a specific bit in the counter vector of the time format, and consider its **toggle rate** as a period. This specific bit is called the *period bit* of the associated period. As the counter vector is binary, this leads to periods based on a power of two, i.e., a period can be 1 second, $\frac{1}{2}$ second, $\frac{1}{4}$ second, and so forth. This restriction leads to the establishment of harmonic periods, and entails a significant reduction of complexity of the scheduling in the TSS.

Figure 4.2 and Figure 4.3 show the distribution of period bits in the time format of the global time base for 16 and 32 periods. Note that the lower the period number the shorter is the period. In Figure 4.2 the period bit of the highest period #15 is placed at index 32, which conforms to a toogle rate or period of $2^1 = 2$ seconds. Then, period #14 has its period bit one bit to the right at index 31, which makes up a period of $2^0 = 1$ second, and so forth.

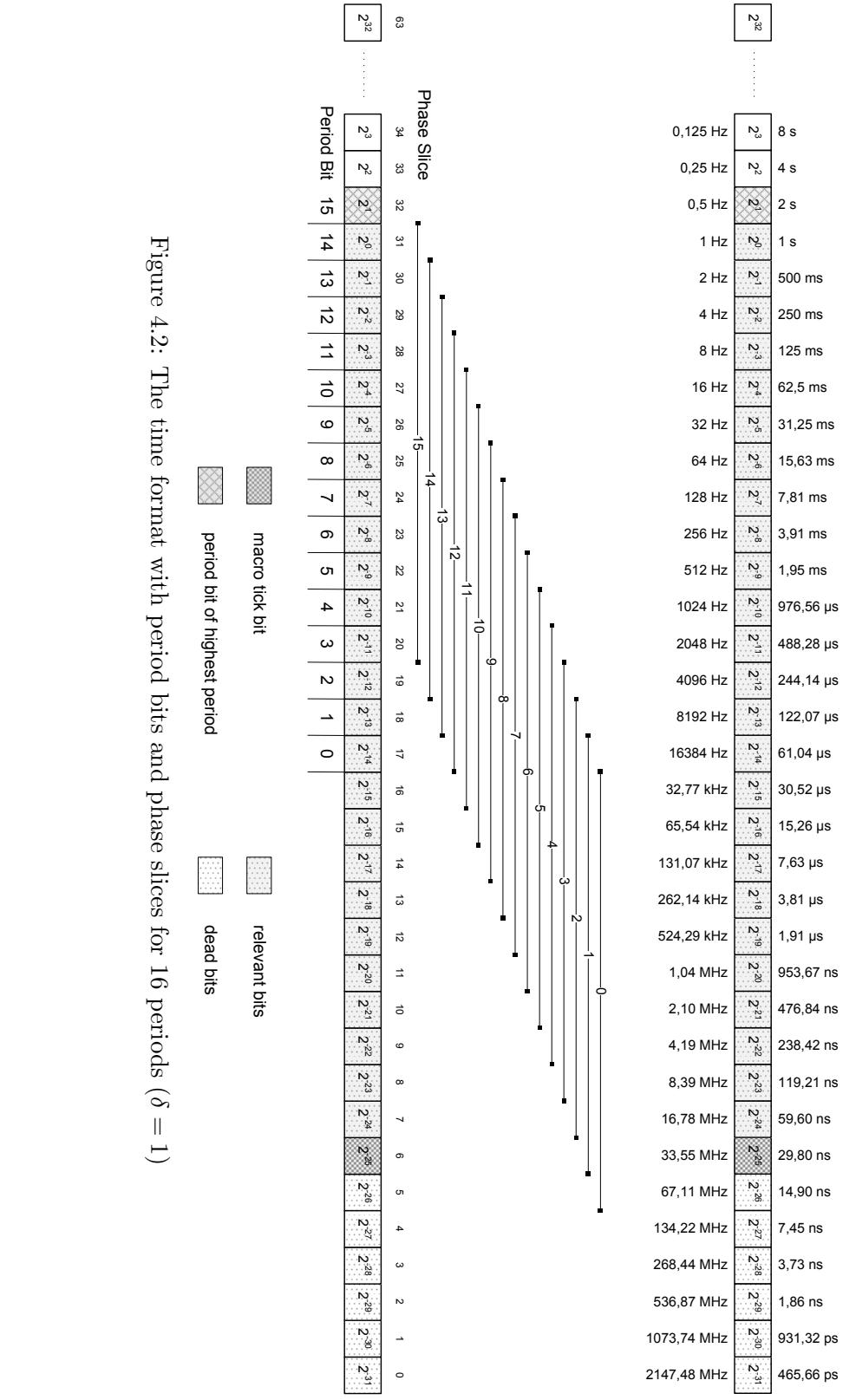


Figure 4.2: The time format with period bits and phase slices for 16 periods ($\delta = 1$)

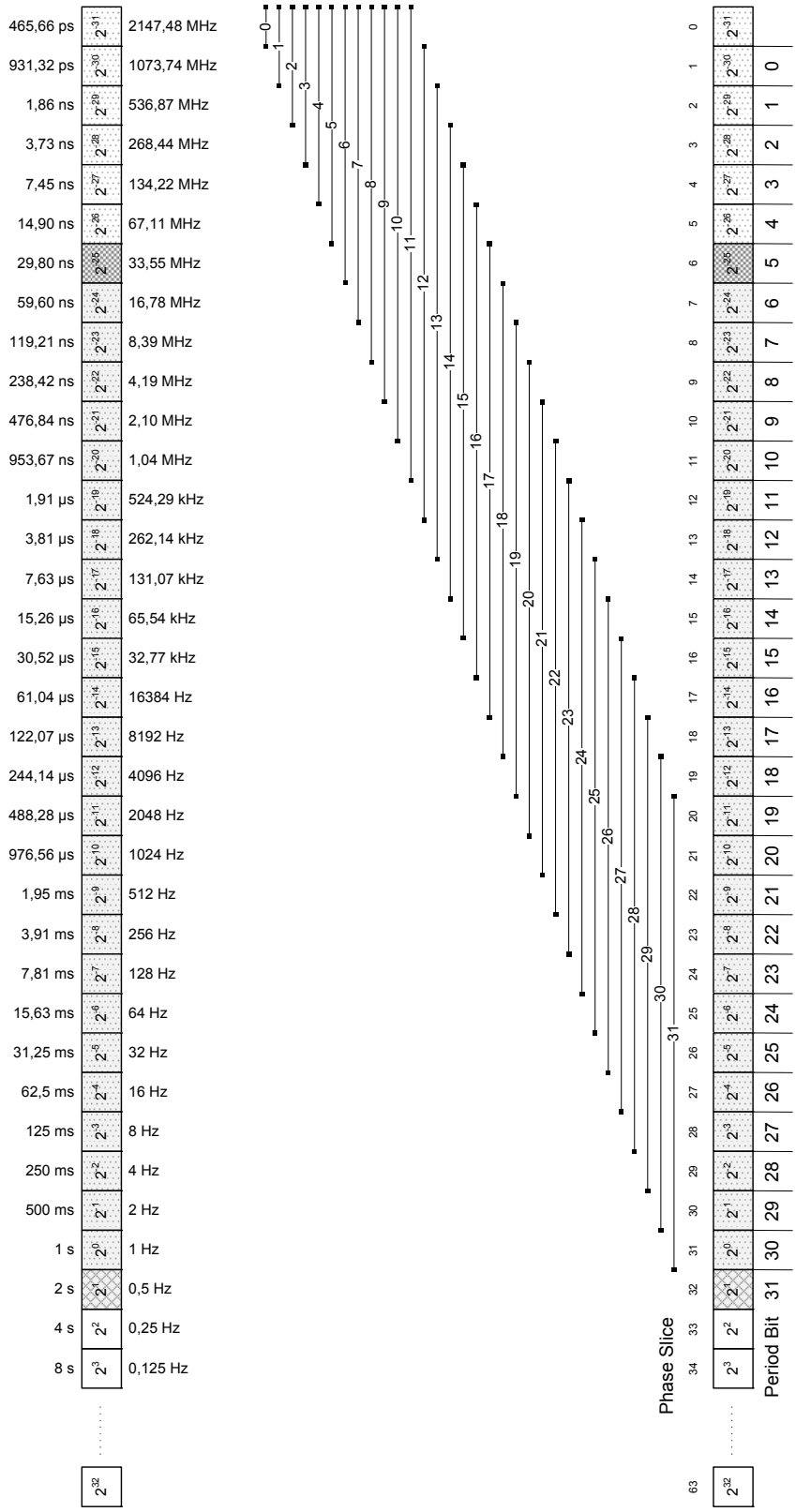


Figure 4.3: The time format with period bits and phase slices for 32 periods ($\delta = 1$)

Period Delta

We introduce the distance between two period bits as the *period delta* δ. In Figure 4.2 and Figure 4.3 $\delta = 1$, which is the default configuration.

The ratio $\mathcal{R}_p$ states, how many lower periods fit into a higher period. It is determined by the distance of the period bits, which can be extended by increasing period deltas. Generally, the ratio $\mathcal{R}_p$ between two periods A and B is expressed in equation 4.1.

$$\mathcal{R}_p(A, B) = 2^{\delta \cdot |A-B|} \tag{4.1}$$

For instance, with a period delta $\delta = 2$, which states that the period bits of two successive periods are 2 bits apart, period #15 would contain 4 periods #14 ($2^{2 \cdot |15-14|} = 4$), 16 periods #13 ($2^{2 \cdot |15-13|} = 16$) etc.

Phases

In general, the alignment of pulses to periods is too coarse. This inherently provokes collisions of pulses. Additionally, it does not allow to specify a sequence of pulses that belong to the same period. Therefore, the pulse phase further subdivides a period.

A phase defines the temporal offset of a pulse with respect to the start of an associated period. In the TTSoC architecture we implement a phase as a **slice in the counter vector** of the time format. Such a slice in the counter vector is named the *phase slice*. The phase slice is located one bit to the right of the period bit of that period. The width of the phase slice is configurable.

We define the start of a period as the periodic instant when all bits of the phase slice are 0.

Figure 4.2 and Figure 4.3 also illustrate phase slices with a width of 12 bit for 16 and 32 periods according to the time format.

Macro Tick and Granularity

It is not always reasonable to run the TSS at the finest granularity of the time format, which is the toggle rate of $465,66$ *ps* of the least significant bit in the counter vector of the time format (this matches a frequency of $2.147,48$ MHz). For such a situation we introduce the *macro tick*. The macro tick denotes the real granularity of the TSS, to which all communication activities are synchronized.

For the bits right of the bit associated with the macro tick (the "macro tick bit"), the time cannot be measured any more, because the duration of a level change of that bit is shorter than the real granularity of the TSS. For instance in Figure 4.2 the macro tick bit is located at index 6 (2^{-25} $sec \approx 29,80$ $ns \Leftrightarrow 33,55$ MHz). Consequently, these "dead bits" remain 0 all the time. However, they are still included

in the time format. In case of better target technology that achieves better system clock frequencies, the macro tick bit moves to the right and activates bits of the time format that have been regarded as dead just before.

The macro tick determines the frequency f_{mt}, with which the counter vector of the time format of the global time base is incremented. On each "tick", i.e., the rising edge of the macro tick signal, we add '1' to the counter vector at the macro tick bit.

Furthermore, the macro tick also limits the *local granularity* of phases of periods. If the phase slice crosses the macro tick bit, the bits right of the macro tick bit are made up of dead bits. So, they are stuck at zero. Consequently, this fact constrains the local granularity of the phase for that period. Additionally, the phase is also cut off in case of the phase slice reaching the lower bound (least significant bit of the counter vector) of the time format. As a result, there might occur periods with phases, which are so much limited that these periods are not useful any more. Such periods should automatically be disabled by the TNA (see section 7.3.3).

In general, the local granularity of a phase associated with a given period is determined by the least significant bit of its phase slice. For instance in Figure 4.2, for period #15 this is bit 20, which corresponds to a toggle rate, hence local granularity, of $2^{-11}\ sec \approx 488\ \mu s$. For a given phase slice width, i.e., in the current implementation this width is 12 bit, we can define $2^{12} = 4096$ different offsets from the beginning of the period, which for period #15 are 488 μs apart.

Obviously, the finest local granularity of any period complies with the macro tick, if its phase slice exceeds the macro tick bit.

Tuning the Periodic Control System

In the TTSoC architecture the periodic control system can be tuned by means of parameters concerning the time format, which are configurable at design time. Each instance of the TTSoC architecture is able to modify

- the number of supported periods
- the index of the most significant bit of the highest period
- the period delta
- the width of phase slices
- the macro tick bit

at design time. On-the-fly configurations (at run-time) of the time format are not supported.

The "window" of period bits and phase slices can be moved or stretched and squeezed along the time format by configuring its parameters. As a consequence,

this adds a certain degree of flexibility, so that the time format can be adapted to the requirements of a specific application, for which the TTSoC is intended. For the current implementation as illustrated in Figure 4.2 and Figure 4.3, we expect that the highest useful period for applications is 2 seconds. So, the MSB of the highest period is assigned at bit 32, which corresponds to 2^1 seconds.

4.3.2 Handling Collisions of Pulses

When defining pulses in different periods, a scenario might occur when a pulse of a lower period interferes with a pulse of a higher period, even though their phases apparently do not coincide. With respect to repetition of pulses due to the periodicity, the pulse of the lower period occurs $\mathcal{R}_p$ - times in the higher period. Consequently, it is possible that a repetition of the pulse of the lower period collides with the pulse of the higher period.

However, the TTSoC architecture enforces the restriction that the specification of pulses (with their periods and phases) is free of collisions.

To avoid conflicts with periods and phases of pulsed data streams in the TTSoC architecture, communication is determined *a priori* before deployment (see section 2.3.3).

Not all periods need to be active. For this purpose, the TTSoC architecture provides a feature to selectively disable periods in order to save energy (see section 7.3.3).

4.4 Realization of Pulsed Data Streams

As illustrated in Figure 4.4, a pulse of a pulsed data stream consists of at least one *fragment* of variable size. Successive fragments of one pulse are not required to be transmitted in a dense sequence on the TTNoC. This enables the *interleaving of fragments* of different pulses, and thus supports the concurrent transmission of multiple pulses, i.e., of different pulsed data streams, over a shared resource such as single physical link. The duration of a pulse assigns the time between the start of the transmission of the first fragment and the end of the transmission of the last fragment.

Figure 4.4 also depicts the allocation of TDMA slots for a pulsed data stream, which consists of three fragments. The free TDMA slots between the last two fragments could be used by fragments of other pulsed data streams.

The fragmentation of pulses into fragments is not visible for the hosts, i.e., outside the TSS. Thus, the DASs in the hosts only deal with completed application-level messages of encapsulated communication channels, and need not concern about the realization by pulsed data streams.

While fragments are not visible to the hosts, from the point of view of the TSS a fragment is further decomposed into a set of atomic, fix-sized *flits*. A flit is the

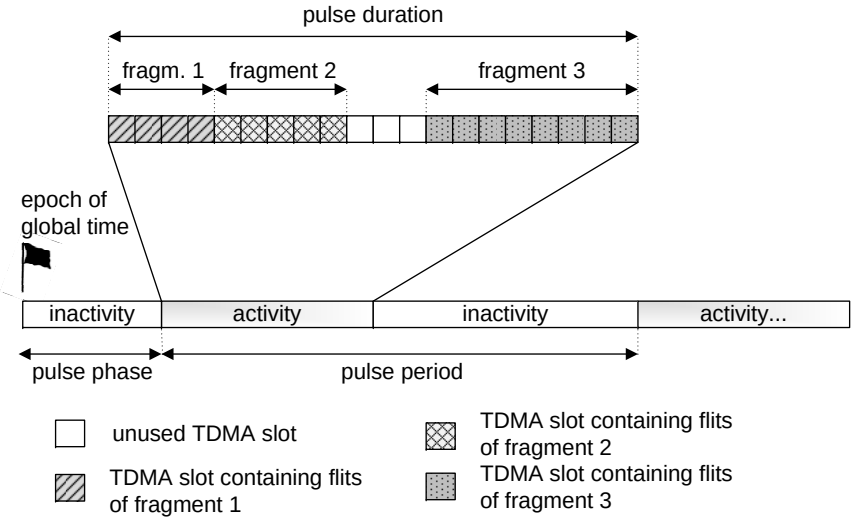


Figure 4.4: Example of an pulsed data stream

basic entity that can be transmitted over the TTNoC within one clock cycle of the system operation frequency f_{sys}, which drives the TSS. A TDMA slot can contain several flits, depending on the ratio $\mathcal{R}_f$ between the frequency f_{mt} associated with the macro tick, which determines the length of a TDMA slot, and the system operation frequency f_{sys} of the TSS. Equation 4.2 expresses this relation between the two frequencies:

$$\mathcal{R}_f = \frac{f_{sys}}{f_{mt}} \geq 1 \tag{4.2}$$

The constraint $\mathcal{R}_f \geq 1$ follows from the fact that (usually) the global granularity of the global time base is coarser than the granularity associated with the system operation frequency, as first mentioned in section 2.1.4.

Each pulse originates in the TISS of the sending micro component and terminates in the TISS of the receiving micro components. Considering the fragmentation of pulses into fragments, a sending TISS generates a sequence of fragments that make up the pulse, which are reassembled at the receiving TISSs. This process is transparent for the host. The transmission instant as well as the receiving instant of each fragment is determined *a priori* according to the Time-Triggered Communication Schedule. The basic mode of operation of the TISS is a *burst*. That is, the TISS transmits or receives a complete fragment of a message flit-by-flit, processing one flit in each clock cycle of the system operation frequency without interruption.

4.5 Pulse Interleaving

The sequence of fragments that make up a pulse need not necessarily be dense. There can be an interval of idle TDMA slots. Besides this, fragments of other pulses

are allowed to interleave the sequence of a specific pulse. This feature named *pulse interleaving* enables the **coexistence of several pulses and conflict resolution among shared resources** of the TTNoC at the same instant of time. Again, this is abstracted from the host, hence a special feature of the TSS.

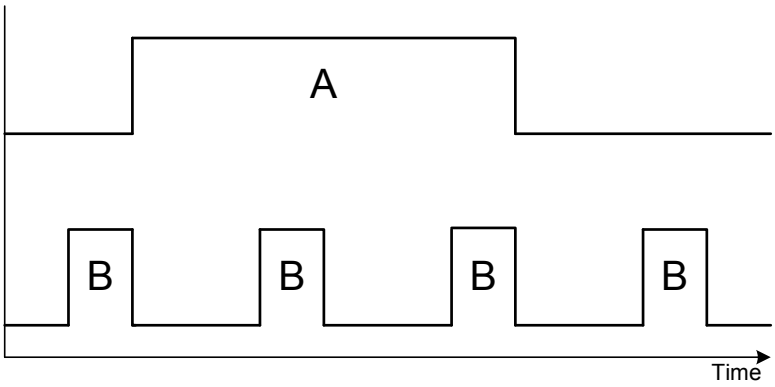


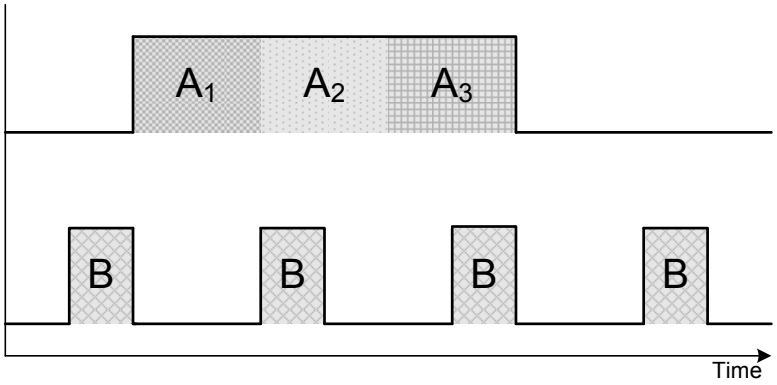
Figure 4.5: Two conflicting pulses

An example for such a scenario is a DAS at one host that produces two pulsed data streams, as illustrated in Figure 4.5. Pulse A has a longer period, but each pulse conveys a large message. Pulse B has a shorter period, but it just transports a small message. In addition to this we assume that shorter periods have the higher priority, as deadlines are not to be missed at all. Because both pulses share the same physical resources, i.e., the same TISS and its TTNoC interface, they are conflicting. So, pulse interleaving offers the mechanism to split pulse A into smaller sized fragments (see Figure 4.6a), and interleave the fragments of pulse A and B so that no conflict arises any more.

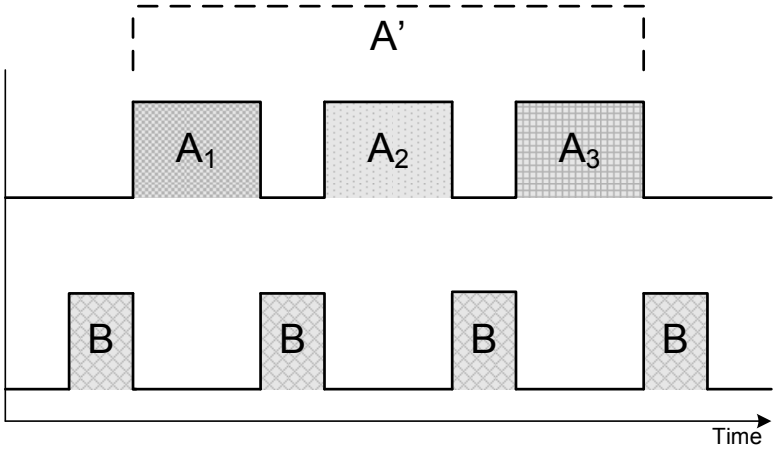
Note that the pulse interleaving **extends the duration** of the split-up pulse. As Figure 4.6b demonstrates, pulse A has longer duration A' than before the interleaving. However, this is the only means to handle the conflicting situation between both pulses. Considering the long period of pulse A, the longer duration A' of the pulse does not impose any penalty. Conversely, it is important that the pulsed data stream, where pulse B belongs to, obtains the bandwidth for its short periodic pulses.

4.6 Allocation of Bandwidth

The TTSoC architecture uses a time format for the global time base that is based on the physical second, as depicted in Fig. 2.2. Fractions of a second are represented as 32 negative powers of two (down to about 465 picoseconds), and the full seconds are presented in 32 positive powers of two (up to about 136 years). The periods and phases of pulsed data streams directly match to this time format, which is closely related to the time format of the GPS time, and takes the epoch from GPS.



(a) Splitting pulse A into smaller fragments



(b) Interleaved pulses

Figure 4.6: Pulse Interleaving

Section 4.3.1 has introduced the macro tick and its associated bit (the "macro tick bit") in the time format. The macro tick defines the granularity, with which communication activities are synchronized. **The interval between two consecutive macro ticks corresponds to the duration of a TDMA slot.** TDMA slots might contain several flits, depending on the ratio $\mathcal{R}_f$ between duration of the TDMA slot (i.e., macro tick) and the TSS's system operation frequency. In each clock cycle of the system operation frequency[1], one flit is processed.

TDMA slots are allocated *a priori* to the micro component that sends the pulsed data stream, respectively is the source of the associated encapsulated communication channel. This allocation manifests in the Time-Triggered Communication Schedule contained in each TISS. This schedule indicates the instants, when a fragment of a pulse is to be sent or received in a burst. All communication activities listed in

[1] sometimes also referred to as *micro tick* [Kop97]

the Time-Triggered Communication Schedule are aligned to the macro tick. In this context, the TISSs are synchronized by means of the macro tick.

A TDMA slot entails a specific bandwidth depending on the number of flits contained in a slot times the size of a flit. The sum of bandwidths of the TDMA slots reserved for an encapsulated communication channel devotes the total guaranteed bandwidth for that encapsulated communication channel.

The usage of reserved TDMA slots is not mandatory. For instance, an encapsulated communication channel with event semantics might not have a new message pending for transmission. Then, the TDMA slots are reserved for the sending TISS, however this TISS does not inject data into the TTNoC. In any case, a dynamic reallocation of the unused TDMA slots to other encapsulated communication channels is not supported. This relaxation of requirements, which best suits to the characteristics of event semantics, allows *sporadic messages* besides the *periodic messages*, which is usually associated with state semantics.

Note that the TTNoC as well as the whole TSS is able to run at an system operation frequency that is a multiple (not necessarily a power of two) of the corresponding frequency of the macro tick. Additionally, the synchronization of communication activities at the TISSs is decoupled from the transmission over the TTNoC by design. As a consequence, the specification of communication activities (in the Time-Triggered Communication Schedule), which is an integral part of application design, and the operation of the TSS reside in different clock domains.

4.7 Message Ordering

For a given encapsulated communication channel, due to different distances from a sender to different receivers it is inevitable that data probably arrives at different instants of time at each receiver. For instance, such multi-path propagation delays are a natural consequence of multi-casting in a non-bus topology of the underlying TTNoC.

It is a part of the communication core service of the TTSoC architecture to provide two types of guarantees with respect to message ordering [Sal08].

4.7.1 Total Temporal Ordering

Within a single encapsulated communication channel message ordering is inherent. The sending TISS might partition a message into fragments, which are allowed to take different routes each. However, a specific fragment belonging to this message periodically takes the same route. Due to different lengths of routes it is possible for the ordering of receiving fragments to deviate from the ordering of sending. Nevertheless, such an inconsistent reception order of fragments is not visible to the micro component, since it reads only completely received messages, which have been reassembled into proper order within the TISS.

Note, that the property of temporal ordering is essential to interpret serial data, i.e., a sequence of messages in a single encapsulated communication channel.

4.7.2 Consistent Delivery Order

A consistent delivery order among several encapsulated communication channels is mandatory in order to establish *replica determinism* [Pol93] for micro components, which is a requirement to transparently mask hardware errors. Consistent delivery order denotes the property that any two micro components see the same sequence of message receptions among a set of encapsulated communication channels.

Multi-path propagation delays, particularly with multi-casting, can cause the following situation. At two encapsulated communication channels α and β with multiple receivers a given micro component A receives the last fragments f_α and f_β of the corresponding encapsulated communication channels within a short interval of time. Micro component B is also attached to α and β as a receiver. The route of α is longer than β for A, but from the point of view of B the route of α is shorter than β. As a result, A notices the instant of arrivals $t(f_\alpha) > t(f_\beta)$ for the fragments, whereas B records $t(f_\alpha) < t(f_\beta)$. The arrival of the last fragment derives the completion of the message. Consequently, A has $\alpha > \beta$, but B has $\alpha < \beta$. This inconsistency is unacceptable.

Dealing with Multi-path Propagation Delays

The TTSoC architecture inherently supports a mechanism in order to avoid the violation of the property of consistent delivery order. If consistent delivery order is required for a set of encapsulated communication channels the reception of the last fragment f_l of a message can virtually be delayed at each TISS to a common instant $t_c \geq \max t(f_l)$. After the last fragment has been completely received by all micro components, all TISSs dispatch a *virtual fragment* at that common instant that causes the associated message to be declared as complete at all TISSs simultaneously. This virtual fragment does not contain any data, it is just a place holder to trigger a receive instant at receiving TISSs. There are no flits transmitted over the TTNoC.

The instant of the virtual fragment is synchronized by means of the global time base. Additionally, this instant is *a priori* known and is therefore included in the Time-Triggered Communication Schedule among the TISSs of all participating micro components.

Note that the mechanism of the virtual fragment is transparent to the TTNoC. Furthermore, it seamlessly integrates into the concept of time-triggered communication within the TISS. Consequently, its realization does not require any additional semantics and control logic neither in the TISS nor in the TTNoC.

Chapter 5

The Uniform Network Interface

This chapter presents the specification of the Uniform Network Interface (UNI), which resides on top of the TISS in the conceptual structure of the implementation, as introduced in Figure 2.4. The purpose of the UNI is to abstract from the implementation of the TSS, so that from the point of view of the host the architectural core services of the TTSoC architecture are accessible via a **memory-mapped interface**.

The UNI consists of two parts:

Port Interface The Port Interface offers access for the TISS to the physical memory, where all application data of ports that are sent or received by the DASs is stored – the Port Memory.

Control Interface The Control Interface is used by the host for configuration and synchronization of architectural core services.

Physically, both interfaces conform to the low-level signal specification as defined by the OCP standard. The choice for OCP shall encourage the industry to widely accept the TTSoC architecture.

In the following we comprehensively describe the Port Interface and Control Interface as memory-mapped interfaces.

5.1 Port Interface

The purpose of the Port Interface is to grant access for the TISS to the ports residing in the physical memory, which hosts the application data transported through encapsulated communication channels – the Port Memory. Transactions over the Port Interface are solely initiated by the TISS itself according to the Time-Triggered Communication Schedule. Therefore, the Port Interface is realized as an *OCP Master* on the TISS-side. Symmetrically, at the host-side (i.e., at the bottom-end of the Front-End) it is an OCP Slave.

Moreover, by design the Port Interface is **required to convey a data word in every clock cycle of the system operation frequency**. Note that this is not a special feature to increase the performance, but mandatory to keep pace with the basic operation of Fragment Switches and TISSs, which is a burst. As a result, it is possible to fetch and store the data words of a port without discontinuation, and further send to and receive from the TTNoC interface without interruption. Otherwise, it would be necessary to stall the burst of Fragment Switches and TISSs and introduce buffers.

5.1.1 Signal Specification

Table 5.1 covers the physical signals that make up the Port Interface. The first column contains the name of the signal as it appears in the source code. Note that for the Port Interface all signals have the prefix `OCPM_` ("**OCP M**aster"), whereas the remaining part of the name refers to the convention defined in the OCP standard. The second column of that table gives the width of the signal in the current implementation. The column "Driver" informs, which entity sets the level of the corresponding signal. For instance, if the driver is "TISS", this means that the signal is set at the TISS-side of the Port Interface. On the other (host-) side, the signal is just read. For "host", the signal is driven by the host-side, i.e., the Front-End, of the Port Interface, whereas the TISS consumes the signal value. The last column gives a short description of the signal.

Name	Width	Driver	Function
`OCPM_MAddr`	16 bit	TISS	address of the data word in the Port Memory
`OCPM_MCmd`	3 bit	TISS	command code of the current operation
`OCPM_MData`	32 bit	TISS	write data; from TISS to Port Memory
`OCPM_SData`	32 bit	host	read data; from Port Memory to the TISS
`OCPM_SError`	1 bit	host	error status

Table 5.1: OCP signals of the Port Interface

In the following we give more detailed information about the function of each signal.

`OCPM_MAddr` This signal, which is actually a vector of 16 bit in the current implementation, specifies the address of the current read or write transfer. This address refers to a memory location in the Port Memory. Note that the OCP specification requires addresses to be **word aligned**. With a width of 32 bit at the data buses (`OCPM_MData` as well as `OCPM_SData`), we have to add 2 additional bits to

the vector OCPM_MAddr. Consequently, the 2 least significant bits are hard-wired to 0, and actually $16-2=14$ bits of this vector address distinct data words (of 32 bit width) in the Port Memory. In the current implementation, this gives as a maximum size of the Port Memory of $2^{14} \cdot 32$ bit = 512 Kbit = 64 KB.

OCPM_MCmd According to the OCP standard, this signal indicates the type of transfer that currently runs through the Port Interface. The standard dictates a width of 3 bit, even though the current implementation supports only 3 out of all possible OCP commands. The supported OCP commands are listed in Table 5.2.

OCPM_MCmd[2:0]	Command
000	idle
001	write
010	read

Table 5.2: Currently supported OCP commands at the Port Interface

OCPM_MData This signal carries the actual application data, which is to be written into the Port Memory. The width of this signal, i.e., this vector, by design **equals the width of the data bus of a lane in the TTNoC**, which is 32 bit in the current implementation. Thus, application data flits can be collected from the TTNoC interface and directly forwarded through the Port Interface to the Port Memory without any wrapping or processing.

OCPM_SData Like OCPM_MData, but inverse direction. This signal conveys read data from the Port Memory. Due to its width that equals the width of the data bus of a lane in the TTNoC, again we can directly forward the application data to the TTNoC interface without any wrapping or processing.

OCPM_SError Over this signal the Port Memory respectively the Front-End is allowed to report error conditions to the TISS. This signal is processed by the error status logic of the TISS, and its state is accessible through the Control Interface. For instance, an error condition arises, when the address at OCPM_MAddr exceeds the address range of the real Port Memory[1]. Furthermore, note that the addresses of ports are outside the TSS, as the Port Memory residing in the Front-End does not belong to the TSS. Consequently, it is possible that addresses can be erroneous. Nevertheless, this does not violate the temporal behaviour of the TSS, nor it breaks up the encapsulation of encapsulated communication channels.

[1] assuming that the physical memory of the Port Memory is smaller than the maximum addressable space of the Port Interface

5.1.2 Memory Layout of State and Event Ports

This section explains the memory layout of state and event ports, which reside in the Port Memory. For details about the synchronization protocol established for ports refer to section 5.3.

As mentioned before, the width of data words in the Port Memory and the width of the data bus of the lane in the TTNoC are the same. As a result, the TISS is enabled to store into or fetch from the Port Memory in each clock cycle of the system operation frequency without any wrapping or further processing.

State Ports

Generally, the memory layout of a state port depends on the **message size in number of data words / flits**, the **synchronization method** (explicit or implicit), and the optional activation of the **time stamping service** (see section 5.4.1) for that port.

Figure 5.1 shows the memory layout for an output state port of size N data words. Apparently, this port uses the explicit synchronization. Namely, the reserved memory area is doubled, hence, it requires $2 \cdot N$ data words in the Port Memory. The double buffer is the means of explicit synchronization. It allows to establish a *shadow buffer*. For details about synchronization of state ports refer to section 5.3.1.

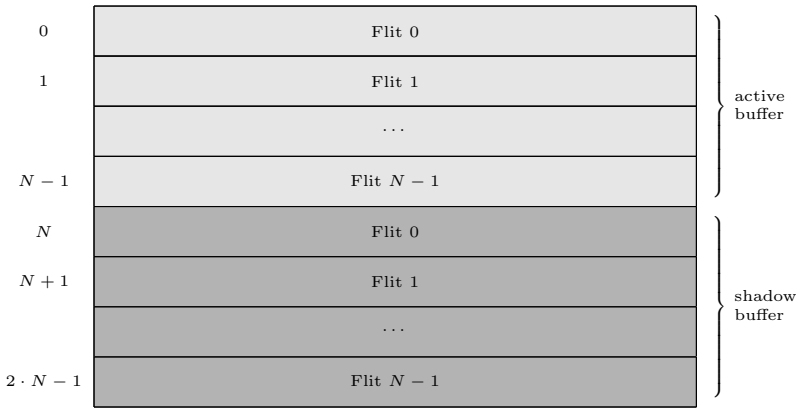


Figure 5.1: Memory layout of an output state port with explicit synchronization

In contrast, an output state port with implicit synchronization consumes only N data words, as it does not include shadow buffers. However, input state ports support the time stamping service. If the time stamping service is activated for a given port, the memory layout increases to $N + 2$ data words due to the additional 64 bits required for storing the time stamp. As we can see in Figure 5.2, the time stamp is placed in front of application data of the port.

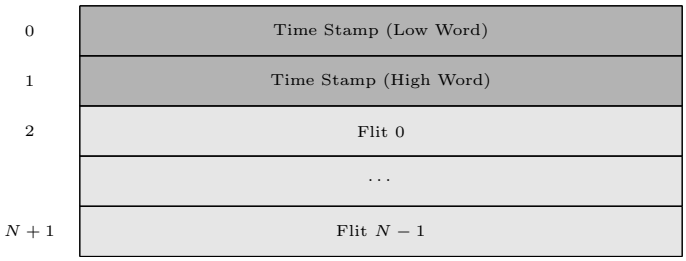


Figure 5.2: Memory layout of an input state port with time stamps

Event Ports

Figure 5.3 depicts an event port in the Port Memory. The size of a single (event) message is given as N, the number of messages in the event queue is denoted as L. Moreover, Figure 5.3 shows the usage of the time stamping service.

The amount of allocated memory is determined by the **message size in number of data words / flits**, the **maximum number of messages in the queue**, and the optional activation of the **time stamping service** for input event ports only. As a result, an event port consumes $L \cdot (N + 2)$ data words in the Port Memory in case of an activated time stamping service. Otherwise, without time stamping service the event port reserves just $L \cdot N$ data words.

For details about the synchronization of event ports refer to section 5.3.2.

5.2 Control Interface

The second part of the UNI is the Control Interface. It is used for configuration of individual ports and synchronization of the access to messages in ports between TISS and host. Unlike the Port Interface, the Control Interface is realized as an *OCP Slave* from the point of view of the TISS. Symmetrically, at the host-side it is made up of the according OCP Master.

Consequently, the sphere of control at the Control Interface remains with the host. That is, the host determines the access to the Control Interface. As a result, the access at the Control Interface is not time-triggered. Indeed, the Control Interface is not time-critical. Additionally, it is not required to convey a data word in every clock cycle of the system operation frequency. In other words, the Control Interface is also not performance critical, as it does not need to go with a burst.

Furthermore, the Control Interface has a fixed width of data words, which is 32 bit in the current implementation. In comparison, the Port Interface always possesses the same width of data words as the width of the data bus of a lane in the TTNoC. In the current implementation, the data width of Control Interface and Port Interface are the same by coincidence.

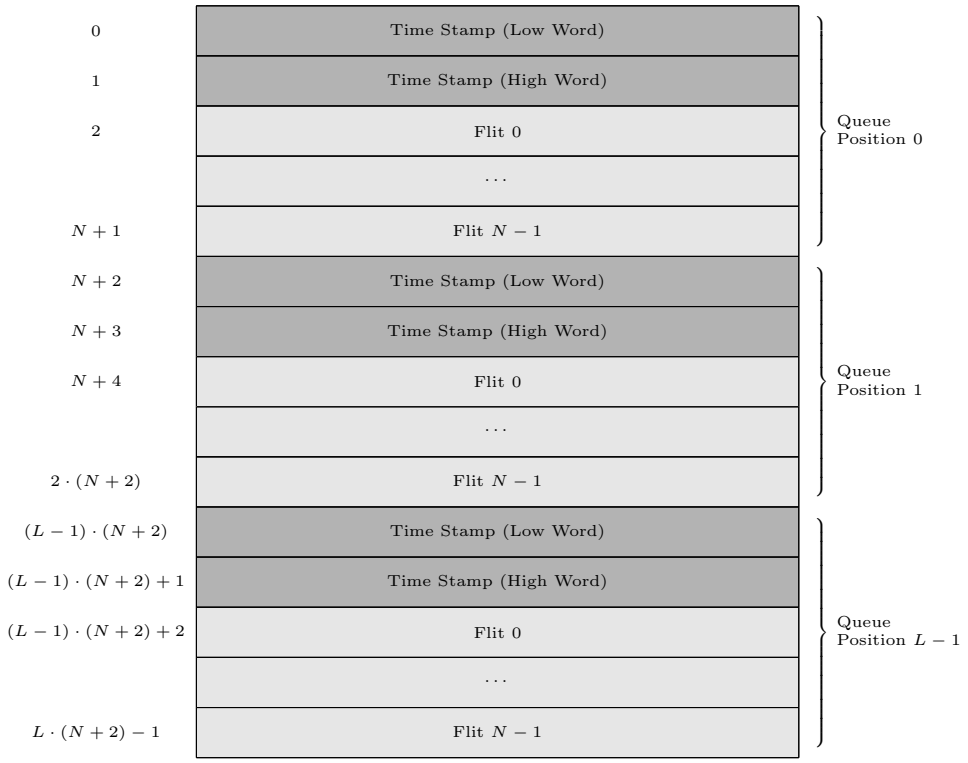


Figure 5.3: Memory layout of an event port with time stamps

5.2.1 Signal Specification

Table 5.3 covers the physical signals that make up the Control Interface. The first column contains the name of the signal as it appears in the source code. Note that for the Control Interface all signals have the prefix `OCPS_` ("**OCP S**lave"), whereas the remaining part of the name refers to the convention defined in the OCP standard. The second column of that table gives the width of the signal. The column "Driver" informs, which entity sets the level of the corresponding signal. For instance, if the driver is "TISS", this means that the signal is set at the TISS-side of the Control Interface. On the other (host-) side, the signal is just read. For "host", the signal is driven by the host-side, i.e., the Front-End, of the Control Interface, whereas the TISS consumes the signal value. The last column gives a short description of the signal.

In the following we give more detailed information about the function of each signal.

`OCPS_MAddr` This signal, which is actually a vector of 9 bit in the current implementation, specifies the address of the current read or write transfer. This address refers to a memory location in the name space given by `OCPS_MAddrSpace`. Like `OCPM_MAddr` of the Port Interface, `OCPS_MAddr` has to append 2 surplus address

Name	Width	Driver	Function
OCPS_MAddr	9 bit	host	address of the data word in the current name space
OCPS_MAddrSpace	2 bit	host	current name space
OCPS_MCmd	3 bit	host	command code of the current operation
OCPS_MData	32 bit	host	write data; from host to the memory given by OCPS_MAddrSpace
OCPS_MByteEn	4 bit	host	byte enable mask for write data
OCPS_MRespAccept	1 bit	host	handshake signal; host accepts current response
OCPS_SCmdAccept	1 bit	TISS	handshake signal; TISS accepts current command
OCPS_SData	32 bit	TISS	read data; from the memory given by OCPS_MAddrSpace to the host
OCPS_SResp	1 bit	TISS	handshake signal; status code of the response
OCPS_SError	1 bit	TISS	error status
OCPS_SFlag	10 bit	TISS	additional status information, e.g, interrupt flags
OCPS_SReset_n	1 bit	TISS	force the host's CPU to perform a reset

Table 5.3: OCP signals of the Control Interface

wires (due to the fixed data with of 32 bit) in order to conform to the OCP standard (with respect to word alignment of addresses). As a result, at the Control Interface just $9 - 2 = 7$ bits are available to address data words in the memory of a given name space. Hence, $2^7 = 128$ different data words can be addressed in each name space.

`OCPS_MAddrSpace` The Control Interface contains a concept of *name spaces*. On the one hand, these name spaces extend the maximum address range. On the other hand, name spaces logically separate different entities of the Control Interface. Each name space is associated with one entity, which is accessible from the host via the Control Interface. These entities are the Port Configuration Memory (see section 5.2.2), the Port Synchronization Memory (see section 5.2.3), and the TISS's Register File (see section 5.2.4). Table 5.4 gives the encodings of each entity. In the current implementation, the Control Interface provides 3 name spaces over a 2 bit vector, which leaves one bit pattern reserved.

OCPS_MAddrSpace[1:0]	Name Space
00	reserved
01	Port Configuration Memory
10	Port Synchronization Memory
11	Register File

Table 5.4: Encoding of the name spaces of the Control Interface

`OCPS_MCmd` According to the OCP specification this signal indicates the type of transfer that currently runs through the Control Interface. The standard dictates a width of 3 bit, even though the current implementation supports only 3 out of all possible OCP commands. The supported OCP commands are listed in Table 5.5.

OCPS_MCmd[2:0]	Command
000	idle
001	write
010	read

Table 5.5: Currently supported OCP commands at the Control Interface

`OCPS_MData` This signal carries the actual configuration or synchronization data, which is to be written into the memory of the name space given by `OCPS_MAddrSpace`. Unlike the corresponding signal in the Port Interface, which width equals the width of the data bus in the lane of the TTNoC, the realization in the Control Interface has a fixed width, which is 32 bit in the current implementation.

`OCPS_MByteEn` Each byte in the data word transported via `OCPS_MData` as well as `OCPS_SData` can be masked so that the value carried in the corresponding byte is not processed. If bit i in `OCPS_MByteEn` is 0, then the bits `OCPS_MData[8i+7:8i]` (in case of a write operation) respectively `OCPS_SData[8i+7:8i]` (in case of a read operation) are set to 0. Otherwise, if i is 1 the corresponding byte in the memory of the given name space is updated (for write operations), respectively is present in the response to a read operation.

`OCPS_MRespAccept` With this signal the host indicates that it is ready to accept current read data from the Control Interface by setting this signal to 1. As long as this signal remains 0, the TISS postpones the delivery of the current read data. The host has to drive this signal to 1 for exactly one clock cycle of the system operation frequency, after the Control Interface has indicated some code other than the "idle state" at `OCPS_SResp`. Such handshake may be necessary in case of slower hosts that cannot process the response data within a single clock cycle of the system operation frequency.

`OCPS_SCmdAccept` The TISS signals that it accepts the current command given by `OCPS_MCmd` by setting this signal to 1 for the very first clock cycle of the system operation frequency, after the new command has been issued at `OCPS_MCmd`. This signal also belongs to the set of handshake signals present at the Control Interface.

`OCPS_SData` Like `OCPS_MData`, but inverse direction. Furthermore, it transports read data from the memory location in the name space, which has been requested during the last command issued at `OCPS_MCmd` and addressed via `OCPS_MAddr`. Like `OCPS_MData` it supports byte masking.

`OCPS_SResp` The Control Interface provides another OCP handshaking signal named `OCPS_SResp`. It indicates the status of the response to a prior issued read operation. According to the OCP standard, the following codes are implemented, as shown in Table 5.6.

`OCPS_SResp[1:0]`	Command
00	no response, idle
01	data valid
10	request failed
11	response error

Table 5.6: Status codes at the `OCPS_SResp` handshaking signal

As long as `OCPS_SResp` shows the code 00, the Control Interface is idle, or no response to a prior issued read operation is available, yet. For instance, this happens after the read operation has been initiated and the TISS currently fetches the data for the response from one of the memories accessible through the Control Interface.

Finally, when the data for the response is visible at `OCPS_SData`, `OCPS_SResp` indicates the validity of data with the code 01.

If the TISS has discovered an invalid address in combination with a read or write operation, the Control Interface shows the invalidity of the most recent operation with the code 10. In this case, no read or write operation has taken place.

However, if the address is valid, but the host tries a write operation at a read-only (write-protected) memory location of some name space, the TISS rejects this write operation, and the Control Interface indicates this rejection by setting a code of 11 at `OCPS_SResp`.

`OCPS_SError` Usually, this signal is used as an interrupt signal that indicates the occurrence of an error. In this case, `OCPS_SError` is driven to 1 for exactly one clock cycle of the system operation frequency. The errors covered here do not consider operational errors of the OCP interfaces, but errors within the core services provided at the UNI. The sources of errors can be queried by the host

via the Control Interface by reading the Error Status Register, as introduced in section 5.2.4.

`OCPS_SFlag` This signal vector with a width of 10 bit also indicates interrupt events and additional status information. In contrast to `OCPS_SError`, these interrupt events are not associated with errors, but with correct operation of the core services. The mapping of the individual bits respectively subset of bits of `OCPS_SFlag` to interrupt events is mentioned in Table 5.7.

Signal	Interrupt / additional information
`OCPS_SFlag[6:0]`	Port Number
`OCPS_SFlag[7]`	Port Operation Complete
`OCPS_SFlag[8]`	Global Reconfiguration Instant
`OCPS_SFlag[9]`	Generic Timer Service Interrupt

Table 5.7: Signals in `OCPS_SFlag` and their associated interrupt events

In the following we explain the interrupt events and additional information of `OCPS_SFlag` in more detail.

Port Number This field always correlates to `Port Operation Complete`. It states that a message has been completed to be sent or received at a given port, and therefore contains the numeric identifier of that port. Note that this is no interrupt event itself, but additional information to an interrupt event. Once the TISS has set the value of the field, it remains stable until the next event.

Port Operation Complete When a message has completely been processed at a port – after the last flit of the last fragment – this interrupt can be triggered to inform the host. Whereas `Port Operation Complete` just indicates the occurrence of the interrupt event, the numeric identifier of the port that has been completed is given by `Port Number`. Other than `Port Number`, which value remains stable until the next "port operation complete" event, `Port Operation Complete` is just driven to 1 for exactly one clock cycle of the system operation frequency. This interrupt can be activated or deactivated for each port individually. This setting is part of the Port Configuration Memory, as described in section 5.2.2.

Global Reconfiguration Instant This is the interrupt signal, which indicates the occurrence of the *reconfiguration instant*. For details about the reconfiguration instant and the reconfiguration procedure in general refer to section 8.4.

Generic Timer Service Interrupt This is the interrupt signal associated with the generic timer service, which is introduced in section 5.4.3.

Note that the TISS does not include an interrupt handling logic. It just signals each interrupt for exactly one clock cycle of the system

operation frequency. The Front-End has to assure that interrupts are recorded and processed properly.

`OCPS_SReset_n` As its name indicates, this signal is low-active. Its purpose is to force the host's application computer to a reset. This signal is associated with the *watchdog service* of the TSS. Whenever the host fails to update the *watchdog life-sign register* in the Register File, the TISS pulls this signal to 0 (low-active!) for one clock cycle of the system operation frequency. For details about the watchdog service refer to section 5.4.2.

5.2.2 Port Configuration Memory

The *Port Configuration Memory* is a dual-ported memory inside the TISS, which is accessed through the Control Interface at name space 01. The purpose of the Port Configuration Memory is to store the set-up of each port individually. As the set-up is in the sphere of control of the host, it is read- and writable for the host. In contrast, the TISS itself just performs read operations to the Port Configuration Memory.

The Port Configuration Memory is organized in consecutive data words, whereas each data word refers to exactly one port. The width of such a data word equals the width of the Control Interface, which is 32 bit in the current implementation. The address of the data word in the Port Configuration Memory corresponds to the identifier of that port. Thus, there is no explicit mapping between port identifier and address in the Port Configuration Memory.

The current implementation supports 128 ports, which are addressable by the 7 useful bits[2] of the address bus of the Control Interface (`OCPS_MAddr`). However, 3 ports are reserved for special operations, as introduced in section 7.4.1.

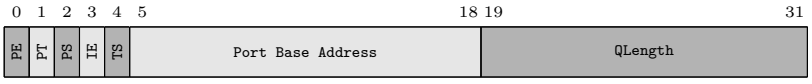


Figure 5.4: Layout of a data word in the Port Configuration Memory

Figure 5.4 shows the layout of a data word in the Port Configuration Memory. The fields within a data word are described in the following.

`Port Enable (PE)` As it name suggests, this binary field activates or deactivates a port. That is, when the port is deactivated, no communication takes place when a fragment of this port is dispatched. The TISS just skips any communication activity of a deactivated port, event though its fragments are still dispatched. As a result, no transfer at the Port Interface is triggered, as well as the Port Synchronization Memory of that port is not affected.

A value of 0 indicates a disabled port, whereas 1 shows that the port is enabled.

[2] excluding the 2 least significant of the total 9 bits of the vector due to word alignment

`Port Type (PT)` The host itself is allowed to determine the semantics of a port. While the TSS just provides the infrastructure (i.e., encapsulated communication channel) for a given port, the host decides whether it uses the bandwidth for the port with an event or state semantics.

If this bit field is 0, the port is used as a state port. Otherwise, a 1 means that this port is an event port.

`Port Sync (PS)` As first mentioned in section 5.1.2, a port can be synchronized either in an explicit or implicit way. While synchronization is actually discussed in section 5.3, here we find the field to configure, whether a given port should use explicit or implicit synchronization.

If this field is 0, explicit synchronization is chosen, whereas 1 corresponds to implicit synchronization.

`Interrupt Enable (IE)` As introduced in section 5.2.1, a completed operation of a port is allowed to trigger an interrupt, which is indicated in the signal `OCPS_SFlag`. This field actually controls, whether a port triggers an interrupt or not.

This "port operation complete" interrupt is deactivated in case of a value of 0, and activated on 1.

`Time Stamp Enable (TS)` The time stamping service is activated for a given port with this field set to 1. In contrast, 0 turns time stamping off.

`Port Base Address` The host itself has to care about the layout of the Port Memory in the Front-End. With this field, the host specifies the base address of a port measured in data words in the Port Memory. Note that in this field, word alignment has not to be considered. In other words, the host need not care to append surplus bits for word alignment. The least significant bit of this field already corresponds to a word address in the Port Memory. Thus, this field comprises just 14 bits, and not 16 like the address bus of the Port Interface (`OCPM_MAddr`).

`Queue Length (QLength)` This field is only meaningful for event ports. Namely, this denotes the maximum length of the queue of an event port. In fact, this maximum length is given in *number of data words in the Port Memory* that the event port consumes. This number is derived from the intended maximum number of event messages and the size of each event message. The formula is obvious:

$$\texttt{QLength} = \text{number\_of}(msg) \cdot \text{size\_of}(msg)$$

Keep in mind that the size of messages also has to consider, whether the time stamping service is currently used for the event port. If so, the message size increases by 2, as a time stamp requires 2 data words in the Port Memory in the current implementation.

In the current implementation, a data word in the Port Configuration Memory reserves 13 bits for this field.

5.2.3 Port Synchronization Memory

The *Port Synchronization Memory* is some kind of "shared memory" to inform the participants of the port synchronization protocol (the TISS and the host) of the status of access of its opponent. In general, it is a dual-ported memory that is read and written by the TISS as well as the host. In fact, it is the most frequent accessed entity via the Control Interface. It is assigned the name space 10 according to Table 5.4.

While the real protocol of port synchronization is introduced in section 5.3, this section illustrates the layout of the Port Synchronization Memory. Like the Port Configuration Memory, for each port there exists a distinct data word in the Port Synchronization Memory. Consequently, it is made up of 128 data words of width 32 bit each.

The first and most important property of the Port Synchronization Memory is the fact that a given data word is *reused* for all combinations of port directions (i.e., input or output port) and port semantics (state or event port) of the same port. This approach considerably saves memory size for port synchronization. As a result, a given data word can be interpreted in 3 different ways, as depicted in Figure 5.5.

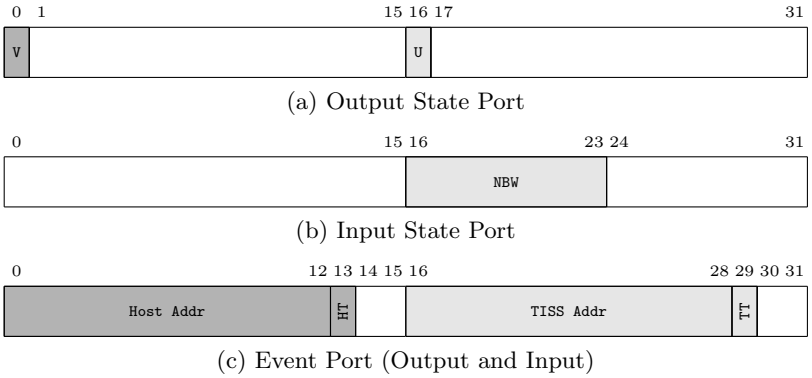


Figure 5.5: Layout of a data word in the Port Synchronization Memory with its interpretations

To sum up, depending on the settings in the corresponding data word in the Port Configuration Memory, the bits in the data word of the Port Synchronization Memory have different meaning. There is a different interpretation for output state ports, input state ports, and event ports (both input event ports and output event ports). In the following we describe the meaning of bits and subsets of bits of a data word of the Port Synchronization Memory.

`TISS Addr` This field is only relevant for event ports. For an input event port it specifies the write position in the event queue, otherwise (i.e., for an output event port) it marks the read position from the point of view of the TISS. Read and write positions are interpreted as offsets from the port base address, which is configured in the Port Configuration Memory. `TISS Addr` is incremented

according to the formulae in Table 5.8, after an event message has completely been processed.

Time Stamping	calculation
disabled	$\texttt{TISS Addr}_{new} = (\texttt{TISS Addr}_{old} + N)\ mod\ \texttt{QLength}$
enabled	$\texttt{TISS Addr}_{new} = (\texttt{TISS Addr}_{old} + N + 2)\ mod\ \texttt{QLength}$

Table 5.8: Calculating new values of `TISS Addr`

N denotes the net message size in the Port Memory's data words, and `QLength` is the same as the field in the Port Configuration Memory. Also note that the time stamping service has to be considered, which affects the size of an event message. The additional 2 for enabled time stamping embodies the data words required for a time stamp according to the current implementation.

As read and write positions in an event queue can reach the value of the queue length, this field possesses the same width as the field holding the queue length (`QLength`) in the Port Configuration Memory, which is 13 bit in the current implementation.

`TISS ToOF (TT)` (Toggle on Overflow of `TISS Addr`) Like `TISS Addr` this field is only present for event ports. This single bit field is toggled, whenever `TISS Addr` has overflown, i.e., has become 0 after a calculation. As we learn from section 5.3, this information is vital to distinguish between full and empty event queues.

`Host Addr` This field has the same semantics as `TISS Addr`, but it considers an event queue from the host's point of view. For input event ports it is the read position, whereas for output event ports it is the write position. Also, new values of read/write positions are calculated like `TISS Addr` according to Table 5.8.

`Host ToOF (HT)` (Toggle on Overflow of `Host Addr`) This is the equivalent of `TISS ToOF`. Obviously, it deals with the overflow of `Host Addr`.

`NBW` Input state ports apply the NBW protocol in order to let the host synchronize to this field, which operates as the NBW sequencer. Note that this field is exclusively written by the TISS, but also read by the host. In the current implementation, a data word of the Port Synchronization Memory assigns 8 bits for this field. For details concerning the synchronization protocol of each port type refer to section 5.3.1.

`Using (U)` This single bit field is used for the synchronization of output state ports. It is one part of the synchronization protocol for output state ports, namely the TISS-side part. For details concerning the synchronization protocol of each port type refer to section 5.3.1.

`Valid (V)` This single bit field is used for the synchronization of output state ports. It is one part of the synchronization protocol for output state ports, namely the host-side part. For details concerning the synchronization protocol of each port type refer to section 5.3.1.

Note that all fields that are exclusively written by the TISS are located in the upper two bytes of the data word. In contrast, the two lower bytes are intended to be exclusively written by the host. This way, the efficiency of write accesses to the Port Synchronization Memory is increased. It also allows for an efficient write protection mechanism by means of hard-coded byte masking. For instance, the fields belonging to the TISS are protected from the host, if the upper two byte mask signals are hard-wired to 0 at the host-side port of the dual-ported Port Synchronization Memory.

The whole data word of the Port Synchronization Memory is automatically reset (i.e., set to 0), when the host updates the set-up of the corresponding port in the Port Configuration Memory. Nevertheless, the host never has (write) direct access to these parts of the data word, which are exclusively written by the TISS.

5.2.4 Register File

The Register File offers additional information and control facilities to the host via the Control Interface. For example, this information includes a copy of the current value of the global time base, and the Error Status Register that contains the status of error conditions. Additionally, it realizes the control mechanism of the special services provided by the TISS. This includes the watchdog service (see section 5.4.2) as well as the generic timer service (see section 5.4.3).

Figure 5.6 gives the layout of the Register File. Note that this is the Register File of the TISS. Besides this, the Front-End is also allowed to supply register files to provide control mechanisms and information to the host. As a consequence, the TISS's Register File[3] is distinguished from other entities such as Port Configuration Memory or Port Synchronization Memory by the name space 11 at the Control Interface.

We learn from Figure 5.6 that the Register File is made up of 8 data words. The width of these data words is aligned to the width of the data buses of the Control Interface, which is 32 bit in the current implementation. If a data word of the Register File does not fill up the whole width, then the data word is stuffed with zeros.

In the following we describe each entity in the Register File.

`Global Time` This field embodies "virtual" registers. That is, actually there exists no real two data words in the Register File, which hold the upper and lower

[3] In the following we simply call the TISS-side Register File "register file".

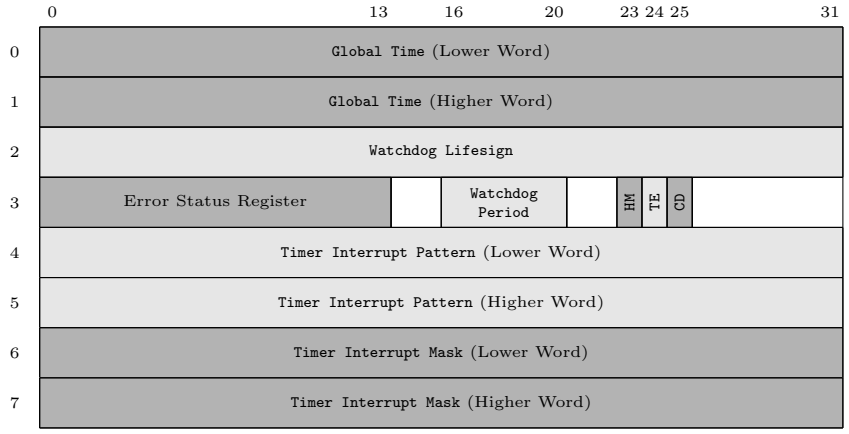


Figure 5.6: Layout of the TISS's Register File

half of the global time base. Whenever the host accesses these addresses via the Control Interface, the TISS extracts a copy of the current value from the counter vector of the time format of the global time base. As the time format used in the TTSoC architecture comprises 64 bit, this copy has to be split into an upper and lower half to be transported through the Control Interface. By design, when the host intends to read the current time (i.e., the current value of the counter vector), **first it must read the lower half (at offset 0) and afterwards the upper half (at offset 1)**. A partial access to any of the halves is not allowed. The reason for this is, that the TISS buffers the whole copy into a shadow register. So, when the upper half is read after the lower one, the TISS supplies the appropriate value from the shadow register. As a result, reading the global time base is always consistent, even though there might elapse some time between the accesses of lower and upper half.

Note that both "virtual" data words are read-only. The host could try to write to them, however this would not have any effect to the global time base. However, the Control Interface would indicate the violation of write protection via `OCPS_SResp` with code 11. As a consequence, the host is not able to set the global time base itself. Setting the global time base is in the sphere of control of the TNA, which is explained in section 8.4.

`Watchdog Life Sign` This is the register, where the host must write periodically in order to avoid triggering the watchdog miss error. This register is named *watchdog life-sign register*, because the host has to write the *life-sign* there, which is a fixed 32 bit string of $0x55555555$. For details about the watchdog service refer to section 5.4.2.

`Communication Disable (CD)` The host is enabled to turn off all communication activities of its TISS via this field. Other than the field `Port Enable` in the Port Configuration Memory, which just affects a single port, `Communication`

`Disable` simultaneously disables the communication at all ports. Be aware that the semantics of this field is tricky. Namely, a value of 0 keeps the communication enabled, whereas 1 disables the communication. Also note that in the current implementation, at start-up respectively reset of the TISS this field is set to 1. **Thus, communication is disabled by default, and has to be explicitly enabled by the host**.

Even though no sending or receiving operation at the Port Interface as well as TTNoC interface takes place, disabling communication does not turn off the dispatching within the TISS, i.e., the Burst Dispatcher. The instants, when a burst associated with a fragment of a message in a given port is to be processed, are still determined by the Burst Dispatcher, although the operation is skipped in the Port Manager afterwards, in case of disabled communication. The reason for this is that dispatching communication activities is in the sphere of control of the TSS, which operates autonomously from the hosts. Disabling dispatching when disabling communication would interfere with the autonomy of the TSS, and would therefore violate this fundamental design assumption.

Furthermore, there is one communication activity that can never be deactivated – the port associated with the reconfiguration of the TISS by the TNA. For details about the on-the-fly reconfiguration have a look at section 8.4.

`Timer Interrupt Enable (TE)` This is the field that enables and disables the generic timer service, introduced in section 5.4.3. The semantics is straightforward: 0 disables the service, 1 enables it.

`Host Mode (HM)` This is additional information included in the TISS's Register File, which has been set by the TNA during the reconfiguration. This information serves to let the host know, whether its application services are wanted or not. A value of 0 indicates that the application services realized in the attached host are not required at the moment, and therefore the host should rather suspend in order to save energy. Apparently, the opposite situation applies for a value of 1. However, the host is not forced to fulfil the recommendation expressed in the Host Mode. It is in its own sphere of control, whether to suspend or not.

`Watchdog Period` The watchdog period is another piece of information that has been set by the TNA during the most recent reconfiguration. Like `Host Mode`, this field is just read-only to the host. The purpose of this field is to inform the host about the period that is associated with the watchdog service. Anyway, the watchdog period has been arranged by being included in the Time-Triggered Communication Schedule. So, with this information the host is enabled to react properly to the current watchdog period, i.e., to know when to update the life-sign register. For details about the watchdog service refer to section 5.4.2.

`Timer Interrupt Pattern` **and** `Timer Interrupt Mask` These two entities control the behaviour of the generic timer service. As both are are related to the time format of the global time base, which is a 64 bit, each has to be split into two

halves, resulting into a total of four data words in the Register File. The usage and meaning of these entities is covered in section 5.4.3, which comprehensively introduces the generic timer service.

Error Status Register Any error, which is detected by the TISS and concerns some of the core services of the TSS, is recorded and reported via the Error Status Register. Figure 5.7 illustrates the layout of the Error Status Register in detail. The following explains the meaning of each field in the Error Status Register.

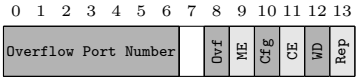


Figure 5.7: Layout of the Error Status Register

`Overflow Port Number` An overflow of the queue of an event port might occur in two different situations. Firstly, for output event ports, when the host produces event messages faster than they are sent by the communication service, the event queue will finally run full, and no further event message can be appended to the queue. Secondly, for input event ports, if the host consumes event messages slower than they are delivered by the communication service, the event queue will also overflow. Note that this situation is not caused by the communication service of the TTSoC architecture, but by a host, which does not put the proper effort into handling that event port.

If such a situation arises, this field holds the identifier of the port, which has caused the overflow. After it has been set, the value remains stable.

`Queue Overflow Error (Ovf)` This field indicates the actual occurrence of an overflow at some event port. It is set, when the TISS first discovers an overflow of some event port. It is a single bit, therefore it can not closer specify, which port has triggered the `Queue Overflow Error`. This additional information is kept in the field `Overflow Port Number`. In contrast, `Queue Overflow Error` can just signal that an erroneous condition has been met.

`Port Memory Error (ME)` As introduced in section 5.1.1, the Port Interface propagates a feedback signal, which concerns errors at the Port Memory. This error signal is wired across the TISS and is recorded in the Error Status Register in the field under consideration (`Port Memory Error`).

`Port Configuration Error (Cfg)` This error arises, whenever a burst is dispatched by the Burst Dispatcher, which belongs to a port that has been disabled by the host by means of `Port Enable` in the Port Configuration Memory.

`Communication Error (CE)` This error is related to `Port Configuration Error`, though, it is more generic. It does not consider a single port,

but all ports in general. The `Communication Error` is triggered, when the Burst Dispatcher dispatches a burst of any port, however, the communication service has been deactivated by the host by means of the field `Communication Disable` in the Register File.

`Watchdog Miss (WD)` This field records the incident, when the host fails to update the watchdog life-sign register in time. This is called a *watchdog miss*. For details about the watchdog service refer to section 5.4.2.

`Repeated Error (Rep)` The intention of this field is to point out a repeated occurrence of any of the error conditions introduced above. This field is set to 1, if another field in the Error Status Register has been set to 1 before, and the same error arises again. However, this feature does not specify, which of the other errors has occurred more than once. Usually, only one source of error is expected for a given interval of time.

The host has no means to delete any field in the Error Status Register. The whole Error Status Register is automatically reset, when the dissemination service sends a copy of the current value of the Error Status Register to the diagnosis unit. As a result, the Error Status Register records all error conditions during the interval between two consecutive occurrence of dissemination. For details about the dissemination service, which is a special service of the TISS, refer to section 5.4.4.

5.3 Synchronizing Access to Ports

In earlier sections we have learned that ports are the end points of encapsulated communication channels, which physically reside in the Port Memory, which in turn is part of the Front-End. The Port Memory is realized as an dual-ported memory, thus, it must be regarded as a shared memory. As the TISS and the host are allowed to arbitrarily access and modify data words that belong to ports in the Port Memory, we identify the need for synchronization mechanisms.

The TTSoC architecture entails synchronization protocols for accessing ports, which are examined in this section. We present the synchronization protocol for input and output state ports as well as event ports in any direction.

5.3.1 Synchronizing State Ports

State ports are used for periodic transmission of messages with state semantics, so-called *state messages*. These are instances of state variables, which contain the state of an observation of an object under consideration [Kop97].

State messages apply state semantics, whereas a new arriving version completely overwrites the old version of the state variable. Consequently, an *exactly-once* semantics is not required, which is in sharp contrast to an *at-least-once* semantics usually applied to event messages.

Due to this strategy, a state ports only holds one single state message at a time. In order to ensure that only consistent data is sent and received through the TSS, the TTSoC architecture defines the following synchronization protocols for input and output state ports.

Synchronization Protocol for Input State Ports

For the purpose of synchronization, an input state port employs the *Non-Blocking Write (NBW)* [Kop97] protocol. The purpose of NBW is to detect a situation, when the TISS updates an input state port owing to a reception of parts of a state message, while the host conducts a read access from that port. The NBW protocol is realized by means of a sequencer that is exclusively written by the TISS and read by the host. As we have learned from section 5.2.3, this sequencer for each input state port resides in the corresponding data word of the Port Synchronization Memory.

At the beginning of the protocol the sequencer is 0. Whenever the TISS starts an update of the port, it increments the sequencer by 1. After completion of the update (i.e., the completion of a whole message), the TISS again increments the sequencer. By design of the protocol, the host has to combine a read from an input state port with a look-up from the NBW sequencer in the Port Synchronization Memory.

If the value is odd, the host retries the access immediately, because an update is currently in progress. If the value is even, an update has been completed and the data of the port is consistent. Consequently, the host is allowed to perform the read operation.

At the end of the host's read operation the host has to check the value of the sequencer once more, whether the even value has turned into odd in the meantime. In such a case, this means that the TISS has begun to update the input state port, while the host has been reading. Then, the host has to abort its previously fetched data and retry the read access in order to not retrieve inconsistent or corrupted data.

This synchronization protocol for input state ports is known as *explicit synchronization*, as it takes usage of auxiliary facilities for synchronization. In contrast to explicit synchronization, input state ports also support *implicit synchronization*. In this case, the synchronization protocol disregards auxiliary facilities such as the NBW sequencer.

However, implicit synchronization of input state ports can only be applied with system operation frequencies at the TISS as well as the host, which are closely synchronized. Based on the *a priori* known points in time (in the Time-Triggered Communication Schedule), when the input state port is updated by the TISS, the host can temporally interleave its read accesses with the updates by the TISS so that no conflict arises.

Synchronization Protocol for Output State Ports

In order to ensure the consistency of data, output state ports reserve a double buffer (a shadow buffer) in the Port Memory, which is explained in section 5.1.2 and illustrated in Figure 5.1. As a shadow buffer can be regarded as auxiliary facility, coping with shadow buffers embodies *explicit synchronization* of output state ports.

The shadow buffer is of the same size as the real state message. As a consequence, it is possible to place two complete state messages into the memory area reserved for the output state port. While one half of the reserved memory area holds a consistent state messages, which is not updated again and is allowed to be sent by the TISS, the other half can be written by the host to place a state message (with a new version of the state variable) into the Port Memory. Generally, each half of the total reserved memory of the output state port toggles its role in the synchronization protocol. Once it holds the consistent state messages, which is to be sent by the TISS, then it is regarded as the active buffer. After the send operation has been completed, the same half becomes the shadow buffer to pick up another state message from the host.

The synchronization between TISS and host incorporates the corresponding data word in the Port Synchronization Memory. In the interpretation of output state ports, the data word includes a field `Valid` and `Using`, as introduced in section 5.2.3. The `Valid` field is written by the host. It indicates, which of the two buffers (the two halves of the total reserved memory area in the Port Memory) is currently containing valid data. In other words, it tells which one of the two buffers currently is the active buffer. In contrast, the `Using` field is written by the TISS. It states, which of the two buffers the TISS is going to use for the next send operation of that port. **The host is only allowed to update the content of the shadow buffer, if both `Valid` and `Using` point to the active buffer**.

At the beginning of the protocol both `Valid` and `Using` are initialized to 0. This claims that the first half of the total memory area of this port (let's call it buffer 1) is currently the active one. At this time, the host would write to the second half (let's call it buffer 2) that is the shadow buffer at the moment. After an update by the host to buffer 2 it sets the `Valid` field in the corresponding data word of the Port Synchronization Memory to 1. As long as the TISS has not completed the send operation with buffer 1, the host can not execute another write access to that port, because the synchronization protocol requires both fields to have the same value in order to update the complementary buffer. When the TISS has finished the send operation of the data of buffer 1, it sets the `Using` field to 1. Now, buffer 1 has become the shadow buffer and buffer 2 is the active buffer. For the next update the host would use buffer 1, and the TISS would fetch the send data from buffer 2 in the meantime. This alternating procedure takes place infinitely often.

Output state ports support *implicit synchronization* like input state ports. In this case no double buffer is required. With the *a priori* knowledge of the instants of send operations of the TISS, the host is enabled to temporally interleave the write access to the port, which consists of a single buffer here. Consequently, inconsistent send data is avoided by means of synchronized system clocks of TISS and host.

5.3.2 Synchronizing Event Ports

Event messages communicate the difference between an old state and a new state of an entity under observation [Kop97]. This difference is also called the event information. Event messages have to be processed in an *exactly-once* semantics. Therefore, messages have to be consumed on reading, and unread messages have to be queued instead of an update-in-place strategy like for state messages.

The TTSoC architecture offers event ports for the sporadic transmission of messages with event semantics. The queue is realized as a ring buffer that can hold multiple event messages simultaneously, as illustrated in Figure 5.3 and described in section 5.1.2. The queue length is variable as well as configurable by the host at run-time via the Control Interface by assigning the proper value to the field `QLength` in the Port Configuration Memory.

For output event ports, the host of a sending micro component is blocked, if it tries to append a new message while the queue is full. Contrary for input event ports, the TISS of a receiving micro component discards arriving messages, if the host has not consumed the messages quickly enough so that there is no space left for at least one event message, i.e., the receiving queue is full. Such a scenario is called on overflow. As introduced in section 5.2.4, the Error Status Register facilitates the indication of full queues respectively overflows by means of the fields `Overflow Port Number` and `Port Overflow Error`.

The status of an event queue can be derived from the fields `TISS Addr`, `TISS ToOF`, `Host Addr`, and `Host ToOF`, which are part of the data word of the Port Synchronization Memory. The relevant information is, whether an event queue is currently full or empty. Table 5.9 summarizes how this information about the status of an event queue is derived from the fields in the Port Synchronization Memory. In section 5.2.3 it is also given, how this fields are processed, which is also relevant in the following.

queue status	condition
full	$\texttt{TISS ToOF} \neq \texttt{Host ToOF} \wedge \texttt{TISS Addr} = \texttt{Host Addr}$
empty	$\texttt{TISS ToOF} = \texttt{Host ToOF} \wedge \texttt{TISS Addr} = \texttt{Host Addr}$

Table 5.9: Determining status of event queues

An *empty queue* is given, when read and write positions (i.e., `TISS Addr` and `Host Addr`) – no matter which field takes the role of the read and write position, as this fact symmetrically applies to input and output event ports – are equal. This means that all event messages in the queue have been consumed, as the read position has been incremented (compare Table 5.8) so that the read position has caught up to the write position. Additionally, the `ToOF` fields have to possess the same value, because this takes the cyclic overflow of the ring buffer into account.

Similarly to an empty queue, the *full queue* manifests in equal read and write positions, but deviating `ToOF` fields. The inequality of the `ToOF` fields results from

the cycling ring buffer, too. If read and write positions are the same, but the `ToOF` fields are not, then write position has slipped forward and completed a cycle in the ring buffer compared to the read position. As a consequence, the appropriate `ToOF` field must have toggled so that both `ToOF` fields are deviating now, because circling through the ring buffer obviously causes an cyclic overflow of the ring buffer and therefore a toggling of the corresponding `ToOF` field.

5.4 Special Services

This section introduces special services that are locally implemented by the TISS. They provide additional utility to the host, and therefore supplement the core services of the TSS.

5.4.1 Time Stamping

The purpose of time stamping is to append the value of the counter vector of the time format of the global time base — a *time stamp* — in front of the data words of the real message of a port. This time stamp is written into the Port Memory by the TISS, when a message of that port has been completed – after the last flit of the last fragment. Thus, time stamping is a useful means to inform the host application about the real instant of complete arrival of a message. This feature might particularly be useful for event ports, where the arrival of messages is sporadic. As the global time base in the TTSoC architecture is 64 bit wide, in the current implementation (with a data word width of 32 bit in the Port Memory) a time stamp requires two data words in the Port Memory.

Note that time stamping is only meaningful for input ports, no matter whether the port has state semantics or event semantics. If the host application activates time stamping for an output port, though, this setting is ignored and has no effect. Besides this, it can be activated and deactivated for each port individually via the Control Interface, as described in section 5.2.3.

Time stamping is in the sphere of control of the DAS that is executed in the micro component's host. Furthermore, activation and deactivation of the service might occur arbitrarily. When receiving messages for an input port, the TISS always has to check, whether time stamping is on or off at the moment, and then either appends the time stamp in front of the message or places the message from the beginning of its intended memory location.

As a consequence, the memory requirements in the Port Memory for a given port might also increase or decrease according to the current set-up of a port. While this has no effect on the TISS's operation, the user application in the host has to assure that the memory location of ports do not overlap due to activation and deactivation of time stamping at any time. Otherwise, the TISS would overwrite parts of messages of other ports.

5.4.2 The Watchdog Service

The health of a micro component's host can be monitored by means of a *watchdog service* that is integrated in the TISS. Therefore, the watchdog forces the host to update a dedicated memory location within a defined period at latest. If the host fails to write the *life-sign* to that memory location, the TISS resets the host automatically, and a failure is recorded to the diagnostic dissemination service.

The control of the watchdog service is realized through the Control Interface of the UNI. As introduced in section 5.2.4, the Register File includes the registers `Watchdog Period` and `Watchdog Life Sign`. The watchdog life-sign register resides at offset 2 in the Register File, which is illustrated in Figure 5.6. The host has to perform a write access to this location with a defined 32-bit string, which is $0x55555555$, to prevent the watchdog service to trigger a *watchdog miss*.

The maximum interval between two updates of the watchdog life-sign register is expressed by the 5 bit wide field `Watchdog Period`, which is embedded in the Register File at offset 3 (see Figure 5.6). The watchdog period conforms to periods of the periodic control system, as introduced in section 4.3.1. The field `Watchdog Period` holds the number of the period that drives the watchdog service. For example, with 16 supported periods, whereas the highest period number 15 has an interval of 2 seconds, a `Watchdog Period` of 01111 configures the watchdog service to check every 2 seconds, whether the host has already updated the watchdog life-sign register or not. Note that the watchdog service does not use phases of periods.

The watchdog service supports up to 31 periods (from values 00000 to 11110) that comply with the periods of the periodic control system. Consequently, there is one period reserved to indicate that the watchdog service is turned off. In the current implementation, if `Watchdog Period` is 11111 (all bits in this field set 1), the watchdog service is turned off.

The host should regularly (for instance, after the reconfiguration instant) check the setting of the field `Watchdog Period` so that it knows, how frequently it has to write to the watchdog life-sign register.

Setting the field `Watchdog Period` is in the sphere of control of the TNA, and is therefore configured during the process of reconfiguration, which is explained in section 8.4. Note that it is not sufficient to set the field `Watchdog Period` by the TNA in order to configure the watchdog period. Actually, the watchdog service uses the infrastructure of burst dispatching of the TISS. That is, checking a watchdog miss is initiated by a special treatment of a "virtual fragment" that belongs to a special purpose port (i.e., port 125, see section 7.4.1), but does not cause any transfer of application data. Consequently, checking a watchdog miss manifests as a burst of a "virtual fragment" in the Time-Triggered Communication Schedule, and is therefore configured together with bursts that deal with the transfer of real application data. So, the concept of "virtual fragments", which has been introduced to cope with consistent delivery order in encapsulated communication channels, is reused to realize the watchdog service.

As a host has no direct read access to the Time-Triggered Communication Schedule by design, the field `Watchdog Period` is the only means to inform the host about the current configuration of the watchdog service. Therefore, during reconfiguration it has to be configured in combination with that burst of a "virtual fragment", which is integrated in the Time-Triggered Communication Schedule.

If the host fails to update the watchdog life-sign register for any reason, the watchdog service triggers the watchdog miss error. This error is propagated to the Error Status Register in the Register File and recorded in the corresponding field `Watchdog Miss`. Consequently, the host notices the occurrence of the watchdog miss when reading the Error Status Register. Additionally, the dissemination service also informs the diagnostic unit (see section 5.4.4) about the watchdog miss.

Besides this, the incident of a watchdog miss is linked to the actual feature of the watchdog: to reset the host, i.e., the host's CPU, when it has failed to update the watchdog life-sign register. For this purpose, the TISS drives the low-active signal `OCPS_SReset_n` of the Control Interface to 0 for exactly one clock cycle of the system operation frequency. Usually, this signal is connected to a reset signal like "cpu_reset" and causes the host's application computer to reset in order to be relieved from a possible deadlock, software hang-up, or some other malfunctioning state.

5.4.3 The Generic Timer Service

The *generic timer service* is a convenient additional service that allows the host's application software to take arbitrary usage of the global time base used in the TTSoC architecture. The generic timer service harnesses the concept of periods and phases of the periodic control system to provide interrupt events aligned to real-time, which in fact are not linked to any other events of core services such as "port operation complete" interrupts. For instance, the application software could have a piece of software triggered by the generic timer service, which execution is independent from any communication activity.

The generic timer service is configured by two registers in the Register File, which has been introduced in section 5.2.4:

- `Timer Interrupt Pattern`
- `Timer Interrupt Mask`

As both registers hold values of the counter vector of the time format of the global time base, which is 64 bit, and the data width of the Control Interface as well as the Register File is bound to 32 bit in the current implementation, they have to be split into two 32 bit data words in the Register File each. Consequently, as illustrated in Figure 5.6, the lower halves of `Timer Interrupt Patter` and `Timer Interrupt Mask` are available at offsets 4 respectively 6, while the upper halves are located at offsets 5 and 7.

`Timer Interrupt Pattern` gives the bit pattern with respect to the time format of the TTSoC architecture. If this bit pattern equals the bit pattern that is present in the internal register of the global time base, a *match* occurs. This match triggers the interrupt of the generic timer service. To relax the complexity of the bit pattern to be set, the register `Timer Interrupt Mask` also contains a bit pattern that marks the bits in the time format that are relevant for the match. So, the generic timer first performs a bit-wise AND between `Timer Interrupt Mask` and the current value of the internal register of the global time base. After that, it compares this intermediate result with the bit pattern residing in `Timer Interrupt Pattern`.

$$match \leftarrow interrupt\ pattern = (global\ time \otimes interrupt\ mask)$$

This mechanism of pattern register and mask register is powerful enough that the generic timer service supports the full range of periods and phases of the periodic control system. Timer interrupts of the generic timer service are enabled to occur periodical (with respect to a given period) and phase aligned by appropriately specifying the registers `Timer Interrupt Pattern` and `Timer Interrupt Mask`.

These registers should be set by the host, while the generic timer service is turned off. Section 5.2.4 introduced the bit field `Timer Interrupt Enable` in the Register File, which is in charge of enabling and disabling the generic timer service. So, after the host has set the configuration registers, it sets `Timer Interrupt Enable` to 1, and the generic timer service is armed.

Eventually, when a match occurs and the timer interrupt is triggered, this event is visible at the Control Interface as a special field in the signal `OCPS_SFlag`, as listed in Table 5.7 in section 5.2.1. Then, the corresponding signal is driven to 1 for exactly one clock cycle of the system operation frequency so the host is notified via the interrupt mechanism.

5.4.4 The Dissemination Service

The TISS includes a mechanism to record error conditions associated with core services – the Error Status Register of the Register File introduced in section 5.2.4. On the one hand, such error conditions are useful to be read by the host. On the other hand, the information contained in the Error Status Register is employed to execute diagnostic analysis algorithms that draw conclusions about the health state of the TTSoC. The *dissemination service* is the auxiliary service of the TTSoC architecture that is in charge of distributing the contents of the Error Status Register to a dedicated diagnosis unit in the TTSoC.

Like the watchdog service, the dissemination service uses the dispatching facilities of the TISS and the concept of "virtual fragments" to perform its periodic dissemination of diagnosis information such as the Error Status Register. As mentioned in section 7.4.1, the dissemination service is associated with the reserved port number 126. The Time-Triggered Communication Schedule contains a burst of a "virtual fragment" referring to this special port. Whenever this burst of the "virtual

fragment" is dispatched, the TISS executes a special treatment that handles the dissemination service. That is, the TISS fetches the Error Status Register and injects to the TTNoC via the TTNoC interface in a single flit (as the Error Status Register can be contained one flit of 32 bit in the current implementation). Before that it has preceded the proper routing information so that the dissemination data finds its way to the diagnosis unit. Note that there is no real transfer at the Port Interface involved.

After the dissemination the Error Status Register is reset. Until the next occurrence of the burst of the "virtual fragment" that triggers the dissemination service, the TISS records all error conditions in the Error Status Register again. As a result, the dissemination service distributes diagnosis data that has been collected in the interval between two disseminations.

In fact, the host is unaware of the existence of the dissemination service. This auxiliary service is under the sphere of control of the TNA. As it harnesses the dispatching infrastructure of the TISS by integration in the Time-Triggered Communication Schedule, the dissemination service is also configured during the on-the-fly reconfiguration, which also handles the Time-Triggered Communication Schedule, as explained in section 8.4.

Chapter 6

The Time-Triggered Network-on-Chip

The Time-Triggered Network-on-Chip (TTNoC) is the architectural component, which physically connects the micro components of a TTSoC. It conveys data that is transported through encapsulated communication channels. These encapsulated communication channels are managed by the TISS of a micro component. Each TISS accesses the TTNoC via its local TTNoC interface. The TTNoC does not incorporate any control or management functions, but it is the medium to carry encapsulated communication channels.

6.1 Basics of the TTNoC

The TTNoC is composed of *Fragment Switches*. Fragment Switches are stand-alone components that transport fragments, i.e., the flits that make up a fragment of a pulse in a pulsed data stream, of data along an encapsulated communication channel. Each micro component, i.e., the TISS of each micro component, is connected to exactly one Fragment Switch via the *TTNoC interface*.

6.1.1 Topological Considerations

Fragment Switches are physically connected by means of separate, uni-directional buses, so-called *lanes*. An *interconnect* consist of a pair of one outgoing and one incoming lane. Each interconnect links one pair of Fragment Switches. Thus, the link between two Fragment Switches is bi-directional. This gives an additional degree of freedom to the routing in the TTNoC. Furthermore, we learn in section 6.1.3 that this also entails performance improvements for transmissions.

A TTNoC interface has the same physical structure as an interconnect. Each TTNoC interface is the terminal point of transmission in an encapsulated communication channel. In other words, any Fragment Switch that is just a switching

node never takes the role of the termination node, but the role of a forwarder. This conforms to the definition of an "indirect network" [DT03].

The arrangement of Fragment Switches and their interconnections construct the topology of the TTNoC. Conceptually, the design of the TTNoC does not dictate a specific topology in order to not restrict the design space. Instead of this, the TTNoC provides a set of templates for Fragment Switches with different parameters, e.g., the number of interconnects and the width of lanes, in a design repository. Hence, a Fragment Switch is used as a generic building block in the topology.

6.1.2 Lanes

In Figure 6.1 we find the structure of a lane. At first, a lane is composed of a data bus, which width can be configured at design time, for instance 32 bit. Additionally, two wires (`valid` and `header`) make up the only control signals of a lane.

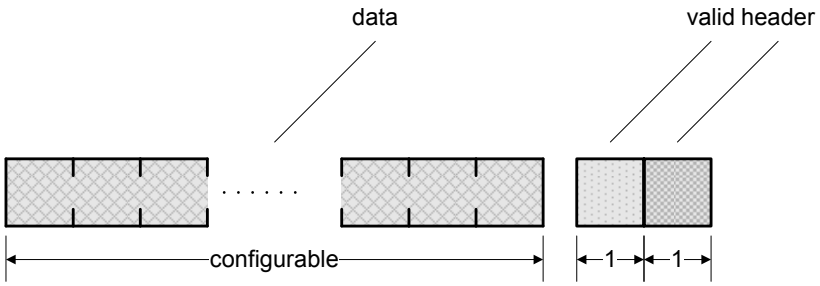


Figure 6.1: Structure of a lane

The `valid` signal indicates the arrival of a flit. Unless `valid` has been driven to 1, the Fragment Switch remains in a power-saving idle state. The `valid` signal is also propagated through the Fragment Switch to the neighbouring switch or a TTNoC interface. Initially, it is driven by the sending TISS on the TTNoC interface and accompanies a given flit along the whole route.

The `header` signal set to 1 specifies that a flit contains switching information. Otherwise, when it is 0, the current flit is interpreted as data. This is the one and only semantics of Fragment Switches. In section 6.2.1 we learn about the purpose of switching information.

6.1.3 Operation of Fragment Switches

Fragment Switches do not possess any knowledge concerning the Time-Triggered Communication Schedule, but they simply handle flits. They do not even include a notion of global time themselves. Instead of this, all operations are initiated by TISSs, which follow a Time-Triggered Communication Schedule and include the global time base.

Like the TISS, the basic mode of operation of a Fragment Switch is a *burst*. That is, a Fragment Switch is able to accept (respectively forward) one flit for each lane at each interconnect in each clock cycle of the system operation frequency. Thus, the TTNoC supports full-duplex transmissions among all interconnects of each Fragment Switch.

Fragment Switches are also unaware of the fragmentation of pulses of a pulsed data stream. The fragment is split in the sender's TISS and assembled in the receiver's TISS. In short, the Fragment Switches do not contain any knowledge of the semantics, neither in value nor in temporal domain of the data conveyed.

6.1.4 Hops

The processing of a given flit entering a Fragment Switch at one interconnect and leaving at another one within the same Fragment Switch is called a *hop*. Conceptually, **no buffering** of data flits takes place, which could extend the propagation delay and lead into a discontinuation in the TTNoC. On the one hand, the lack of buffering contributes to the predictability and determinism of the TTNoC. On the other hand, this enables an efficient realization of Fragment Switches. However, the hop is pipelined. A Fragment Switch includes a 4-stage pipeline. These stages are:

1. **flit entering** A flit arrives at an interconnect and is placed in its input register. The Fragment Switch awakes from the idle state.

2. **decoding and forwarding** In case of a flit containing switching information (and it is the first of such flits in the current fragment), the multiplexer settings are decoded according to the information in that flit, and forwarded afterwards. In case of a data flit, just forwarding takes place and the multiplex decoder remains idle.

3. **multiplexing** The flit passes through the multiplexer of the output lane of the appropriate interconnect.

4. **flit leaving** The flit is kept in the output register of the output interconnect, until it is replaced by the consecutive flit.

In the current prototype implementation, the pipeline stages are operated within one clock cycle of the system operation frequency. Thus, there is no need for output registers. The output signals of a multiplexer are directly connected to the input register of the incoming lane of the neighbouring Fragment Switch.

Obviously, Fragment Switches are *multiplexer-based*. That is, the internal structure focuses an outgoing lane and tracks the incoming lanes that feed that outgoing lane. Note, that the settings of these multiplexer are derived from switching information that has been transported over the same physical wires as the data.

6.2 Switching in the TTNoC

This section comprehensively deals with the switching in the TTNoC. We present the basic concepts of switching, and investigate the frequently used scenario of multicasting, as it is realized in the TTNoC.

Actually, Fragment Switches are unaware of the switching of encapsulated communication channels through the TTNoC. Each Fragment Switch has to be fed locally with the information, how flits entering a particular interconnect have to be passed on to which interconnects – the so-called *switching information*. Each chunk of switching information concerns a particular hop, the sequence of switching information directly maps to a sequence of hops. The complete sequence of switching information from the sending TISS's TTNoC interface to the receiving TISS's TTNoC interface is called *routing information*, and defines the *route* of an encapsulated communication channel. Considering hops, it is obvious that routes are segmented by Fragment Switches.

6.2.1 Switching opcodes

A particular flit, the *routing flit*, contains the switching information for a given Fragment Switch. The routing flit is conveyed over the data bus of a lane and is identified by the `header` signal set to 1. Even though the routing flit is passed on to the neighbour, the Fragment Switch processes parts of that flit. This part of the routing flit, which is aligned at the least significant bit of the data bus, contains the switching information for the current Fragment Switch – the so-called *switching opcode*. The switching opcode is consumed at the current Fragment Switch. The remaining routing flit is forwarded to the neighbour with the switching opcode for the next Fragment Switch again aligned at the least significant bit, as depicted in Figure 6.2. If all opcodes in a routing flit have been consumed, the first Fragment Switch that discovers the empty flit deletes that flit and processes the consecutive flits.

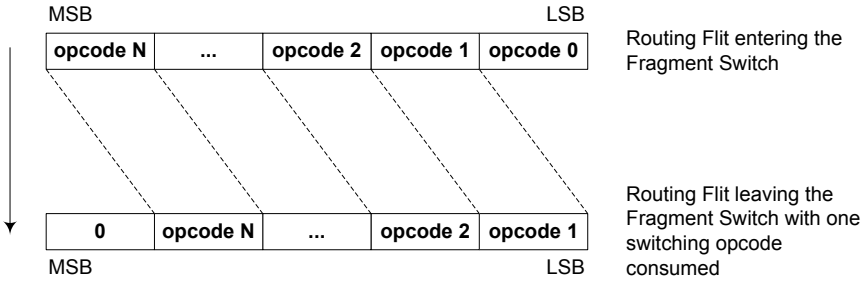


Figure 6.2: Consuming a switching opcode

Usually, the *route* of an encapsulated communication channel manifests in a concatenation of switching opcodes that are wrapped among at least one flit. Note, that

this concatenation is the most efficient approach, because it allows to place several switching opcodes into one routing flit. In contrast, the TTNoC also supports the inclusion of exactly one switching opcode for each hop in a distinct routing flit. Then, we have as many routing flits as the number of hops.

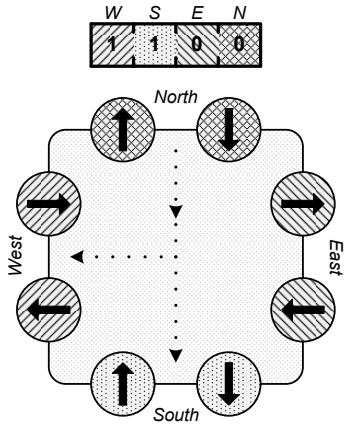


Figure 6.3: Example of a switching opcode

Figure 6.3 gives an example of a switching operation at an exemplary Fragment Switch with 4 interconnects, i.e., 8 lanes. The interconnects are named after directions: North, East, South, West. In general, a Fragment Switch with n interconnects uses an n-tuple as its switching opcode. Each bit in this n-tuple maps to exactly one interconnect. If a given bit in the opcode is set to 1, the associated interconnect is switched through from that interconnect, where that switching opcode (resp. routing flit) has arrived. Referring to the example in Figure 6.3, the switching opcode (resp. routing flit) enters at the North interconnect. With the bits of South and West set in the opcode, the Fragment Switch forwards the current routing flit and all consecutive flits to the South and West interconnects simultaneously.

If the routing information of an encapsulated communication channel has to be spread among several routing flits, the Fragment Switch considers the very first routing flit arriving at an interconnect. Thus, it extracts its switching opcode from that very first routing flit and decodes the multiplexer settings, while simply forwarding the other routing as well as data flits. The setting of a single multiplexer A in the Fragment Switch remains until the first routing flit of later routing information entering at interconnect B causes a further decoding and a new setting of the multiplexer A, so that multiplexer A switches through interconnect B. If the switching opcode at interconnect B had not set bit A to 1, then the setting of multiplexer A would not have been affected and would have remained in its previous state.

6.2.2 Routing Modes

The TTNoC supports two modes for a route to be set up.

- The routing information is placed into a continuous series of at least one routing flits preceding the data flits of a fragment. Thus, we have the notion of a *header* with routing information and a *payload* containing application data. We call this mode the *header-payload mode*[1].
- The routing information can also be proclaimed by a stand-alone continuous series of routing flits (without payload). As the multiplexer settings in the Fragment Switches remain stable unless reconfigured, the route is constructed quasi persistently. Afterwards, several fragments containing application data may follow over that route without explicit set-up. We call this mode the *circuit-switching mode*.

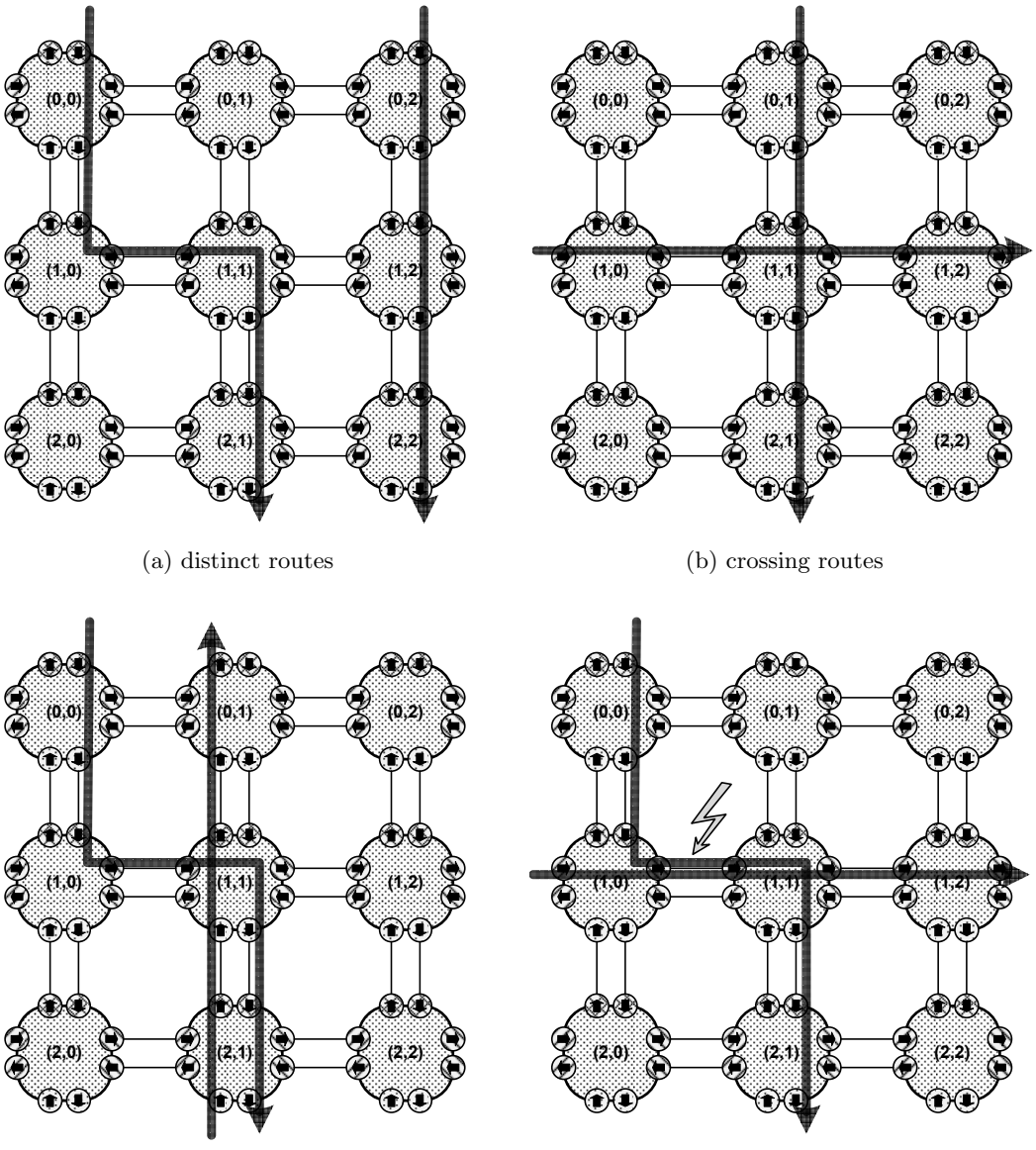
Except for the state of the multiplexer settings, routing information is never stored locally, but originates at the sender's TISS. Both modes of route set-up are supported and can even coexist. From the point of view of the TTNoC interface the sender's TISS combines the routing information with the application data. Hence, the sender affects the route, which will be taken. This is *source routing* [DT03]. Besides the Time-Triggered Communication Schedule a TISS houses the routing information for all encapsulated communication channels, which the TISS sends to. Moreover, routing information is pre-defined and trusted. This means that it is arranged in a way throughout the entire TTNoC that no collision of simultaneous routes ever happens.

6.3 Simultaneous Routes

The TTNoC supports the coexistence of several encapsulated communication channels at **the same instant of time** among the network. Figure 6.4 shows three scenarios with valid realizations of simultaneous routes and one malfunctioning. For these examples we choose an 3×3 mesh topology for easier demonstration.

Figure 6.4a considers the trivial case, whereas the routes are physically displaced, and no interference occurs at all. In Figure 6.4b the crossing of two routes at the same Fragment Switch is depicted. Here, the crossing is orthogonal, while in Figure 6.4c the routes meet at the same Fragment Switch and run in parallel over the same interconnect, but in reverse directions. This example demonstrates the full-duplex capabilities of interconnects. However, the last scenario in Figure 6.4d illustrates the conceptual limits of simultaneous routes. Here we find two routes that collide at the same interconnect, because both routes head into the same direction. Such a conflict must be resolved either by avoiding this situation at all or by means of interleaving of fragments (pulse interleaving), as mentioned in section 4.5. This constraint ensures that the TTNoC is free of contentions.

[1] Usually, this is called *worm-hole routing*.



(a) distinct routes

(b) crossing routes

(c) bi-directional routes

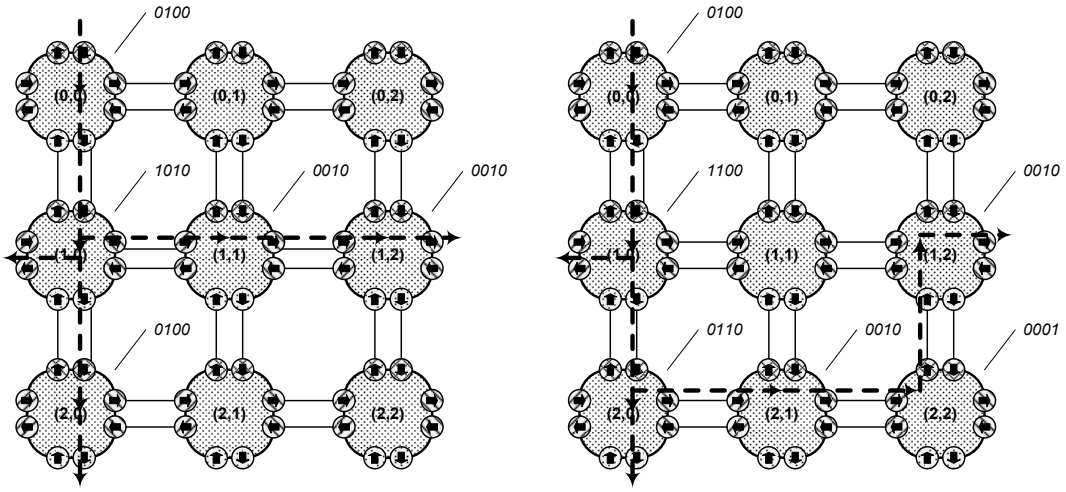
(d) colliding routes

Figure 6.4: Scenarios of simultaneous routes

6.4 Multi-casting

While multi-casting in a bus topology is trivial, in all other topologies this operation suffers from the problem, how to specify branched routes and multiple receivers. The TTNoC also faces this problem as illustrated in Figure 6.5a.

We learn from Figure 6.5a that after the splitting of the exemplary route, the switching opcodes are not identical for each branch anymore. For instance, after



(a) The multi-cast problem (b) Example of the solution concept

Figure 6.5: Multi-casting in the TTNoC

splitting at Fragment Switch $(1, 0)$ we have switching information $0010|0010 \neq 0100$. The mismatch appears in the number of switching opcodes as well as in the values of each switching opcode. Up to Fragment Switch $(1, 0)$ the mismatch of switching information is irrelevant. Afterwards, it is not possible to provide the following Fragment Switches $(1, 1), (1, 2)$, and $(2, 0)$ with the correct switching opcodes.

We give an example of a solution to this problem in Figure 6.5b. Here the branched route is modified in a way that the same destinations are reached sequentially by means of a non-split, linear route. Compared to Figure 6.5a from Fragment Switch $(1, 0)$ the route continues via Fragment Switches $(2, 0), (2, 1), (2, 2), (1, 2)$. Still, there are branches included, for example at Fragment Switch $(1, 0)$ and $(2, 0)$. However, these branches do not fork other routes that would demand distinct routing information. In contrast, they are necessary in order to reach the destinations. Nevertheless, the West interconnect at Fragment Switch $(1, 0)$ or the South interconnect of Fragment Switch $(2, 0)$, which must be a TTNoC interface each, would receive routing flits that are going to be ignored at this terminal TTNoC interface. The relevant fact is that the same data flits arrive, but we accept the arrival of possible surplus routing flits at a given TTNoC interface. We call this approach the *split point multi-casting* of the TTNoC.

The split point multi-casting solves the multi-cast problem in a contra-intuitive way, though. However, we strongly emphasize that this approach works without introducing new concepts to the switching in the TTNoC. So, the semantics implemented in the Fragment Switches remains valid, which does not require additional control logic in the Fragment Switches. The reduction of branched routes to linear routes is always applicable, because Time-Triggered Communication Schedule and routing information of encapsulated communication channels are determined *a priori*

in the TTSoC architecture. Consequently, we recognize multi-casts in advance, and can therefore transform branched routes into linear routes.

Chapter 7

The Trusted Interface Subsystem

The Trusted Interface Subsystem (TISS) is this part of a micro component that belongs to the Trusted Subsystem (TSS). The TISS realizes major parts of the TTSoC's core services, particularly the communication service. The services the TISS provides are available at the UNI for the hosts.

This chapter covers the implementation and the mode of operation of the TISS.

7.1 Structure of the TISS

This section introduces the structure of the TISS in the current implementation. Figure 7.1 shows the modules that make up a TISS. With reference to their realization in hardware (in VHDL), the TISS 's modules can be counted to 3 categories.

component A component is realized by a VHDL design. In terms of VHDL, it is a VHDL component that has been instantiated within the TISS, while the TISS is the top-level VHDL entity.

memory A memory incorporates a VHDL code stub to be instantiated within the TISS. The realization of the memory can either be generic VHDL code descriptions, or technology- or vendor-specific memory blocks, which are synthesized into the TISS.

interface The interface at the conceptual level of the TISS corresponds to a "port" of a VHDL entity. In this case, the TISS is implemented as VHDL design itself, and such an interface directly maps to some signal specifications in the VHDL entity definition, i.e., VHDL ports.

Each of these components and memories can be associated with an architectural layer, as introduced in chapter 2.4 and depicted in Figure 2.4, or other features of

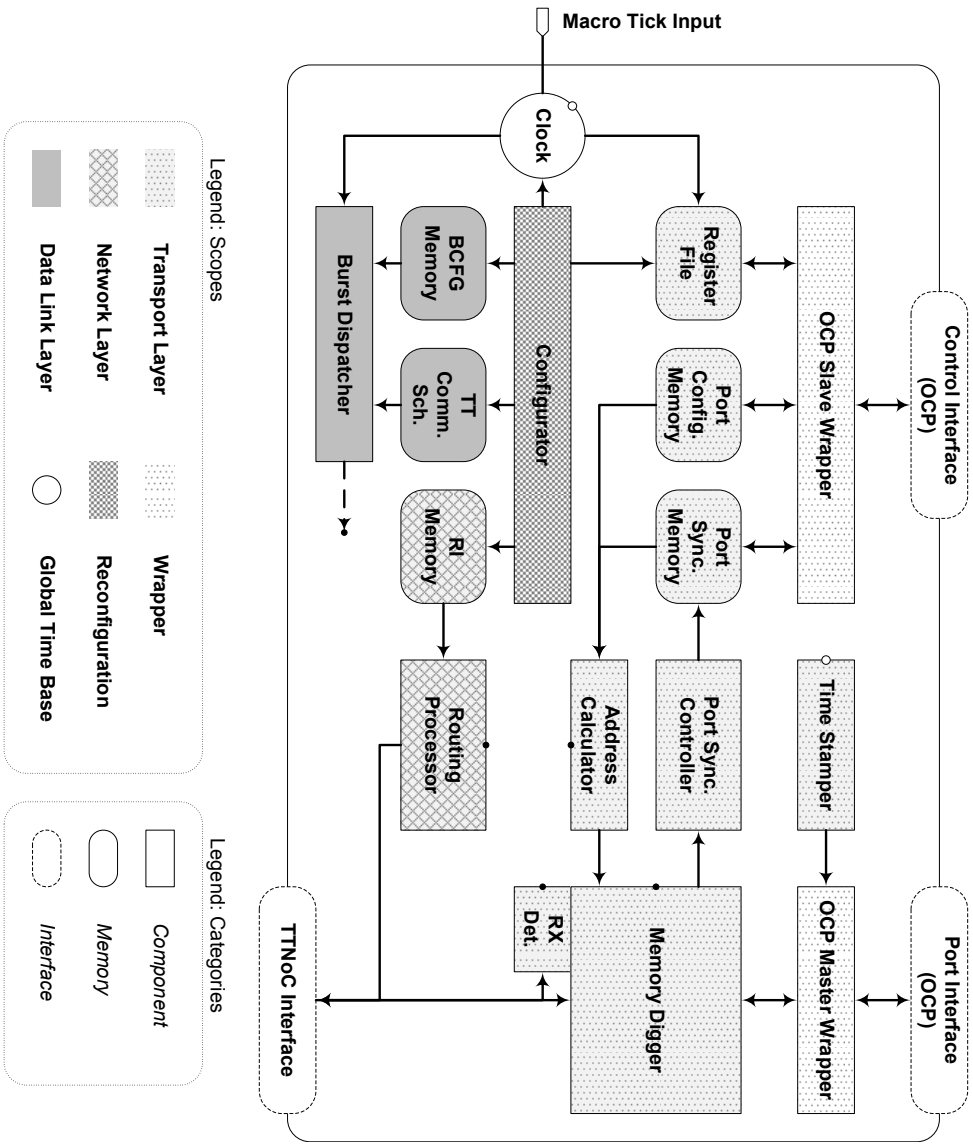


Figure 7.1: Structure of the TISS

the TTSoC architecture. We call such associations the "scopes" of the respective component or memory. Besides the 3 architectural layers (Transport Layer, Network Layer, Data Link Layer) the implementation of the TISS entails 3 more scopes, which are described in the following.

Wrapper A wrapper adapts the internal signals that are used during coding to the physical specification of interfaces. As introduced in chapter 5, the TISS provides two OCP interfaces for physical connections to hosts. Each interface requires an own wrapper to match the signal specification as well as naming convention of the OCP standard.

Reconfiguration Actually, the scope of reconfiguration logically belongs to the application layer. The only component of the TISS that refers to this scope is the special control-logic, which is necessary to realize the on-the-fly reconfiguration feature of the TTSoC architecture. The on-the-fly reconfiguration is comprehensively explained in section 8.4.

Global Time Base Generally, the global time base is used for arbitration of the TTNoC and dispatching of communication activities. Therefore, it would logically belong to the layer Data Link Layer. However, the global time base is accessible to components of other layers, too. Consequently, we define an own scope for the global time base. There is only one component that can be counted to this scope: the *clock component*. The clock component is the local replication of the global time base, which is contained in each TISS. It maintains the 64 bit counter vector, which is incremented on each rising edge of the macro tick, as introduced in section 4.3.1.

Note that the names for components and memories in Figure 7.1 are abbreviations sometimes. In the later text these abbreviations will be replaced by their full names.

7.2 Memories in the TISS

This section describes the role and the structure of some memories, which are housed in the TISS. All of these memories are implemented as *simple dual-ported RAM*, that is, they possess one read and one write port. The read port is attached to a component, which uses the information stored in the memory for processing. The write port of each memory is occupied by the Configurator (see section 8.4.3). As a result, these memories are exclusively accessible by the TISS.

In contrast, other memories reside physically in the TISS (Port Configuration Memory, Port Synchronization Memory, Register File). However, they are accessible to the host via the UNI. Also, these other memories are realized as *true dual-ported RAM*, whereas the host from the outside as well as TISS from the inside can perform read and write operations on each side simultaneously.

This section only deals with those memories, which are exclusively accessible by the TISS. The remaining memories (by definition) belong to the scope of the UNI, therefore they are comprehensively covered in chapter 5.

7.2.1 The Time-Triggered Communication Schedule

The Time-Triggered Communication Schedule plays the vital role in the TISS. It contains the information, when communication activities are initiated. To be more precise, the Time-Triggered Communication Schedule describes the instants with respect to the periodic control system, when the burst of a fragment of a pulse of a pulsed data stream, which transports a message within an encapsulated communication channel, is to be launched. Therefore, all further operation of the TISS is triggered according to the information in the Time-Triggered Communication Schedule.

Entries in the Time-Triggered Communication Schedule

Conceptually, the Time-Triggered Communication Schedule is made up of circular linked lists. Each list is associated with a period of a pulsed data stream. The entries in the lists are unique and belong to exactly one list. The information included in an entry refers to the *burst* of exactly one fragment of a pulse in a pulsed data stream, as introduced in section 4.4. As a result, each entry in each circular linked list contains a description of exactly one burst within a given period. Figure 7.2 illustrates the layout of such an entry.

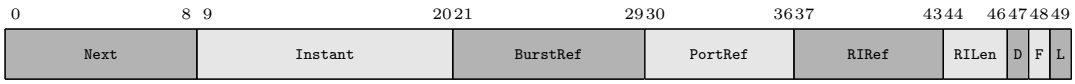


Figure 7.2: Layout of an entry in the Time-Triggered Communication Schedule

In the following we describe each field in an entry of the Time-Triggered Communication Schedule.

`Next` This field is the "next pointer". It points to the next entry of the circular linked list. In case of a single entry in the whole list, it points to itself. If the entry is the last in the list, it points to the very first entry, thus closes the circle.

The width of this field is dependent on the size of the Time-Triggered Communication Schedule. It must be able to address each data word of the memory of the Time-Triggered Communication Schedule.

In the current implementation this memory is made up of 512 data words in total. Consequently, `Next` has a width of 9 bit.

`Instant` This field denotes the phase of the fragment of the current pulse (of the pulsed data stream).

In the current implementation this field has a width of 12 bit.

`BurstRef` This field refers to a data word of the Burst Configuration Memory (see section 7.2.2). It is an address in that memory. The reference identifies a record, which contains additional information about the current burst.

In the current implementation the Burst Configuration Memory contains 512 data words in total. As a result, `BurstRef` has a width of 9 bit.

`PortRef` Like `BurstRef`, this is a reference to a record of another memory. In this case, it is an address in the Port Configuration Memory. This field links the current burst with the port (i.e., encapsulated communication channel), to which the message of the current fragment is mapped.

According to section 5.2.2, the current implementation supports 128 ports. Therefore, this field is 7 bit wide.

`RIRef` This field is a reference to routing information (see section 6.2). So, it is an address in the Routing Information Memory (see section 7.2.3). For send operations this field gives information, how to build the encapsulated communication channel, through which the current fragment has to be sent. For receive operations the value of this field has no meaning.

As the Routing Information Memory provides 128 routing flits, whereas a routing flit is an entry of the Routing Information Memory, `RIRef` has a width of 7 bit.

`RILen` Each encapsulated communication channel possesses its own routing information. While `RIRef` refers to the location of the routing information, this field states, how many routing flits the routing information is made of. Also, it is just relevant for send operations.

The width of this field is fixed to 3 bit.

`Direction (D)` This single bit field informs about the direction of the encapsulated communication channel, whether it is an input (receive operation) or an output (send operation) channel. A value of `0` indicates an input channel, in contrast a value of `1` corresponds to and output channel.

`IsFirst (F)` If the current fragment to be processed is the very first of the message associated with the current port, this single bit field is `1`, otherwise `0`. However, if the message solely consists of one fragment, this field also carries `1`.

`IsLast (L)` In contrast to `IsFirst`, this single bit field indicates that the current fragment is the last of the message associated with the current port. However, if the message solely consists of one fragment, this field also carries `1`.

If the message of a port must be split into more than one fragments, `IsFirst` and `IsLast` serve to identify the very first and the very last of those fragments. We learn from section 7.4.7 that this information is vital for the TISS to realize the port synchronization protocol, as introduced in section 5.3. The existence of `IsFirst` and `IsLast` implies an ordering of fragments, which belong to the message of the same port.

In this context, we impose the restriction that all fragments of a message must belong to the same period, i.e., must be linked within the same circular, linked list. This conforms to a characteristics of pulsed data stream that a pulse (which carries the message) is completely transmitted exactly once within its associated period.

Layout of the Time-Triggered Communication Schedule

While the previous section has introduced the layout of entries of the circular linked lists, this section explains, how these lists reside physically in the memory of the Time-Triggered Communication Schedule.

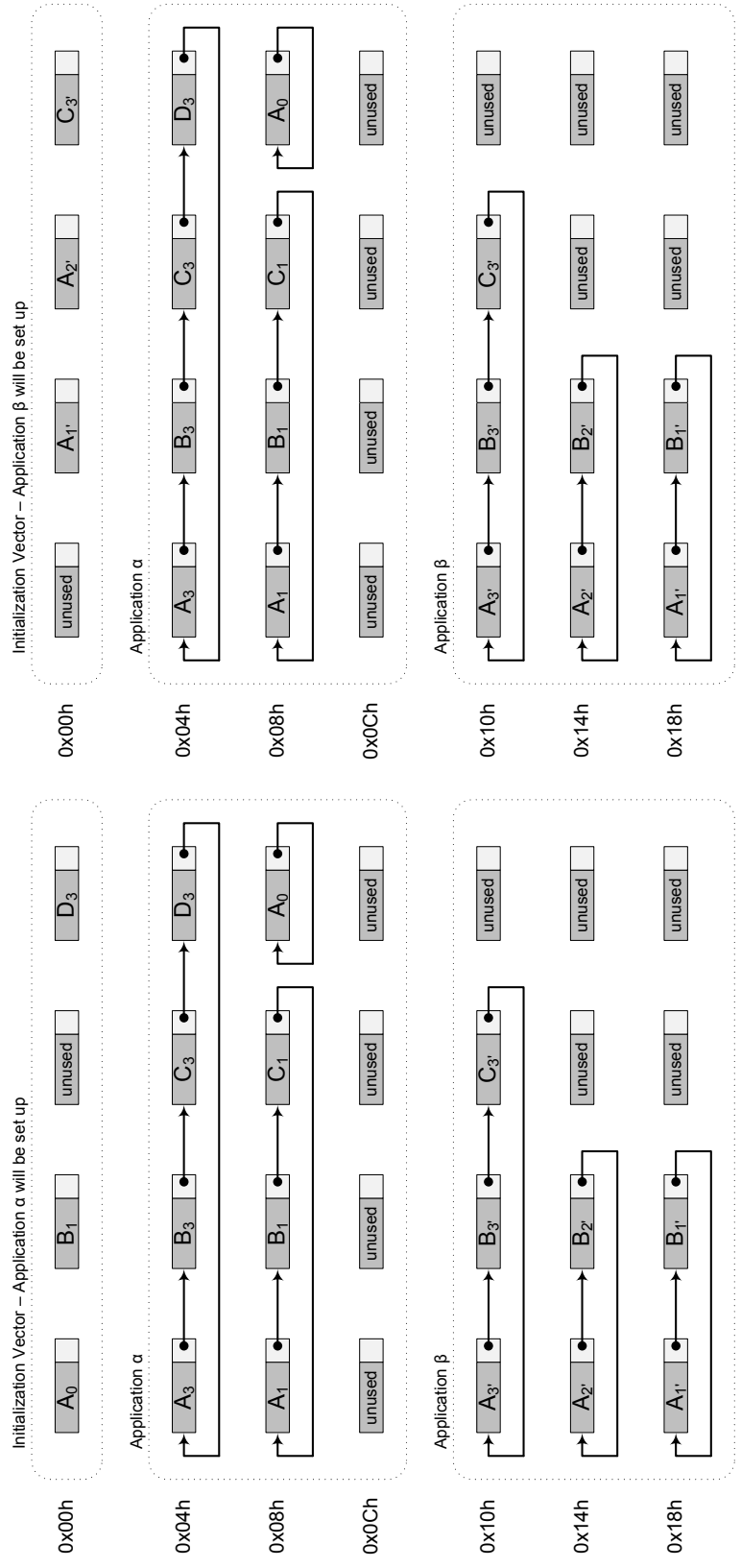
Each entry of any circular linked list resides in exactly one data word of the physical memory of the Time-Triggered Communication Schedule. Figure 7.3 depicts the memory map of the Time-Triggered Communication Schedule for a fictitious application. (Note that the grouping into rows and columns of the memory is arbitrary. Generally, the memory is sequential, and that grouping contributes to better illustration.)

The memory of the Time-Triggered Communication Schedule in Figure 7.3 comprises 3 sections.

1. Initialization Vector
2. Application α
3. Application β

Application Sections The Time-Triggered Communication Schedule is allowed to embed different configurations in an *application section*. Such a configuration corresponds to communication activities for a given primary mode or its degradation level (see section 8.2) of the TTSoC. For a given instant of time, exactly one application section is active, that is, the entries in its lists are traversed and the corresponding communication activities operated. The existence of several application sections supports the on-the-fly reconfiguration. The TISS is able to continue the processing of communication activities of the current application section, while the on-the-fly reconfiguration installs a new application section. As a consequence, no interruption of the communication service is required during reconfiguration.

The number of data words, which can be occupied by an application section, as well as their location in the Time-Triggered Communication Schedule is not defined. It can take as many data words as necessary. However, the location of the



(a) The initialization vector points to application α.

(b) The initialization vector points to application β.

Figure 7.3: Layout of the Time-Triggered Communication Schedule with two fictitious applications.

initialization vector is reserved, and therefore can not be overlapped by application sections.

Each application section contains at most as many circular linked lists as the number of supported periods of the periodic control system. Conversely, for a given configuration within an application sections some linked lists, which correspond to a specific period, can be omitted. For instance, in Figure 7.3 application α uses the periods $\{3, 1, 0\}$, but leaves out period 2. Similarly, application β omits period 0, but includes lists for all other periods $\{3, 2, 1\}$.

We also learn from figure Figure 7.3 that within an application section the arrangement of data words of list entries need not necessarily be dense. While application α organizes the data words in a sequential order, application β skips data words between lists. This arrangement has no impact on the operation of the TISS, even though it might lead to a fragmentation of the Time-Triggered Communication Schedule memory. The organisation of the Time-Triggered Communication Schedule is beyond the scope of the TISS, but in the sphere of control of the TNA during on-the-fly reconfiguration.

Each linked list is circular and closed within its application section. No entry's `Next` field points to entries of other lists (i.e., periods), nor to entries outside its own application section. Entries within a given list are numerically ordered in an ascendending way by the value of the `Instant`, which denotes the phase of the associated fragment. So, the numeric ordering equals temporal ordering. In Figure 7.3 we express the ordering by alphabetic letters. For instance, in application α the list of period 3 begins at address $0x04h$ with entry A_3. As $A_x < B_x < C_x \ldots$ for any period x, the entry B_3 is semantically the next entry of the list associated with period 3.

The numeric (as well as temporal) ordering manifests in the setting of the `Next` pointers. In Figure 7.3 the numeric ordering of entries matches the physical ordering in the memory. Generally, it is not necessary to have entries physically ordered in the memory, as the linking by the `Next` pointer always creates the proper numeric (as well as temporal) ordering.

The Initialization Vector The size and location of the initialization vector is defined in the Time-Triggered Communication Schedule. The initialization vector is relevant for initialization of the TISS (see section 7.6). In case of initialization, the initialization vector presents the entry point of traversal for each period. The lists associated with these periods must belong to exactly the same application section.

The initialization vector contains a copy of the first entry to be processed of each period. For period x this copy resides at address x of the memory. If a period is not present in the application sections, which the initialization vector refers to, the corresponding data word in the initialization vector is also unused. For instance, in Figure 7.3a the initialization vector points to application α, which omits period 2. Consequently, the data word at address $0x02h$ is unused. Similarly, in Figure 7.3b, where the initialization vector refers to application β, period 0 is not present, and therefore the data word at address $0x00h$ remains unused.

The initialization vector always consumes as many data words as the number of supported periods of the periodic control system. For example, in Figure 7.3 we support 4 periods, hence an application section can possess at most 4 circular linked lists. Consequently, the initialization vector takes the very first 4 data words in the memory of the Time-Triggered Communication Schedule, even though there might be unused data words in the initialization vector on behalf of omitted periods in any application section.

An entry of the initialization vector marks the entry point, where a given period should be started to be processed. This need not necessarily be the "first" entry A with the lowest `Instant` field. For instance, in the initialization vector of Figure 7.3a at address 1 lays a copy of B_1. Consequently, period 1 will not be started at A_1, but at B_1. According to its ordering in the list, after the processing of B_1 the next entry to be handled will be C_1. As a result, at initialization of the TISS we can begin to execute a period at any communication activity associated with that entry.

7.2.2 The Burst Configuration Memory

The Burst Configuration Memory is an auxiliary memory in the current implementation of the TISS. It contains additional information about bursts of fragments, which has not been included in the memory of the Time-Triggered Communication Schedule on behalf of implementation issues. Semantically, the information contained in the Burst Configuration Memory belongs to the context of the Time-Triggered Communication Schedule. The main purpose of the Burst Configuration Memory is to cut down on the size of the multiplexers in the Burst Dispatcher. As we see in section 7.3.1, most parts of a data word of the Time-Triggered Communication Schedule flow through a multiplexer. When we outsource parts of the information in the context of the Time-Triggered Communication Schedule to an own memory, the outsourced information need not be multiplexed any more.

Figure 7.4 depicts the layout of a data word in the Burst Configuration Memory. Actually, the data word consumes 30 bit. The 2 additional bit that make up a complete 32 bit word are reserved for future purpose. In total, the Burst Configuration Memory provides 512 data words.

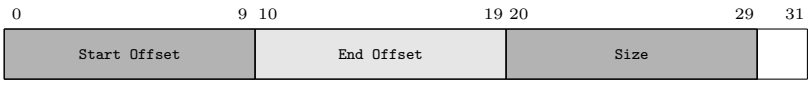


Figure 7.4: Layout of a data word in the Burst Configuration Memory

In the following, we explain the fields in a data word of the Burst Configuration Memory.

`Start Offset` A burst corresponds to a fragment of a message, which is accessible at the port of an encapsulated communication channel. Physically, the messages of a port reside in the Port Memory outside the TISS. This field denotes the

start offset of the current fragment within the Port Memory with respect to the port base address[1].

`End Offset` Contrary to `Start Offset`, this field is the end offset of the current fragment within the Port Memory with respect to the port base address.

`Size` This field has different interpretations according to the type of ports.

- For event ports, `Size` gives the size of an event message in number of continuous data words in the Port Memory.
- In the other case, `Size` stands for the number of continuous data words in the Port Memory, which the corresponding state message occupies.
- When explicit synchronization is chosen for output state ports, then this field denotes the number of continuous data words in the Port Memory, which one shadow buffer for a complete message takes. For implicit synchronization, `Size` states the number of continuous data words in the Port Memory for the complete message.

This field does not care about the increase of memory, when the time stamping service for the a given port is activated, nor does it concern possible shadow buffers. In other words, `Size` means the *net size* of a complete message of a port.

The offsets (`Start Offset` and `End Offset`) mark the beginning and the end of the burst. Taking the port base address into account, the offsets comprise a continuous chunk, which equals the current fragment of the message of the associated port, of data words in the Port Memory. According to the fragmentation of messages, there is one entry in the Burst Configuration Memory for each fragment. The conjunction of start and end offsets must span over the whole message, so that all chunks make up the complete message. Note that the order of chunks is not relevant according to the property of total temporal ordering of encapsulated communication channels. However, an intersection between any chunks built by the offsets is not allowed. Otherwise, there would be data words in the Port Memory, which are processed more than once.

All fields in the data word of the Burst Configuration Memory are input for the address calculation (see section 7.4.4), which determines the physical address of the data words in the Port Memory for the current fragment. All of them have a width of 10 bit, thus each field can address $2^{10} = 1024$ offsets, which refer to a data word in the Port Memory.

Like the Time-Triggered Communication Schedule, the Burst Configuration Memory is in the authority of the TNA. Therefore, the set-up information of bursts in the data words of the Burst Configuration Memory are also determined *a priori* and regarded as fault-free, i.e., no chunks embraced by start and end offsets do overlap.

[1] See the corresponding field in the Port Configuration Memory in section 5.2.2

7.2.3 The Routing Information Memory

The Routing Information Memory belongs to the Network Layer. It stores the information, how encapsulated communication channels are routed through the TTNoC: the routing information, as introduced in section 6.2. The Routing Processor reads the data of the Routing Information Memory, and inserts the routing information into the TTNoC, so that the route of the encapsulated communication channel is established due to the switching of the Fragment Switches according to the routing information.

The Routing Information Memory contains 128 data words of 32 bit width in the current implementation, which is the same as the width of the data bus of lanes in the TTNoC. The data words do not possess any particular layout. Instead of a predefined structure, each data word holds a routing flit, which is composed of switching opcodes for Fragment Switches. The routing information that describes the route of an encapsulated communication channel is spread across at least one routing flit respectively data word of the Routing Information Memory.

The Routing Information Memory provides the input to the Routing Processor only for send operations, when the TISS takes the role of the source (sender) of the encapsulated communication channel, and consequently is in charge of setting up the route. In this case, two fields in the entries of the Time-Triggered Communication Schedule are relevant to refer to the proper routing information in the Routing Information Memory. As the field `RILen` in the Time-Triggered Communication Schedule has a width of 3 bit (and the value 000 is reserved), the routing information of an encapsulated communication channel can consist up to 7 routing flits, which occupy 7 continuous data words in the Routing Information Memory. The field `RIRef` in the Time-Triggered Communication Schedule points at the very first data word of this sequence.

7.3 Dispatching Bursts

This sections explains, how the instant, when a burst is to be processed with respect to the global time base, is determined in the TISS. We call this function the *dispatching* of bursts. The component in the TISS, which is in charge of dispatching, is the *Burst Dispatcher*.

The Burst Dispatcher retrieves the information about instants of bursts from the Time-Triggered Communication Schedule. Therefore, the Burst Dispatcher is directly attached to the read port of the memory of the Time-Triggered Communication Schedule. Moreover, the operation of the Burst Dispatcher harnesses the organisation of the Time-Triggered Communication Schedule in circular, linked lists and their entries. In the following sections we introduce the components, which the Burst Dispatcher is built of, and explain, how these components use the Time-Triggered Communication Schedule.

7.3.1 Structure of the Burst Dispatcher

The main components of the Burst Dispatcher are the *phase comparators* and the *period controllers*. One pair of phase comparator and period controller is assigned to exactly one period in order to process the bursts of that period. While the phase comparator checks the arrival of the instant of the current burst, the period controller traverses the circular, linked list in the Time-Triggered Communication Schedule, which goes with the corresponding period. The cooperation of these two kinds of components is the key mechanism of dispatching in the TISS.

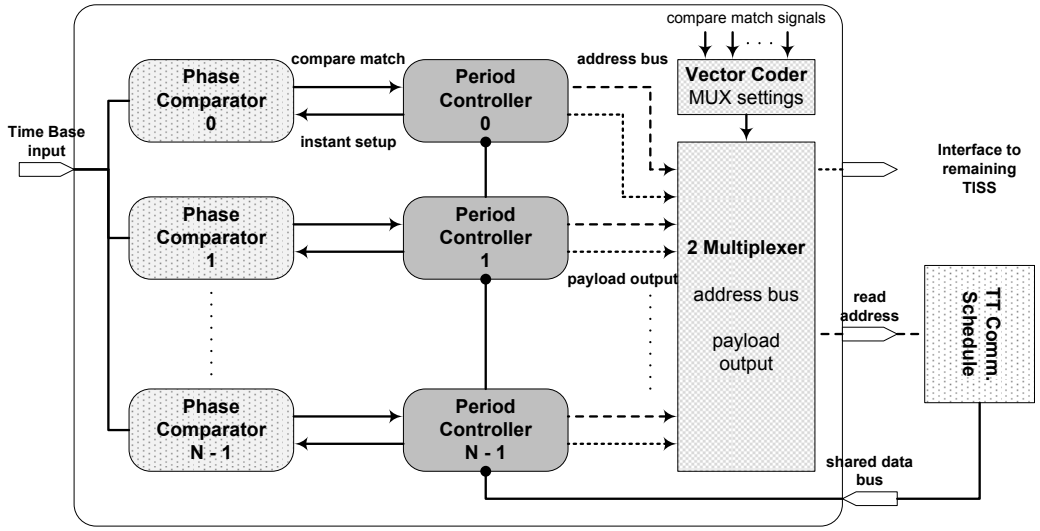


Figure 7.5: Structure of the Burst Dispatcher

We see in Figure 7.5 that the Burst Dispatcher contains as many pairs of phase comparator and period controller as the number of supported periods N of the periodic control system. Other than the Time-Triggered Communication Schedule, which is realized in a memory and can omit periods, these components exist in hardware. Even though a period might not be active in the current application section (of the Time-Triggered Communication Schedule), the associated phase comparator and period controller are still present, but remain in an idle mode.

In the following we examine the operation of a phase comparator and period controller.

The Phase Comparator

The task of the phase comparator is to indicate the instant, when a burst has to be processed by the TISS. The required information about instants is kept in the entries of the Time-Triggered Communication Schedule. The `Instant` field holds the value of the phase slice of a given period that is associated with a burst's phase in that period. Whenever the phase slice (in the time format of the global time base) of

that period equals the value of `Instant`, the instant for processing of the burst has arrived. So, a phase comparator is attached to the phase slice of the corresponding period. Moreover, it contains a register with a local copy of the value of `Instant` of the next burst to be dispatched.

In this context, the phase comparator performs a bit-wise comparison between the value of the local copy of `Instant` and the phase slice it is attached to. When this comparison evaluates to equality, the phase comparator indicates a *compare match* for exactly one clock cycle of the system operation frequency. This internal control signal triggers the further processing of the burst within the Burst Dispatcher as well as the TISS.

The Period Controller

The period controller serves two purposes for dispatching.

1. The period controller traverses the circular, linked list in the Time-Triggered Communication Schedule, which is associated with the same period, to which its paired phase comparator and itself are assigned to. In this context, it continuously fetches the entry corresponding with the next burst to be processed from the list, and makes a local copy in a register.

2. The period controller maintains a register in its paired phase comparator, which stores the local copy of the `Instant` field for the next burst to be processed. As a result, its paired phase comparator indicates the compare match according to this local `Instant` value, which has been set by the period controller.

These two tasks are executed sequentially, after a compare match has been signaled by the paired phase comparator. In this case, the period controller fetches the next entry in the circular, linked list, which corresponds to the burst to be dispatched next. It stores a local copy of that next entry and extracts the field `Instant` in order to set up a copy in the phase comparator. When the instant of the next burst occurs, this procedure starts all over again. Consequently, the period controller traverses the circular, linked list, whereas it hops from one entry to the successor in the list after each compare match.

A period controller provides four connections to other components of the Burst Dispatcher:

1. the link to its paired phase comparator, through which it supplies the register in the phase comparator with copies of the `Instant` field of the next burst ("instant setup" in Figure 7.5).

2. an "address bus" to the memory of the Time-Triggered Communication Schedule in order to issue the address of the next entry to be fetched.

3. a corresponding input "data bus", on which the next entry is carried from the memory of the Time-Triggered Communication Schedule to all period controllers.
4. a "payload output", on which it provides relevant information about the current burst to be further processed by the TISS.

The period controller achieves the traversal of the list by evaluating the `Next` field in the current entry. According to the layout of entries in the Time-Triggered Communication Schedule, a `Next` field is a pointer to the next entry in the circular, linked list. In this context, the `Next` field denotes an address in the memory of the Time-Triggered Communication Schedule. In fact, the period controller fetches the entry of the next burst to be dispatched by assigning the `Next` field of its local copy of the current entry at its address bus.

Besides setting the address bus after the compare match to fetch the next entry, the period controller simultaneously dumps the relevant information for further processing of the current burst on its payload output. The Burst Dispatcher further maintains the payload output (as we will see in the following section) in order to establish an internal (unnamed) interface to the remaining TISS. This interface provides the relevant information about the current burst to the other processing components of the TISS.

This internal (unnamed) interface to the remaining TISS comprises the following fields of an entry of the Time-Triggered Communication Schedule:

- `BurstRef`
- `PortRef`
- `RIRef`
- `RILen`
- `Direction`
- `IsFirst`
- `IsLast`
- an additional signal `InfoValid`, which indicates that the information in the other fields at the interface are valid. `InfoValid` carries a level of 1 for the very first clock cycle of the system operation frequency, when the payload output of the period controller, which paired phase comparator most recently has fired, has arrived at the Burst Dispatcher's interface.

Note that `Next` and `Instant` are not part of the Burst Dispatcher's interface to the TISS, because these fields have no utility outside the Burst Dispatcher. `Next` only concerns the fetching of the next entries from the Time-Triggered Communication Schedule, and therefore is only used on the address bus of period controllers. `Instant` is only relevant between a period controller and its paired phase comparator.

Multiplexing in the Burst Dispatcher

As there is one pair of phase comparator and period controller replicated for each supported period, the dispatching of the current burst of each period is performed simultaneously throughout the parallel hardware. In fact, only one phase comparator will fire at a given instant of time. The reason for this is that all pairs of phase comparator and period controller strictly follow the Time-Triggered Communication Schedule, and communication activities among all periods never collide according to the requirement of section 2.3.3.

The Burst Dispatcher provides information of the current burst at an internal (unnamed) interface to the remaining TISS. This information comprises a subset of the fields of an entry in the Time-Triggered Communication Schedule, which are relevant for further processing. However, we have several period controllers in parallel, and each of them frequently supplies such information at its payload output, whenever a burst occurs in its dedicated period. As a consequence, the payload output of all period controllers must be multiplexed, so that the information from the period controller with the most recent burst is available at the Burst Dispatcher's interface. Figure 7.5 also illustrates, how the payload output of each period controller flows through a multiplexer. The output of that multiplexer is forwarded to the interface of the Burst Dispatcher.

The address buses of the period controllers are in a similar situation like the payload outputs. There is only one memory of the Time-Triggered Communication Schedule, but every period controller needs to assign addresses to the address port of that memory. Hence, a driver conflict among the wires of all address buses from the period controllers is inevitable. To circumvent this situation, the Burst Dispatcher includes a second multiplexer, which directs the address bus of the period controller, which paired phase comparator most recently has fired, to the read port of the memory. Note that the data bus of the memory, which carries the next entries to be fetched by the period controllers, need not be multiplexed. The reason is that period controllers just read the data bus of the memory, while there is only the memory itself as single driver.

The setting of the multiplexers is derived from the collection of all compare match wires, which originate at phase comparators. Note that both multiplexers simultaneously select the address bus and payload output of that period controller, which paired phase comparator most recently has triggered the compare match. So, there need only be one setting, which is applied at both multiplexers.

The collection of all compare match wires forms a signal vector, whereas signal with index i corresponds to the compare match wire of the phase comparator associated with period i. As there can only fire at most one phase comparator at a given instant of time, this vector either has all signals to 0, or at most one signal carries level 1. Let us denote the index of the signal, which carries 1 at the moment, as m, while the numeric value of m corresponds to period m. As a consequence, we want the multiplexers to select the address bus and payload output of period controller m. For this purpose, the multiplexer setting must take the numeric value of m. In other

words, the Burst Dispatcher has to implement a function f_c, which transforms the bit pattern of the signal vector (where the m-th bit is 1) into the numeric value of index m. For example we assume 16 periods, and the 8-th phase comparator has most recently fired ($m = 8$). The signal vector looks like this: $\vec{s} = 0000000100000000$. The function f_c, which transforms $\vec{s}$ into m, calculates: $0000000100000000 \rightarrow 1000$. In other words, f_c determines the logarithm dualis of the signal vector, which then is used as the setting of the multiplexers.

The Burst Dispatcher includes an own component named *vector coder*, which implements f_c. The input of the vector coder are all compare match wires, and the output is the proper setting of the multiplexers.

7.3.2 Timing of Dispatching

This section explains the timing in the Burst Dispatcher. It shows, which activity that has been described in the previous section happens where at which time after the arrival of the instant of a burst. Figure 7.6 gives the latency of each activity in cycles of the system operation frequency.

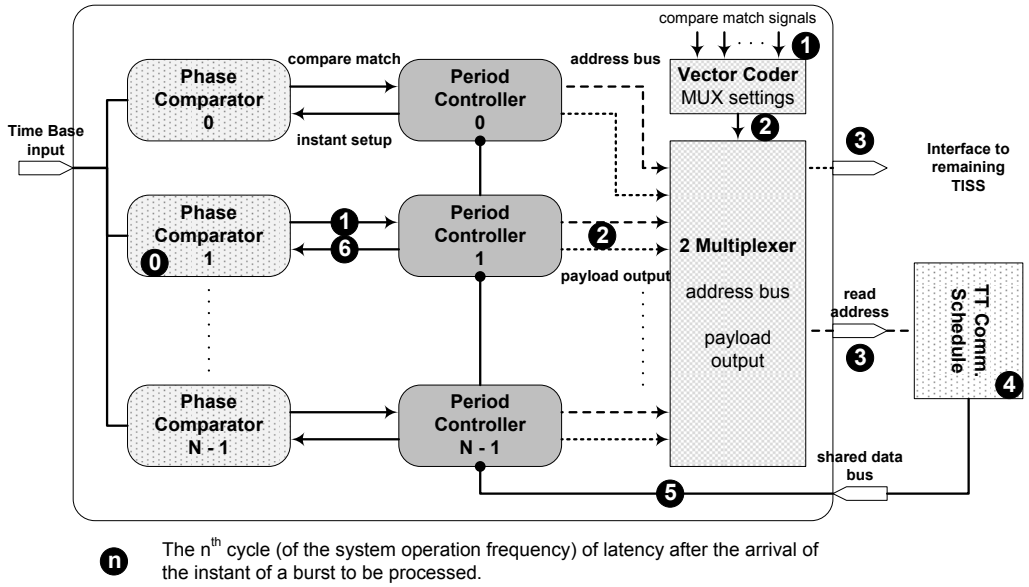


Figure 7.6: Timing of activities in the Burst Dispatcher

Even though Figure 7.6 shows an example scenario, where a burst is dispatched for period 1, the dispatching of burst among the phase comparators and period controllers of all periods is deterministic and takes the same time.

In the following we describe the sequence of activities with respect to their timing.

0. This is the initial state in the phase comparator, when the bit-wise comparison between attached phase slice and the local copy of the `Instant` value evaluates

to equality. It takes 1 clock cycle for the phase comparator to process its internal state machine and to produce the compare match signal.

1. The compare match signal travels to the period controller. The period controller responds to this input by issuing the payload output and placing the address (`Next` "pointer") hold in the local copy of the current entry on the address bus. This causes the next entry in the circular, linked list of the associated period in the Time-Triggered Communication Schedule to be fetched from the memory. Afterwards the period controller enters wait states, until that next entry arrives at the data bus, which is shared among all period controllers.

   At the same time the compare match signal also enters the vector coder, which generates the proper multiplexer setting.

2. The payload data and the address of the next entry from the period controller are on its way to their corresponding multiplexers. The vector coder has already calculated the setting of both multiplexers, therefore the multiplexers are prepared to select the correct payload output respectively address bus. In the example of Figure 7.6 this is the payload output and address bus of period controller 1.

3. The payload output and address bus leave the multiplexers. The payload data directly flows into the internal (unnamed) interface to the remaining TISS. The additional control signal at the Burst Dispatcher's interface named `InfoValid` goes to 1 during this clock cycle. As a result, the TISS is notified and can proceed with the processing of the burst after these 3 cycles.

   On the other side the read port of the memory of the Time-Triggered Communication Schedule is assigned with the address of the next entry in the circular, linked list of the corresponding period.

4. This is a wait state for the period controller, as the memory of the Time-Triggered Communication Schedule requires one clock cycle to process the address, which has been set up in the previous clock cycle.

5. The next entry is available on the shared data bus from the memory of the Time-Triggered Communication Schedule. According to its internal state, the period controller, which phase comparator has most recently fired (in this example period controller 1), fetches that data into a local register. Additionally, it extracts the `Instant` field in order to copy its value to the phase comparator.

6. The period controller sets the value in the local register for the `Instant` field in its paired phase comparator. As a result, the phase comparator is ready to indicate the arrival of the next burst in the period.

We learn from Figure 7.6 that the relevant information about the burst, which is used for further processing by the TISS, arrives at the Burst Dispatcher's interface to the remaining TISS before the period controller has completed the set-up of the

next burst to be dispatched. Consequently, that set-up procedure overlaps the further processing of the burst in the remaining TISS. We define this interval between arrival of the instant of the associated burst till the availability of relevant information at the Burst Dispatcher's interface as $\Delta_{disp} = 3$.

The total duration of that set-up procedure determines the lower bound, how often a burst can be dispatched for a given period, i.e., by the same pair of phase comparator and period controller. In the current implementation this total duration is 6 cycles of system operation frequency. Thus, a given pair of phase comparator and period controller can dispatch two consecutive bursts every 7 cycles of system operation frequency.

Theoretically, it is possible to have a different phase comparator firing every clock cycle, so that the single activities among all active period controllers and phase comparators are pipelined. In this context, the essential condition is that exactly one phase comparator indicates the arrival of a burst at a specific clock cycle, thus, no collisions in the vector coder as well as the multiplexers ever occur afterwards.

7.3.3 Activating and Deactivating Periods

As introduced in section 7.2.1, the Time-Triggered Communication Schedule can omit circular, linked lists in its layout, if the corresponding period is not used in the current application section. This flexibility is possible due the realization of the Time-Triggered Communication Schedule as a memory.

The Burst Dispatcher can not provide such a flexibility. Its internal components such as the phase comparators and period controllers are realized in hardware, therefore they are always present at run-time and can not simply "disappear".

However, the Burst Dispatcher offers a mechanism to activate and deactivate pairs of phase comparator and period controller independently of each other. The deactivation is only reasonable, if the corresponding period is not available in the current application section of the Time-Triggered Communication Schedule. On the contrary, such a pair of components should be activated in case of a circular, linked list present in the current application section. Otherwise, the Burst Dispatcher would fail to dispatch the bursts of that period.

The activation and deactivation is not automatically executed depending on the layout of the current application section in the Time-Triggered Communication Schedule. This must explicitly be done by the TNA, which is able to activate and deactivate pairs of phase comparators and period controllers for the Burst Dispatcher of all TISSs independently.

For this purpose, each TISS contains a vector of control signals named `PeriodEna`, which is wired through to the phase comparators in its Burst Dispatcher. The width of that vector is the number of supported periods of the periodic control system, which is the same as the number of pairs of phase comparator and period controller

in the Burst Dispatcher. So, the task of the TNA is to maintain the vector `PeriodEna` in each TISS[2].

If `PeriodEna(i)` = 1 for the phase comparator associated with period i, then the phase comparator and period controller for period i are active, and the bursts of period i can be dispatched. Otherwise if `PeriodEna(i)` = 0, phase comparator i suppresses the generation of a compare match signal. As a consequence, no compare match signal ever arrives at the paired period controller, therefore no circular, linked list in the Time-Triggered Communication Schedule is traversed any more.

7.4 The Port Manager

In this section we discover, how the TISS manages the data flow of bursts and reassembles this burst to complete messages in the Port Memory. It explains, which calculations and internal states are processed in the single components of the TISS, and which interactions among these components of the TISS are involved. With reference to the schematics of Figure 2.4, this set of functions logically belongs to the Port Manager in the TISS.

7.4.1 The State Machine of the Port Manager

In general, the operation of the Port Manager is driven by a state machine, as illustrated in Figure 7.7. The state machine includes 5 states (excluding the `Reset` state), which are processed in a cyclic way. The begin of such a cycle of processing corresponds to the dispatching of a burst. Even though the operation of the Port Manager is driven by a state machine, the sequence of states is triggered by the dispatching of bursts according to the Time-Triggered Communication Schedule.

The Reset State

The very first state (as well as starting state) in the state machine is the `Reset` state. This only covers the reset condition concerning the hardware, which houses the TTSoC. During this state all internal registers are initialized to their start-up values. Additionally, the `Reset` state keeps the state machine of the Port Manager in a controlled condition so that no unintended behaviour of the TISS can occur. Usually, the `Reset` state lasts as long as a dedicated reset wire implemented in hardware drives a level of 0, thus the corresponding reset signal is low-active. This level of the reset signal can be caused by user input, for instance the user pushes a reset button. As a consequence, the state machine, i.e., the processing of the Port Manager, can be interrupted at any time by user interaction with the target hardware. In such a case, the `Reset` state is engaged, and all pending operation is aborted.

[2]This is subject of section 8.4.3.

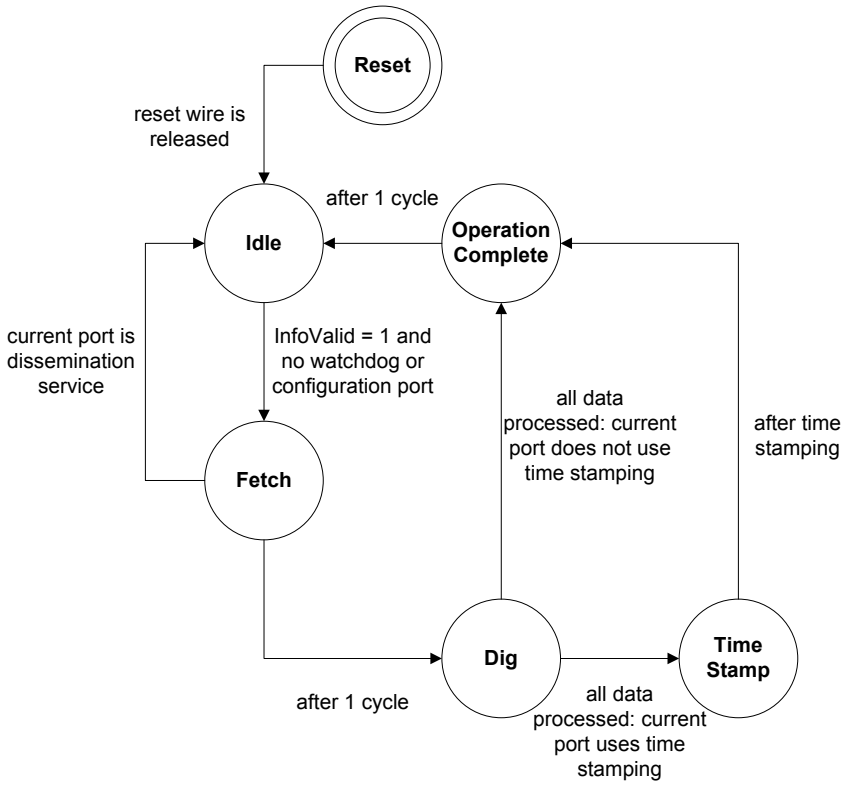


Figure 7.7: State machine of the Port Manager

The `Reset` is present in all state machines throughout all components of the TISS and serves the same purpose. Therefore, we will not further mention the activities during the `Reset` state in each component.

The Idle State

After the `Reset` state, when the TTSoC has completely reached its start-up condition, the state machine enters the `Idle` state. As its name suggests, then the Port Manager is idle and waits for the start of a new processing cycle of the state machine. In fact, this start event is derived from the Time-Triggered Communication Schedule in the TISS. As the Port Manager handles the data flow of bursts, the state machine begins its cyclic processing whenever the Burst Dispatcher indicates the arrival of an instant associated with a burst. As we have seen in section 7.3, this start event corresponds to that clock cycle of the system operation frequency, when the Burst Dispatcher issues a level of 1 at the signal `InfoValid` of its internal (unnamed) interface to the remaining TISS. In case of the Port Manager, `InfoValid` is the input notification signal, which causes the state machine to awake from the `Idle` state. Note that `InfoValid` not always causes the `Idle` state to be left.

In the current implementation, the TISS includes a special handling for specific operations. The state machine of the Port Manager recognizes the special handling by monitoring the associated port of the current burst (given in the field `PortRef` of the Burst Dispatcher's interface). If `InfoValid` indicates a burst for a port belonging to this special set, then the operation of the state machine is skipped, and dedicated components in the TISS take the control to fulfil that special operation. Currently, the state machine of the Port Manager recognizes the following 3 port identifiers as specially treated ports:

port 125 This is the "virtual" port associated with the watchdog service.

port 126 This is the port for the diagnostic dissemination service.

port 127 The reconfiguration, as explained in section 8.4, uses port 127 to execute the reconfiguration procedure.

If `PortRef` holds a value of 125 or 127, then no transition from the `Idle` state to its successor in the state machine takes place, because in this case a dedicated module in the TISS processes the special operation. Also, port 126 causes the state machine to abort, but as we can see in Figure 7.7 from the state `Fetch`.

The `InfoValid` signal not exclusively triggers progress in the state machine of the Port Manager. In combination with the `Idle` state, a level of 1 at `InfoValid` activates further modules of the Port Manager, which are in charge of one specific function that contributes to the overall function of the TISS, i.e., the Port Manager. Depending on the direction of the burst, i.e., send or receive operation, the following components take up their operation, which have been first mentioned in Figure 7.1:

Routing Processor is only relevant for send operations. It injects routing flits before the application data of a fragment so that the route of the encapsulated communication channel, where the current burst belongs to, is built (see section 7.4.2).

Receive Window Detector (RX Det.) is active during receive operations. When a burst is dispatched, it observes the TTNoC interface until the very first data flit of the burst to be received arrives (see section 7.4.3).

Memory Digger keeps track of the number of data words from or to the Port Memory. The Memory Digger handles the burst of send as well as receive operations (see section 7.4.5).

The Fetch State

Some fields of the Burst Dispatcher's interface drive the address bus of memories in the TISS. When the Port Manager is in the `Idle` state and `InfoValid` goes 1 for exactly one clock cycle of system operation frequency, all of these fields have already

reached a stable value. As a result, each memory processes its address bus and puts the appropriate data word onto its data output bus. Each memory takes one clock cycle of system operation frequency to provide the corresponding output, starting with the clock cycle, when `InfoValid` becomes 1. After that, it also takes one clock cycle for the Port Manager's state machine to reach the `Fetch` state. At the same time, the output data at each memory is ready to use.

The following fields of the Burst Dispatcher' s interface flow to the respective memory:

- `PortRef` is attached to the address bus of the Port Configuration Memory as well as Port Synchronization Memory.
- `BurstRef` goes to the address bus of the Burst Configuration Memory.
- `RIRef` is used in the Routing Processor, and indirectly used to drive the address bus of the Routing Information Memory.

In the `Fetch` state, the Port Manager makes a copy into local registers of the data words of the Port Configuration Memory and Port Synchronization Memory, which has become available at the data output of the corresponding memory. The information contained in the data words, and further in the local register, is used throughout the further operation of the Port Manager. The data word of the Routing Information Memory is managed within the Routing Processor.

The Dig State

During the `Dig` state, the operation in the Port Manager is executed by dedicated modules. The Port Manager completely hands over control to these entities, and just waits for the completion of their operation in order to proceed with the state machine. The central role during the `Dig` state plays the Memory Digger. It is active for send as well as receive operations, i.e., bursts that process the fragments of outgoing or incoming messages of a given encapsulated communication channel.

The Time Stamp State

The state machine of the Port Manager only reaches this state, if the host has enabled the time stamping service for the port, where the current burst belongs to. Furthermore, the current burst must correspond the the last fragment (i.e., `IsLast` = 1) so that the message will be completed after that burst. During this state, the Time Stamper component of the Port Manager appends a copy of the current value of the global time base to the data of the message.

The Operation Complete State

The state machine occupies this state after the complete processing of the current burst. At this time, the Memory Digger and probably the Time Stamper have completed their operation, and now the control is released back to the Port Manager. The state `Operation Complete` is the last state, before the state machine in the Port Manager finishes a clock cycle and gets back to the `Idle` state.

This state `Operation Complete` is used for some housekeeping in the Port Manager. Firstly, it triggers the updating of port synchronization flags. Secondly, if "port operation complete" interrupts are activated for the current burst, and the current burst corresponds to the last fragment of the message of the current port (`IsLast` = 1), then the Port Manager indicates a "port operation complete" interrupt at the control interface of the UNI. Finally, this state causes some internal registers to be reset in order to be prepared to process the next burst.

7.4.2 The Routing Processor

The Routing Processor manages the insertion of routing flits from the Routing Information Memory to the TTNoC interface in order to set-up the route of an encapsulated communication channel. For this purpose, the Routing Processor maintains two counters:

- an *address counter*, which drives the address of the Routing Information Memory
- a *flit counter*, which counts the number of routing flits that are injected into the TTNoC via the TTNoC interface

Actually, the Routing Processor is driven by an own state machine, which is illustrated in Figure 7.8. The operation of the Routing Processor is initiated by the signal `InfoValid` (driven to 1) of the Burst Dispatcher's interface, if and only if `Direction` is also 1, hence, the current burst belongs to an output port. Then, the state machine leaves the `Idle` state and enters the `Setup` state.

During the `Setup` state, the Routing Processor captures the field `RIRef` of the Burst Dispatcher's interface into the local address counter, and resets the flit counter to 0. Besides this, the Routing Processor controls, whether its operation is really required. The Routing Processor skips some states in the state machine or goes back to the `Idle` state, if the following conditions are fulfilled:

- `Port Enable` in the Port Configuration Memory is 0. In other words, the port to which the current burst belongs to is deactivated, and no communication activities referring to this port should take place at all. The state machine aborts operation and goes back to the `Idle` state.

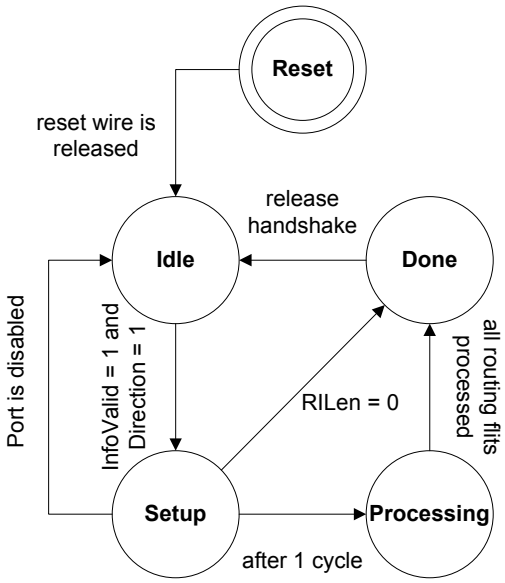


Figure 7.8: State machine of the Routing Processor

- `RILen` equals a null vector, which tells the Routing Processor that no routing flits are associated with the current burst. For instance, this is the case for the circuit-switched routing mode of the TTNoC, or for "virtual fragments" that are used to ensure the consistent delivery order of messages. Then, the state machine continues in the state `Done`.

If the state machine has not been aborted so far, the Routing Processor continues in the `Processing` state. In this state the Routing Processor increments the address counter and the flit counter in each clock cycle of the system operation frequency. As the address counter is attached to the address port of the Routing Information Memory, the increment of the address counter simultaneously causes the next data word (and therefore the next routing flit) to be fetched from the Routing Information Memory. As a result, the Routing Processor traverses the sequence of routing flits that make up the route of the encapsulated communication channel, to which the fragment of the current burst belongs to. The state machines leaves the `Processing` state, when the address of all routing flits for the current burst have been issued. This is the case, when $flit\ counter = \texttt{RILen}$.

Afterwards the Routing Processor remains in the `Done` state. The Routing Processor signals that the TISS has just completed to insert routing flits at the TTNoC interface, and that application data of the burst can be transmitted. This signal named `RIDone` is one part of a handshake between modules of the Port Manager, which postpone their operation until the arrival of this notification from the Routing Processor. For instance, for send operation the Memory Digger waits until this handshake signal arrives from the Routing Processor and then takes up its own operation. In this case, the Memory Digger responds with another handshake signal named

`RelDone` (see section 7.4.5), which tells the state machine of the Routing Processor to go back to the `Idle` state.

In fact, the data words of the Routing Information Memory never flow through the Routing Processor. That is, the Routing Processor never captures the data words of the Routing Information Memory in local registers. Instead of this, the data port of the Routing Information Memory is a direct input to the TTNoC interface, whereas the Routing Processor only monitors the wire-through of the sequence of data words from the data port of the Routing Information Memory to the TTNoC interface. Moreover, the Routing Processor controls the `valid` and `header` signals of the outgoing lane of the TTNoC interface. The Routing Processor drives both signal to 1, when the state machine is in the state `Processing` as well as `Setup`. Note that in the `Setup` state the very first data word in the sequence of data words has already become available at the data port of the Routing Information Memory, and therefore it is issued at the TTNoC interface to avoid further delays.

The Fragment Switch, which is attached to the TTNoC interface of the TISS, interprets that data from the Routing Information Memory as routing flit that contains switching opcodes, i.e., routing information. As a result, the procedure of switching in the TTNoC, as described in section 6.2, begins due to the operation of the Routing Processor.

7.4.3 Establishing Receive Windows

For receive operations, i.e., the port to which the current burst belongs to embodies the sink of an encapsulated communication channel, the Port Manager has to wait, until the real application data arrives at the incoming lane of the TTNoC interface. Even though communication activities are synchronized to each other by means of the macro tick, the real communication among TISSs runs at the system operation frequency. So, the arrival of data at the TTNoC interface belongs to a finer scope than it could be expressed with a macro tick. Furthermore, the TISSs as well as the TTNoC entail processing delays and propagation delays. As a consequence, the application-level data of a burst does not leave the sending TISS exactly at the rising edge of the macro tick, but a deterministic number of cycles of the system operation frequency later. Same applies to receiving TISSs, whereas the application-level data arrives a deterministic number of cycles of the system operation frequency after the rising edge of the macro tick.

Due to this circumstances, for receive operations the Port Manager has to span a *receive window* in order to postpone the processing of an incoming burst until the very first data flit arrives at the TTNoC interface. The module in the TISS, which is in charge of this task, is the *Receive Window Detector*. After an incoming burst has been dispatched, the Receive Window Detector senses the incoming lane of the TTNoC interface. It indicates the arrival of the very first data flit by issuing a notification signal to all the modules in the TISS, which have been waiting for this particular event, for instance the Memory Digger.

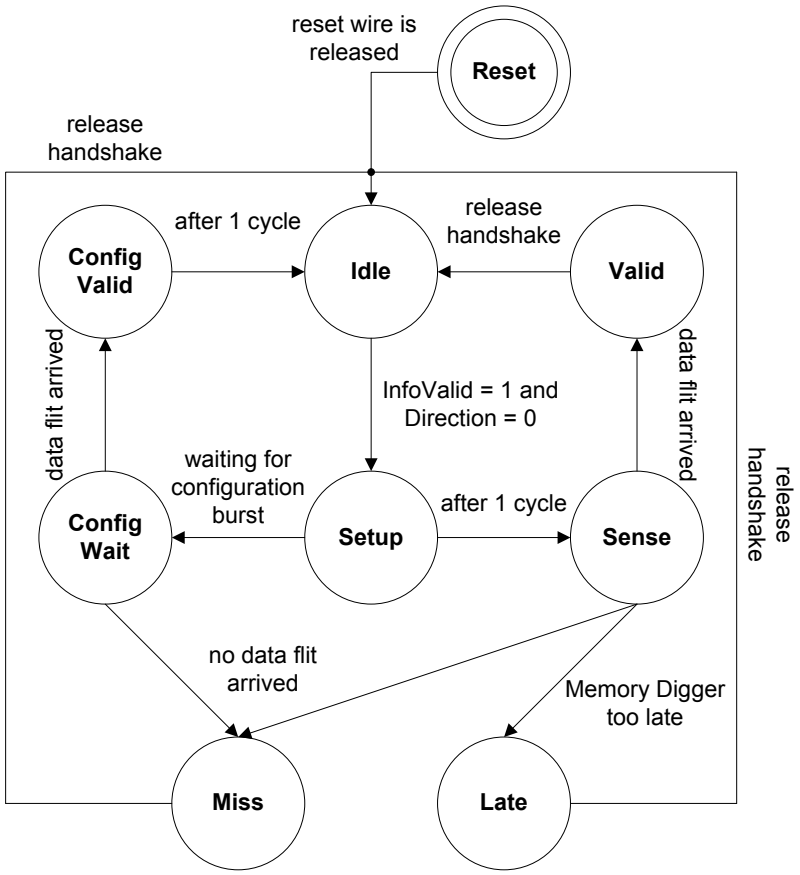


Figure 7.9: State machine of the Receive Window Detector

In the current implementation, we define a receive window as the macro tick that corresponds to the instant of the current incoming burst (i.e., the value of the `Instant` field in the associated entry of the Time-Triggered Communication Schedule) up to the consecutive macro tick. In other words, the receive window spans over an interval of one macro tick.

The State Machine of the Receive Window Detector

Obviously, the Receive Window Detector is driven by a state machine, which is depicted in Figure 7.9.

The operation of the state machine begins in the state `Idle` with the dispatching of an incoming burst, whereas `InfoValid` = 1 and the field `Direction` = 0 (input port) of the Burst Dispatcher's interface. The next state is `Setup`, where the Receive Window Detector differentiates between a "normal operation", whereas the port associated with the current burst is not the reserved port 127, and "special operation" of the on-the-fly reconfiguration (see section 8.4). In case of normal operation the following state is `Sense`, while in the other case it is `Config Wait`. Actually, the

Receive Window Detector performs the same functions in these both states. The only difference is that from `Sense` the state machine can reach two error states, i.e., `Miss` and `Late`, while `Config Wait` can only be followed by one error state, i.e., `Miss`.

In either state (`Config Wait` or `Sense`) the Receive Window Detector captures the current level of the macro tick bit in the global time base into a local single-bit register. The associated macro tick denotes the beginning of the receive window. Then, the Receive Window Detector compares the current level of the macro tick with the stored value. As long as they are the same, the receive window is still present. If they deviate and no data flit has arrived so far, the receive window has passed by, and an error condition is met.

The Receive Window Detector monitors the signals `header` and `valid` of the incoming lane of the TTNoC interface during the states `Sense` and `Config Wait`. According to the semantics of lanes in the TTNoC, data flits can be recognized by $\texttt{header} = 0 \wedge \texttt{valid} = 1$. If the levels of these signal take this combination during the state `Config Wait` respectively `Sense`, then a data flit that corresponds to the application-level data of the current incoming burst has arrived in time within the receive window. So, the following state in the state machine embodies a successful arrival of data.

The successor state (in case of a successful detection within the receive window) of `Config Wait` is `Config Valid`, and for `Sense` it is `Valid`. In this states, the notification signal named `RXValid` is driven to one. For normal operation the Receive Window Detector applies a handshake protocol with the Memory Digger like the Routing Processor. As a consequence, `Sense` is kept until the Memory Digger responds via the signal `DigBusy` (see section 7.4.5). On the contrary, the state machine remains in `Config Valid` for exactly one clock cycle of the system operation frequency, as the Memory Digger is not involved with the special operation of port 127.

Finally, the Receive Window Detector enters the `Idle` state, while the modules that have been waiting for the notification from the Receive Window Detector continue to process the current (incoming) burst.

The Receive Window Detector applies a handshaking protocol for the error states, too. For this purpose, the Receive Window Detector maintains two notification signals, which are input for the modules that are waiting for arrival of data:

- The notification signal `RXMiss` is associated with the error state `Miss`.
- The error state `Late` is indicated by the notification signal `RXLate`.

Again, if the state machine dwells in any error state, it waits for a response from the modules that have been waiting for the arrival of data in order to unlock the state machine and get back to the `Idle` state (see section 7.4.5). In fact, it waits for the notification signal `DigErr` from the Memory Digger.

The following section gives background information about the two error states of the Receive Window Detector.

About Error States

Considering the error states, the first one is `Miss`. Literally this means, that an incoming burst has missed to arrive at the receiving TISS's TTNoC interface within the given receive window. Certainly, for a time-triggered communication service, whereas all communication is determined *a priori* and synchronized towards a common global time base, this is not possible. However, the TTSoC architecture distinguishes between periodic and sporadic messages, as introduced in section 2.3.3. While a "miss" will never occur for bursts belonging to periodic messages, which are always transmitted, a sporadic message can be omitted, if the sender decides to not transmit a message. Usually, this behaviour correlates to messages based on an event semantics. In short, the `Miss` state in the Receive Window Detector has been introduced to handle such a situation caused by sporadic messages, so that the Port Manager as well as the overall TISS can cope with such a situation.

The second error state addresses the situation, when a data flit arrives at the TTNoC interface, but the modules that process the incoming data, for instance the Memory Digger, are not ready yet. This scenario can occur, when there is so much propagation delay on the TTNoC that the incoming burst arrives at the receiver at a later macro tick than the macro tick, when the burst has been sent. In such a case, the receive would have dispatched the burst at the later macro tick to be synchronous with the arrival time of the data flits at the receiver's TTNoC interface. However, a TISS takes some cycles of the system operation frequency to be ready for processing of a burst, for instance section 7.3.2 explains that the Burst Dispatcher needs 3 cycles until the instant of a burst to be processed is visible to the remaining TISS. If the incoming burst happens to arrive at such an instant, when the TISS is still busy to prepare its modules for the further processing of the current burst, then this error condition is met. The Receive Window Detector covers this scenario by introducing the state `Late`. Literally it says that the TISS has been too "late" to capture all data flits of the incoming burst.

In practice such a situation should never occur. As all communication is determined *a priori* and all propagation delays and set-up delays are also *a priori* known and deterministic, we can design the Time-Triggered Communication Schedule or the routing in the TTNoC this way that the error condition `Late` will never arise.

7.4.4 Address Calculation

Address calculation serves the purpose to determine the start and end address of the continuous data chunk in the Port Memory, which corresponds to the fragment of the message to be processed during the current burst. The module Address Calculator performs this address calculation.

For this purpose, the Address Calculator needs several input signals from the other modules of the Port Manager and TISS, which have been introduced in earlier section. We list these input signals in the following Table 7.1:

Signal Name	Origin

`Port Base Address`
`Port Type`
`Port Sync`
`Time Stamp Enable`
	Port Configuration Memory

`Direction`	Time-Triggered Communication Schedule

`Start Offset`
`End Offset`
`Size`
	Burst Configuration Memory

`Using` `TISS Addr`	Port Synchronization Memory

Table 7.1: Input signals for the address calculation

With these inputs the Address Calculator produces 3 output signals, which will be used by the Memory Digger and Time Stamper during the further processing of the current burst. The output signals are:

`StartTrans` This signal denotes the start address of the continuous data chunk in the Port Memory, which corresponds with the fragment of the current burst. This represents a real physical address in the Port Memory, which is word aligned. Therefore, this signal vector has the same width as `OCPM_MAddr` of the port interface excluding the additional bits in the vector due to word alignment. In fact, `StartTrans` has a width of 14 bit in the current implementation.

Moreover, `StartTrans` concerns, whether time stamping is activated for the current port or not. In case of time stamping, `StartTrans` is increased by an additional offset of 2, so that a time stamp (which consumes 2 data word in the Port Memory in the current implementation) has enough (memory) space to be placed in front of the application-level message in the current port.

`EndTrans` Similar to `StartTrans`, but this signal means the (real physical) end address of the continuous data chunk in the Port Memory. It also takes care of time stamping and is 14 bit wide in the current implementation.

`RealStartAddr` This signal refers to the real physical start address of the complete message in the Port Memory, but not to the start address of the continuous data chunk of the fragment of the current burst. As a time stamp is always placed in front of the application-level message in a port, `RealStartAddr` equals the physical address where the lower half of a time stamp is put, in case of an active time stamping for the current port. Consequently, the upper half of the time stamp resides at `RealStartAddr` + 1, and the beginning of the application-level

messages is located at `RealStartAddr` + 2. If no time stamping is activated for the current burst, `RealStartAddr` automatically points at the beginning of the application-level message. The signal also has a width of 14 bit in the current implementation.

Address calculation has to consider the layout of state and event ports. For instance, it has to be aware of the fact that state ports can contain shadow buffers, while event ports can contain several complete messages within the memory space of a port. As a consequence, the address calculation has to incorporate information about the current state of port synchronization. Algorithm 7.1 describes the address calculation, which is realized within the Address Calculator.

Algorithm 7.1 Algorithm of address calculation

```
1: if Time Stamp Enable = 1 ∧ Direction = 0 then
2:   ts ← 2
3: else
4:   ts ← 0
5: end if
6: if Port Type = 1 then {This is an event port.}
7:   align ← TISS Addr
8: else {This is a state port.}
9:   if Direction = 0 then {This is an incoming state port.}
10:    align ← 0
11:  else {This is an outgoing state port ...}
12:    if Port Sync = 1 then {... with implicit synchronization}
13:      align ← 0
14:    else {... with explicit synchronization}
15:      if Using = 1 then {The upper half of the port is active ...}
16:        align ← 0 {... use the lower half as shadow.}
17:      else {The lower half of the port is active ...}
18:        align ← Size{... use the upper half as shadow.}
19:      end if
20:    end if
21:  end if
22: end if
23: StartTrans ← Port Base Address + Start Offset + align + ts
24: EndTrans ← Port Base Address + End Offset + align + ts
25: RealStartAddr ← Port Base Address + align
```

The Port Manager begins to process the current burst after `InfoValid` = 1 at the Burst Dispatcher's interface. At the same time the fields of the Burst Dispatcher's interface, which drive the address ports of the relevant memories inside the TISS (i.e., Port Configuration Memory, Port Synchronization Memory, Burst Configuration Memory), are visible. As we know from section 7.4.1, it takes another clock cycle of the system operation frequency, until the current data words are available

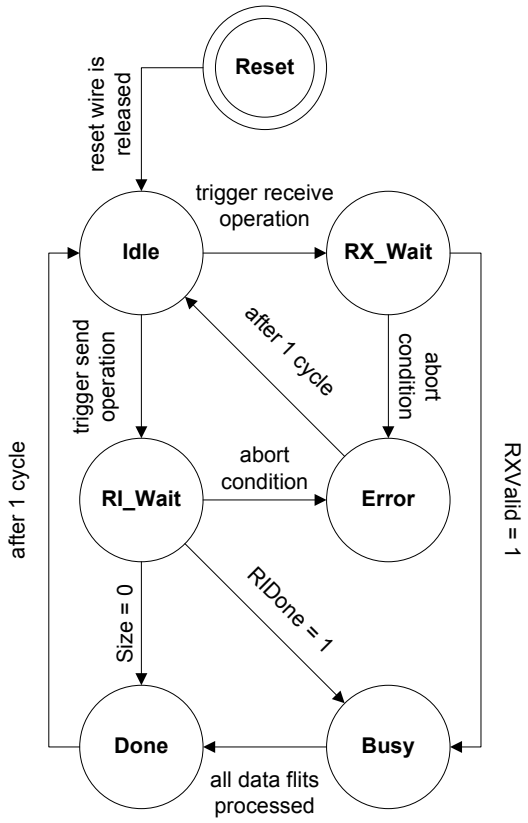


Figure 7.10: State machine of the Memory Digger

at the memories and can be fetched during the `Fetch` state of the Port Manager. Even though in the realization of the address calculation in hardware the counters, adders, and other basic register-level constructs are always active, the proper result of the output signals of the Address Calculator are available at the end of the Port Manager's state `Fetch`, i.e., at the beginning of the state `Dig`. In other words, the real address calculation takes place during the `Fetch` state, because the input values for the current burst have arrived from the memories by the `Fetch` state.

7.4.5 Digging Application Data

The central role of the Port Manager is to manage the data flow of bursts from the Port Memory to the TTNoC interface (in case of send operations) respectively from the TTNoC interface into the Port Memory (in case of receive operations). The state `Dig` in the Port Manager is dedicated to this task. However, the Port Manager delegates this task to a special module, the Memory Digger, which entails its own state machine, as depicted in Figure 7.10.

The State Machine of the Memory Digger

The state machine of the Memory Digger dwells in the `Idle` state, until the superordinate Port Manager indicates a trigger signal. This trigger corresponds to the `Fetch` state of the Port Manager's state machine. Then, the following state in the Memory Digger depends on the direction of the current burst, whether it is a send or receive operation. In the first case, the next state is `RI_Wait`, in the latter case it is `RX_Wait`.

As mentioned in earlier sections, for send operation the Port Manager has to inject routing flits to the TTNoC via the TTNoC interface in order to set up the route of the encapsulated communication channel, to which the fragment of the current burst belongs to. As we have learned in section 7.4.2, this is the responsibility of the Routing Processor. While the Routing Processor is busy, the Memory Digger has to wait for the completion of the Routing Processor. `RI_Wait` embodies this wait state. The Memory Digger leaves this state, when the notification signal `RIDone` from the Routing Processor drives a level of 1, which indicates that the Routing Processor has just completed its operation. Afterwards the Memory Digger's state machine continues in the `Busy` state.

Note that for send operations the Memory Digger might skip the `Busy` state and directly go on with `Done` state. This happens, whenever the field `Size` = 0 (from the Burst Configuration Memory), which indicates that the current burst does not transmit any data words from the Port Memory. Particularly, such a scenario occurs in case of "virtual fragments" that are used to ensure the consistent delivery order of messages, as such "virtual fragments" need not convey any data. Also note that the value of `Size` is only checked in the state `RI_Wait` (referring to send operations) in the Memory Digger's state machine. As a consequence, a burst of a "virtual fragment" must be configured as a send operation, even though the prior burst, which have really processed data, might belong to an incoming port (receive operation).

The second wait state of the Memory Digger's state machine concerns receive operations. As we know from section 7.4.3, receiving a burst requires to observe the incoming lane of the TTNoC interface, when the very first data flit arrives. As long as the Receive Window Detector monitors the incoming lane of the TTNoC interface, the Memory Digger's state machine remains in the state `RX_Wait` and waits for the notification signal `RXValid` = 1 in order to enter the state `Busy`.

During the `Busy` state the real transportation of data flits / data words from the TTNoC interface into the Port Memory (for receive operations) or from the Port Memory to the TTNoC interface (for send operations) takes place. Similar to the Routing Processor, the Memory Digger never captures data words itself, but it just monitors the flow of data words through the TISS. Actually, the port interface of the UNI and the TTNoC interface are directly connected. To be more precise, `OCPM_MData` is mapped to the data bus of the incoming lane of the TTNoC interface. In the reverse direction, `OCPM_SData` has a direct connection to the data bus of the outgoing lane of the TTNoC interface. The role of the Memory Digger in this process is twofold:

- For send operations it drives the `valid` signal of the outgoing lane of the TTNoC interface.
- For all operations it keeps track of the progress of the data flow passing through the TISS. In other words, the Memory Digger counts the data words that are transmitted respectively received.

Considering the counting of data words during the `Busy` state, the Memory Digger maintains an *address counter*. This counter contains the real physical address of the data words of the continuous data chunk in the Port Memory, which makes up the fragment of the current burst. As the Memory Digger does not count flits from the base 0, but the real physical addresses, the Memory Digger can directly drive the signal `OCPM_MAddr` of the port interface. The initial value of the address counter is given by `StartTrans`, which is a result of the address calculation and used as an input for the Memory Digger. The Memory Digger captures the value of `StartTrans` into its local address counter, as soon as the state machine leaves the `Idle` state. When the Memory Digger reaches the `Busy` state, it increments this address counter every clock cycle of the system operation frequency. As a result, the Memory Digger issues the addresses of all data words, which belong to the continuous data chunk of the fragment associated with the current burst in the Port Memory, via `OCPM_Maddr`. The Memory Digger proceeds with this operation, until *address counter* = `EndTrans`, whereas `EndTrans` is also a result of the address calculation. Then, the whole continuous data chunk that corresponds to the current burst, has been traversed by the Memory Digger. The operation of the Memory Digger can be regarded as complete for the current burst, therefore the next state is `Done`.

Besides controlling the signal `OCPM_MAddr`, the Memory Digger affects the command code according to the OCP signal specification that is assigned to the signal `OCPM_MCmd`. Table 5.2 shows, which OCP commands are supported by the current implementation. The command code at `OCPM_MCmd` and the value of `OCPM_MAddr` have to correlate. As long as the Memory Digger's state machine is not in the `Busy` state, `OCPM_MCmd` says "idle" (000). When the Memory Digger reaches the `Busy` state, the command depends on `Direction` of the current burst. In case of a receive operation, the received data has to be written into the Port Memory, thus the appropriate command for `OCPM_MCmd` is "write" (001). In case of send operation, the transmitted data has to be read from the Port Memory, therefore `OCPM_MCmd` shows a "read" command (010).

For the handshake with other modules of the Port Manager, the Memory Digger exports the state `Busy`. That is, in `Busy` state it drives the notification signal `DigBusy` to 1. The purpose of this signal is described in later section 7.4.5.

The last state of the Memory Digger's state machine (before it completes its circle of processing) is `Done`. This state causes the internal registers of the Memory Digger to be reset in order to be prepared for the next burst. It lasts for one clock cycle of the system operation frequency. Moreover, the Memory Digger notifies the

superordinate Port Manager that it has just completed its operation. Therefore, the Memory Digger drives a notification signal `DigDone` to 1 during the `Done` state. As a result, the Port Manager can continue to process its own state machine.

Aborting A Burst

The state machine of the Memory Digger also includes a state `Error`, which covers all conditions that force the Memory Digger (and further the superordinate Port Manager) to abort the processing of the current burst. The causes of this abort conditions are listed in the following:

- For receive operations, the Receive Window Detector reports that a receive window has passed by, and no data flit has arrived at the incoming lane of the TTNoC interface, or the Memory Digger has been too late to capture all incoming data flits from the beginning. According to section 7.4.3, the Receive Window Detector controls two notification signals `RXMiss` and `RXLate`, which model these error conditions.
- For receive operations of event ports, the Port Manager has discovered that the queue of that event port is full. Table 5.9 in section 5.3.2 explains, how the TISS, i.e., the Port Manager, recognizes a full queue.
- For send operations of event ports, the Port Manager has found out that the queue of that event port is empty. The TISS, i.e., the Port Manager, determines an empty queue as described in Table 5.9 in section 5.3.2.
- The host has disabled the port,to which the fragment of the current burst belongs to, via the control interface of the UNI. Then, the field `Port Enable` = 0 in its associated entry of the Port Configuration Memory.
- The communication service of the TISS has been deactivated by the host via the control interface of the UNI. In this case, the field `Communi- cation Disable` equals 1 in the Register File.

If any of these conditions evaluates to true while the Memory Digger dwells in one of its wait states (`RI_Wait` and `RX_Wait`), the Memory Digger issues the signal `AbortBurst`.

On the one hand, state machine of the Memory Digger uses `AbortBurst` = 1 to leave its wait states and enter the the `Error` state. In the error state the Memory Digger drives a signal `DigErr` to 1, which is used as the response from the Memory Digger to the Receive Window Detector in their handshake protocol. The Memory Digger leaves the `Error` state after one clock cycle of the system operation frequency and returns into the `Idle` state.

On the other hand, `AbortBurst` is directly relevant for the handshaking protocol between Memory Digger and the Routing Processor. In fact, it play a vital role to unlock the state machines other modules of the Port Manager such as the Routing Processor, which is explained in the following section.

Unlocking Other State Machines

The Memory Digger's state machine contains two wait states (`RX_Wait` and `RI_Wait`). In these states, the Memory Digger postpones its operation, until other modules of the Port Manager provide notifications about special events. In fact, the Receive Window Detector reports the arrival of data within a receive window, or the Routing Processor signals the completion of its own operation. Afterwards, each of these modules keeps on driving the notification signal. Furthermore, they remain in a specific state of their state machines, until the Memory Digger acknowledges the receipt of the notification signal. As a result, the Receive Window Detector as well as the Routing Processor employ a handshake protocol with the Memory Digger.

The Receive Window Detector controls 3 different signal towards the Memory Digger, whereas 1 indicates success (`RXValid`) and the other 2 refer to error conditions (`RXMiss` and `RXLate`). If the Receive Window Detector is tied down to the state that signals `RXValid` = 1, the signal to release it from this state is `DigBusy` from the Memory Digger. In case of the error states the Receive Window Detector's state machine is unlocked, when the Memory Digger acknowledges that error condition by issuing `DigErr`.

The Routing Processor is released from its `Done` state that drives `RIDone` = 1 by a signal `RelDone`. This handshake signal is a composition of two internal signals of the Memory Digger.

$$\texttt{RelDone} = \texttt{DigBusy} \vee \texttt{AbortBurst}$$

The Routing Processor shall release `RIDone` when either the Memory Digger has taken up operation, i.e., has reached the `Busy` state (therefore `DigBusy` = 1), or a condition to abort the burst has been satisfied, i.e., `AbortBurst` has triggered.

7.4.6 Realizing Time Stamping

The module Time Stamper of the Port Manager controls the time stamping, which is only reasonable for input ports (state ports as well as event ports). The Time Stamper follows a simple state machine, which includes an idle state, and two "stamping" states. The first stamping state produces the lower half of the time stamp, which are the lower 32 bit of the counter vector of the time format of the global time base. Logically, the second stamping state deals with the upper half (the higher 32 bit) of that counter vector.

The Port Manager triggers the operation of the Time Stamper by means of an internal trigger signal if both of the following conditions hold:

- The Memory Digger has successfully completed its operation. It indicates this event by driving `DigDone` = 1 for one clock cycle of the system operation frequency. If the Memory Digger has aborted the current burst, no time stamp is produced.

- The port, to which the fragment of the current burst belongs to, has the time stamping service enabled. That is, `Time Stamp Enable` = 1 in the associated entry of the Port Configuration Memory.

Like the Memory Digger, the Time Stamper has to write data, i.e., two data words for the time stamp, into the Port Memory. Consequently, it requires access to the port interface. During its activity, the Time Stamper manages the signals `OCPM_MAddr`, `OCPM_MCmd`, and `OCPM_MData`.

Obviously, the Time Stamper issues the values of the two data words, which correspond to the time stamp, via `OCPM_MData`. As time stamping is only supported for input ports, it just indicates a "write" command (001) at `OCPM_MCmd`. According to the memory layout of state and event ports, the time stamp has to be placed in front of the application-level message in the port. This implies that the Time Stamper must maintain the address, where to place the two data words of the time stamp in the Port Memory. For this purpose, the Time Stamper uses `RealStartAddr`, which is a result of the address calculation and points at the beginning of the current message (including time stamp plus application-level data) in the port. For the first data word (lower half of the time stamp) the Time Stamper sets `OCPM_MAddr` = `RealStartAddr` + 1, whereas the proper address of the second data word (upper half of the time stamp) is given by `OCPM_MAddr` = `RealStartAddr` + 2.

After writing these two data words to the Port Memory via the port interface, the Time Stamper reports its completion to the superordinate Port Manager by an internal notification signal . The Time Stamper returns into the idle state, while the Port Manager is unlocked from its state `Time Stamp` and continues to execute its own state machine.

7.4.7 Managing Port Synchronization

As the Port Memory is a (true) dual-ported RAM, the host as well as the TISS, i.e., Port Manager, are allowed to modify data words in this memory at any time. In order to assure the consistency of data, a synchronization protocol (according to section 5.3) for the ports must be applied. The TISS, i.e., Port Manager, and the host use the Port Synchronization Memory to exchange information about the progress of access towards a port and the consistency of data within the port.

From the point of view of the TISS, the Port Manager is responsible to implement this synchronization protocol and handle the Port Synchronization Memory. For this purpose, the Port Manager includes a component named *Port Synchronization Controller*, which is in charge of realizing the synchronization protocol from the TISS-side.

The Port Synchronization Controller is attached to the Port Synchronization Memory at its TISS-side (internal) write port. So, the Port Synchronization Controller is the only module in the TISS, which is able to manipulate entries in the Port Synchronization Memory. Contrary, the read port is mainly used to provide the

required information for the address calculation in the Address Calculator. As the host also requires access to the Port Synchronization Memory in order to keep up its part of the synchronization mechanism, the host-side of this (true) dual-ported RAM is accessible via the control interface of the UNI. Therefore, there exist a wire-through from the host-side read/write port of the Port Synchronization Memory to the control interface, i.e., the OCP Slave Wrapper that converts the internal signals to OCP-compatible naming schemes.

According to the synchronization mechanism, from the TISS-side the Port Synchronization Controller has to update the entry of the Port Synchronization Memory, which is associated with the port to which the fragment of the current burst belongs to. Depending on the direction as well as the port type (state or event port), there are particular events during the processing of the current burst, when such an update must take place. Table 7.2 lists these update events and sketches the actions on the proper entry of the Port Synchronization Memory.

Port Type	Direction	Update Event	Action
state	output	`IsFirst` $= 1 \wedge$ `DigBusy` $= 1$	toggle `Using`
	input	`IsFirst` $= 1 \wedge$ `DigBusy` $= 1$	increment `NBW`
		`IsLast` $= 1 \wedge$ `DigDone` $= 1$	
event	any	`IsLast` $= 1 \wedge$ `DigDone` $= 1$	determine `TISS Addr` and `TISS ToOF`

Table 7.2: Events and actions for the Port Synchronization Controller

The fields `IsFirst` and `IsLast` originate from the entry of the Time-Triggered Communication Schedule that corresponds to the current burst. They are visible at the Burst Dispatcher's interface and propagated into the Port Synchronization Controller. `DigBusy` and `DigDone` are the notification signals from the Memory Digger to the Port Manager, which are reused in the Port Synchronization Controller.

Actually, the Port Synchronization Controller modifies data words in the Port Synchronization Memory on two distinct events. `IsFirst` $= 1 \wedge$ `DigBusy` $= 1$ denotes the start of processing of the current burst, which covers the very first fragment of the current message. By contrast, `IsLast` $= 1 \wedge$ `DigDone` $= 1$ refers the the end of processing (after the last data word) of the very last fragment of the current message.

For output state ports, the Port Synchronization Controller simply toggles the field `Using` of the entry corresponding with the current port, and then writes this new value into the Port Synchronization Memory. According to section 5.3.1, input state ports maintain an NBW sequencer. This sequencer is incremented at the beginning of the processing of the burst of the very first fragment as well as the end of processing of the burst of the very last fragment of the message.

For event ports, the direction of the port is not relevant, the Port Synchronization Controller always calculates new values for `TISS Addr` and `TISS ToOF` at the end of the burst of the last fragment of the messages, which equals the completion of the

whole message. The calculation of the new values for these two fields conforms to their definition in section 5.3.2.

7.5 Latencies of Operations

In this section we deal with the timing of receive and send operations within the TISS. We investigate, how long it takes to set-up the modules in the TISS involved in the data transfer. Moreover, we give the latency of the availability of the first data flit at the Port Interface (for receive operations) and TTNoC interface (for send operations) after the arrival of the instant associated with the current burst. In this context, we introduce the variables Δ^+ and Δ^- for receive respectively send operations. Additionally, we construct the theoretical minimum latency for Δ^+ and Δ^-.

For receive operations, the minimum latency Δ^+_{min} denotes the number of clock cycles of the system operation frequency that it takes to dispatch a burst, to set-up all modules of the TISS involved in the receive operation, and to issue the very first received data flit at the Port Interface.

Contrary for send operations, the minimum latency Δ^-_{min} is the number of clock cycles of the system operation frequency, until the first data flit is available at the outgoing lane of the TTNoC interface after the dispatching of the burst.

Besides this, we have to consider the time, until the Burst Dispatcher has issued the relevant information about the current burst at its interface to the remaining TISS (i.e., Port Manager) after the arrival of the instant of the current burst. As mentioned in section 7.3.2, Δ_{disp} models this latency. Therefore, the progress by the Port Manager during receive as well as send operations must be added to Δ_{disp}.

7.5.1 Receive Operations

For receive operations, the minimum latency Δ^+_{min} means the best case, whereas the Receive Window Detector recognizes the data flit of the incoming fragment during its very first clock cycle of wait state (i.e., `Sense` state).

Figure 7.11 shows that the minimum latency is $\Delta^+_{min} = 7$ clock cycles of the system operation frequency, starting at $t = \Delta_{disp} = 3$. Actually, it takes 2 additional clock cycles until the Port Manager is prepared to receive data flits from the incoming lane of the TTNoC interface. In fact, at this time the Receive Window Detector has reached the state `Sense`, so that incoming data can be recognized. Figure 7.11 shows such a best case, when the very first data flit arrives exactly at the same clock cycle, after the Receive Window Detector has become ready. We also learn from Figure 7.11 that it takes additional 2 clock cycles for a given data flit to pass through the TISS and appear at the Port Interface. The sum of all this single latencies makes up the minimum latency for receive operations.

$$\Delta^+_{min} = \Delta_{disp} + 2 + 2 = 7 \tag{7.1}$$

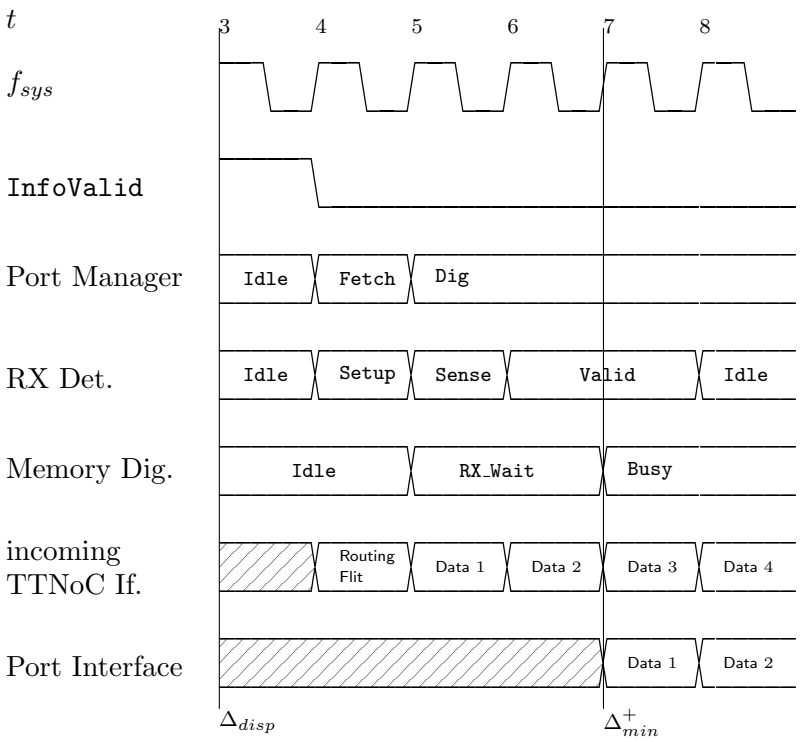


Figure 7.11: Minimum latency of receive operations

In the general case, such a best case can not be achieved. The assumption that the first data flit arrives during the very first clock cycle of the `Sense` state is too optimistic. Δ^{+} refers to this general case of receive operations. In this case, the Receive Window Detector dwells in its wait state `Sense`, and therefore delays the receive operation. We define the number of clock cycles caused by the wait state as $\Delta_{\texttt{Sense}}$.

Nevertheless, we can derive two fixed latencies from Figure 7.11. In fact, it always takes 2 additional clock cycles of the system operation frequency after `InfoValid` $= 1$ at $t = \Delta_{disp} = 3$, until the Receive Window Detector is ready to monitor the incoming lane of the TTNoC interface. If data has eventually arrived, it takes another 2 clock cycles to forward the data flits to the Port Interface. So, we can define Δ^{+} as follows:

$$\Delta^{+} = \Delta_{disp} + 2 + \Delta_{\texttt{Sense}} + 2 \tag{7.2}$$

If we generalize Δ^{+}_{min} by $\Delta_{\texttt{Sense}} = 0$, equation 7.2 holds for the minimum latency with its best case scenario as well as for the general case.

Note that $\Delta_{\texttt{Sense}}$ refers to a wait state. Apparently, this imposes a sense of local uncertainty on the calculation of Δ^{+}. From the point of view of the receiver, we can not reason about $\Delta_{\texttt{Sense}}$, as we can not locally decide when data flits arrive. However, the latency of receive operations is deterministic, indeed.

If we consider communication from the global point of view, we are aware of the fact that a receive operation is always combined with a send operation. If we

track the route of data through the TTNoC, whereas each Fragment Switch causes a deterministic propagation delay, and incorporate the (deterministic) latency of the sender, we can exactly calculate the clock cycle of the system operation frequency, when the data arrives at the receiver's TTNoC interface. As all necessary information is known *a priori*, we can predict $\Delta_{\texttt{Sense}}$ for each receive operation.

7.5.2 Send Operations

For send operations, we can distinguish between two cases. According to the routing modes introduced in section 6.2.2, we can have a send operation without preceding routing information (circuit-switched mode), and the more frequently used header-payload mode that involves routing flits sent before the data flits of the current fragment.

In the first case, the Routing Processor does not postpones the operation of the Memory Digger, as the Routing Processor has no operation to perform. This embodies the best case for send operations, which is modelled by Δ^{-}_{min}.

Figure 7.12a illustrates that $\Delta^{-}_{min} = 7$ clock cycles of the system operation frequency elapse after the arrival of the instant associated with the current burst, until the very first data flit is available at the outgoing lane of the TTNoC interface. It takes 3 clock cycles for the Memory Digger to set the proper address of the first data flit at the Port Interface. Moreover, the Port Memory needs 1 clock cycle to respond. Keeping Δ_{disp} in mind, we can explain Δ^{-}_{min} by the following equation:

$$\Delta^{-}_{min} = \Delta_{disp} + 3 + 1 = 7 \tag{7.3}$$

In the latter case, the injection of routing flits by the Routing Processor causes a delay, which equals the number of routing flits r. Hence, we can say that the latency of send operations Δ^{-} is a function in r: $\Delta^{-}(r)$. Based on the best case expressed by Δ^{-}_{min}, we can derive the latency for send operations as follows:

$$\Delta^{-}(r) = \Delta_{disp} + r + 3 + 1 \tag{7.4}$$

As we have one routing flit (hence, $r = 1$) in Figure 7.12b, the latency is given by $\Delta^{-}(r) = 7 + r = 7 + 1 = 8$ clock cycles of the system operation frequency. If we generalize the best case case of send operations with $r = 0$, the equation 7.4 holds for the best case as well as the general case of send operations.

Although in the latency of send operation in the general case is dependent on the number of routing flits, we can still draw deterministic conclusions about $\Delta^{-}(r)$. The reason for this is that the number of routing flits r is known *a priori* for each send operation. Unlike receive operations, we can predict the latency of a send operation from information that solely resides within the sender's TISS.

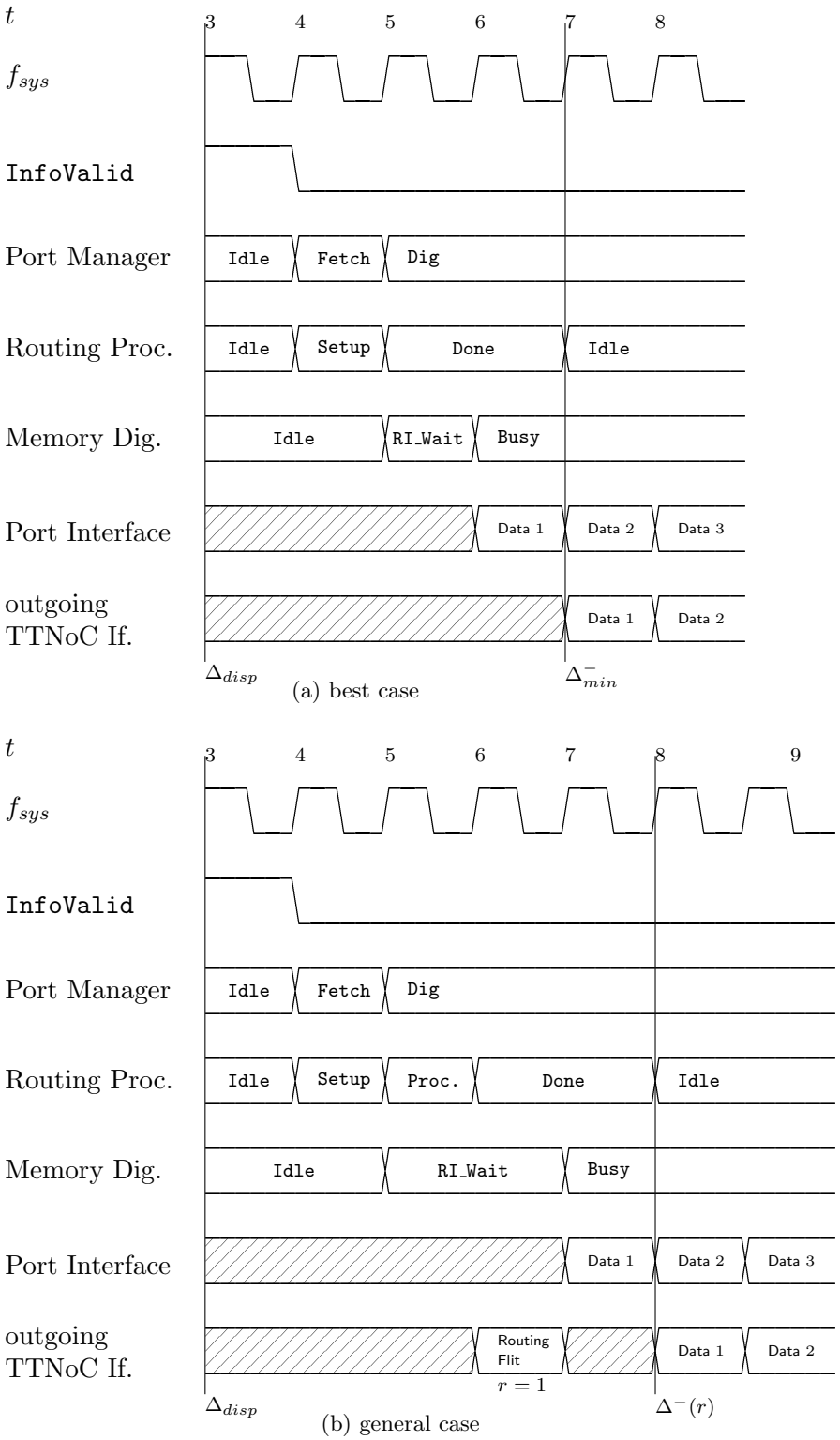


Figure 7.12: Latency of send operations

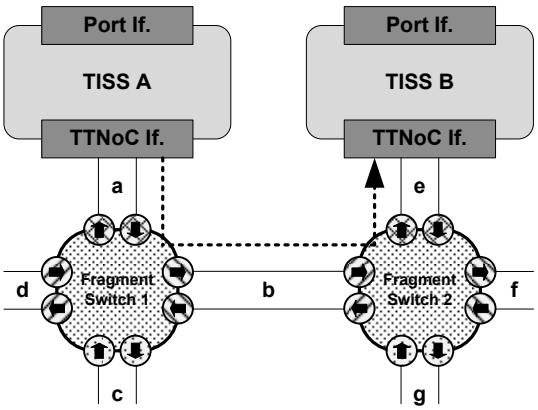


Figure 7.13: Example topology

7.5.3 Example of Transmission

Figure 7.14 illustrates the data flow from a sending TISS `A` to a receiving TISS `B` and the latencies of transmission. The topology, on which this example is based, is shown in Figure 7.13.

The corresponding encapsulated communication channel has its source at TISS `A` and is then routed over Fragment Switches 1 and 2 to the TTNoC interface of TISS `B`. The interconnects, which are traversed on this route, are $\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{e}$. The burst consists of 4 flits. Moreover, there are $r = 2$ routing flits involved. Note that each routing flit contains exactly one switching opcode. Consequently, when that switching opcode is consumed at a Fragment Switch, the routing flit becomes empty and is therefore discarded by the Fragment Switch. This explains the absence of routing flit 1 on interconnect `e` in Figure 7.14, respectively the absence of both routing flits on the TTNoC interface of TISS `B`.

We see in Figure 7.14 that $\Delta^{-}(r) = 9$, which conforms to the definition in equation 7.4. Contrary, $\Delta^{+} = 14$ so that it takes additional 5 clock cycles of the system operation frequency compared to Δ^{+}_{min}. The receiving TISS `B` has to wait for the arrival of the very first data flit due to the propagation delay on the TTNoC, which is modelled by $\Delta_{\texttt{Sense}} = 5$ according to equation 7.2. That 5 clock cycles result from the sum of the number of routing flits $r = 2$, which postpone the transmission of real data flits, and the number of interconnects $\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{e} \Rightarrow 3$ each flit has to go through.

7.6 Initialization of the TISS

Usually, the start-up of a system is a problematic matter. While its quite convenient to have a system running in a stable and defined state, its much more sophisticated to get the system up from an inactive state, for instance from the power-up. The TTSoC architecture is no exception of this fact.

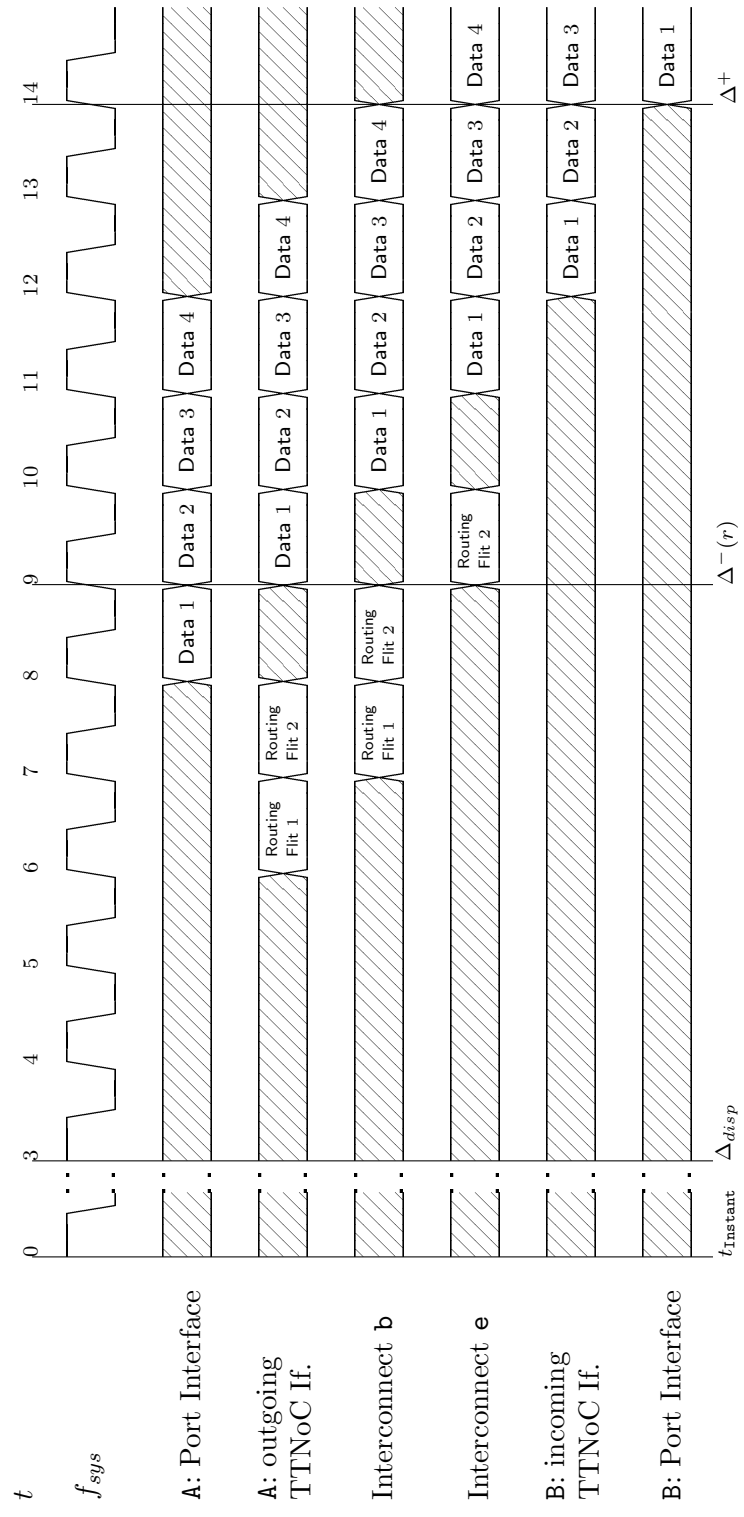


Figure 7.14: Example of latency at sending and receiving TISSs

In addition to a start-up, the TTSoC architecture entails an additional degree of complexity. Actually, the integrated resource management allows a reassignment of resources and communication activities during run-time, which is the on-the-fly reconfiguration (see section 8.4). So, this requires the TTSoC architecture to cope with a scenario, when it must not only handle the start-up, but switch from one running state into another one immediately after the on-the-fly reconfiguration – this event is associated with the *reconfiguration instant*. Fortunately (due to the current implementation of the TTSoC architecture), the start-up and the activities caused by the reconfiguration instant can be reduced to the same problem. We call the procedures necessary during start-up as well as after the reconfiguration instant the *initialization*.

7.6.1 What must be initialized?

Not all entities in the TTSoC architecture require an initialization. For instance, the TTNoC does not decide, when a burst must be processed, nor does it determine the route of an encapsulated communication channel. Hence, it strictly obeys the directives of the TISSs, which handle the dispatching of bursts and contain the routing information for encapsulated communication channels. Consequently, there are no conditions or states that must be restored after start-up or reconfiguration in the TTNoC. Therefore, the TTNoC does not require an initialization.

The TISS demands for a partial initialization, that is, some components in the TISS need to take care of initialization and others do not. The Port Manager and (all of its components) is driven by a state machine, which begins a cyclic execution on behalf of the dispatching of a burst. That cyclic execution of the state machine is a closed loop, which always starts at the beginning, i.e., the `Idle` state. Consequently, its simple to establish an operational condition in the Port Manager, as the triggering event comes from the "outside", i.e., the dispatching of a burst that is covered by the Burst Dispatcher.

The Burst Dispatcher includes period controllers and phase comparators. As we have seen in section 7.3, the phase comparator in turn is controlled by a paired period controller. Each period controller traverses an associated cyclic, linked list in the Time-Triggered Communication Schedule. Eventually, we have pinpointed the entity, which requires initialization. It are the period controllers in the Burst Dispatcher, which have to catch up with the traversal of the cyclic, linked lists in the Time-Triggered Communication Schedule.

7.6.2 Initialization of Period Controllers

During initialization each period controller has to load the proper entry from the Time-Triggered Communication Schedule, where the traversal of the cyclic, linked list should begin. The initialization vector, which has been introduced in section 7.2.1, contains a copy of the first entry (in the cyclic, linked list) to be processed for each

period. So, during the initialization each period controller has to grab the entry in the initialization vector, which is dedicated to the same period of the periodic control system.

As there is only one read port at the memory of the Time-Triggered Communication Schedule (and as many period controllers as the number of supported periods in the periodic control system), a collision at the Time-Triggered Communication Schedule is obvious, if each period controller tries to grab its entry from the initialization vector immediately after the launch of the initialization. As a consequence, the period controllers have to agree on an access pattern to avoid driver conflicts. For this purpose, each period controller i is aware of the period number i it is assigned to. After the initialization is launched, each period controller waits for i cycles of the system operation frequency until it issues the address of the corresponding entry in the initialization vector via its address bus. As there is an exact mapping of period number between entry in the initialization vector and period controller, for instance period controller 4 waits 4 cycles after the beginning of the initialization procedure, and then assigns 4 to its address bus. According to the timing in the Burst Dispatcher, the period controller fetches the entry from the data bus, extracts the `Instant` field, copies the payload information, and sets up its paired phase comparator like in the ordinary operational case, which is described in section 7.3.1.

As each period controller has to fetch one entry from the Time-Triggered Communication Schedule, the initialization takes as many clock cycles as the number of supported periods plus a constant delay of 4 clock cycles of the system operation frequency. In the current implementation, (for 16 and 32 periods) the whole initialization takes 20 respectively 36 clock cycles of the system operation frequency, then.

During this time, the Burst Dispatcher masks the signal vector `PeriodEna`, so that no phase comparator is able to accidentally trigger the dispatching of a burst. Thus, during the initialization no communication activity can take place in a given TISS. Moreover, as the start-up as well as the reconfiguration instant occurs among all TISSs simultaneously, the communication activities of each TISS are postponed. As a result, it can not happen that some TISSs continue their operation while others do not. Hence, the state of the communication service in the overall system is consistent all the time.

7.6.3 Scenarios of Start-up

The whole procedure of initialization relies on the condition that the initialization vector is present in the Time-Triggered Communication Schedule. In the current implementation, which is based on FPGA target technology, the RAM blocks in the FPGA can be equipped with values after start-up, i.e., power-up. As a result, the initialization vector in the Time-Triggered Communication Schedule can be preloaded at start-up. This feature is immanent in this target technology, as we have to load a design that embodies a circuit at power-up. So, FPGA technology leverages this mechanism to set initial values into RAM blocks, while it configures the design.

However, we pursue the vision that the TTSoC architecture will run on ASIC technology in the near future. Although ASIC technology supports RAM blocks on its silicon die, it is more sophisticated to achieve a pre-load of these RAM. For instance, it would require a dedicated boot loader to extract the required RAM data from a permanent memory, e.g., a flash memory device, and write the initial values into the RAM. Nevertheless, such a mechanism causes an overhead to realize this boot loading mechanism for ASIC technology, which manifests in additional die area plus the permanent memory device and therefore causes additional costs and energy consumption.

Actually, there is no need for a dedicated boot loading mechanism in the TTSoC architecture even for ASIC technology. We can yield the concepts of on-the-fly reconfiguration (see section 8.4) to fill the RAM memory of the Time-Triggered Communication Schedule with pre-defined values, as it would be done by an ordinary boot loading mechanism for ASIC technology. As a result, there is no need for additional circuitry, and no further costs and energy consumption arise.

For this purpose, the TISS supports two distinct modes of start-up.

boot strapping The memory of the Time-Triggered Communication Schedule as well as other memories such as the Burst Configuration Memory and Routing Information Memory are equipped with pre-loaded values. In particular, there already exist application sections and a suitable initialization vector in the Time-Triggered Communication Schedule, and the other memories are also initialized with appropriate values. The configuration of this initial setting is called the *boot strapping application* – thus, "boot strapping mode".

Usually, the boot strapping application provides a minimalistic set of encapsulated communication channels. These channels enable a subsequent on-the-fly reconfiguration, which installs the configuration of an intended, full-scale application. Note that this requires a specific period, which must be enabled at start-up time, i.e., the corresponding signal of `PeriodEna` must be set, so that the messages of these encapsulated communication channels can be transported from the very beginning.

The boot strapping mode is predestined for FPGA technology.

start-up configuration In the mode of start-up configuration all memories in the TISS are blank. Consequently, the TISS must be supplied with proper configurations for its memories. For this purpose, all components involved in the on-the-fly reconfiguration are triggered and await the configuration data for the memories. Eventually, if the data of an intended, full-scale application arrives, it is written in the memories and an initialization is triggered. The procedure is the same as the on-the-fly reconfiguration of the integrated resource management, which is described in section 8.4.

The start-up configuration mode is the best choice for ASIC technology. Certainly, it can also be applied for FPGA technology.

Both modes of start-up are exclusive. We can have the TISS support either the boot strapping mode or the start-up configuration mode. The reason for this is that each mode must be "hard-wired" in the TISS, that is, written in the VHDL source code. For instance, for boot strapping mode a period must be enabled by setting a signal of `PeriodEna`. Another example, for the start-up configuration mode all components involved in on-the-fly reconfiguration must be active at start-up, which requires adaptions of their state machines.

To summarize, there is no chance to switch between start-up modes. If once branded into the design of the TISS, the support for a start-up mode is permanent and can not be revoked.

Chapter 8

Integrated Resource Management

Integrated resource management is the ability of a system to dynamically (i.e., at run-time) modify the allocation of the system's resources to its hosted application subsystems[1]. The purpose is to react to changing resource demand or resource availability, which fluctuates during run-time. Integrated resource management yields better utilization of the available resources, improved dependability, and the enabling of power-aware system behaviour.

In this chapter we sketch how the TTSoC architecture incorporates the generic concept of integrated resource management, whereas a full specification of integrated resource management in the TTSoC architecture can be found in [Hub08]. Section 8.4 comprehensively elaborates the details of the process of on-the-fly reconfiguration.

8.1 Scope of Resource Management

Generally, integrated resource management is made up of three orthogonal spheres of influence on the operation of any SoC architecture:

- allocation of computational resources, i.e., (in the context of the TTSoC architecture) micro components
- assignment of communication resources
- assignment of power to particular micro components

Computational Resources In any SoC architecture, management of computational resources addresses the assignment of sets of processing components to particular application subsystems. Typically, e.g., for safety-critical applications, these assignments are static and performed before the deployment of the system. In contrast

[1] In the context of the TTSoC architecture also called Distributed Application Subsystem (DAS).

to this inflexibility, integrated resource management allows to perform the allocation of computational resources. For instance, this could be applied during start-up in order to achieve an initial configuration of the SoC, or during run-time initiated by changing requirements of given application subsystems. Furthermore, if an SoC contains multiple components, which are realized by general purpose CPUs, components could load application software at run-time and execute the code. In case of components of identical hardware the physical mapping of application subsystems to processing components can be reconfigured, which is useful if components are stuck at a permanent fault.

Relaxing the requirement of 100 % correctness for devices and interconnects may dramatically reduce the costs of manufacturing, verification and testing of SoCs, as the integrated resource management entails some sort of robustness. Moreover, it contributes to an efficent usage of the available physical resources in the SoC.

The TTSoC architecture is designed to support the adaption of computational resources. Each micro component contains different application software modules. For each configuration selected by the mechanisms of integrated resource management, the micro component redirects the execution to the corresponding software module so that all micro components fulfil the services specified for a given configuration.

Communication Resources In the TTSoC architecture we consider communication as a shared resource that is distributed among the participants of communication, hence the micro components. On the one hand, the strong encapsulation mechanism of encapsulated communication channels let us draw this conclusion. On the other hand, it is the modularization of the entire TTSoC into nearly independent micro components that let us regard communication as a shared resource.

Since the access to the TTNoC is mediated by the use of an *a priori* TDMA slot scheme, the coordination of this shared resource is performed via the construction of the Time-Triggered Communication Schedule that is contained within the TISSs. Integrated management of this resource means to dynamically adapt the allocation of (TDMA) send slots to micro components in order to fulfil the temporal requirements (e.g., bandwidth or latency) of the pulsed data streams utilized in encapsulated communication channels, which traverse a particular micro component.

Power We also regard power as a shared resource from two points of view:

- the maximum power dissipation that is permitted to occur without physically harming the TTSoC
- the available energy that is provided by the power supply, for instance the capacity of a battery of mobile devices

For both, integrated resource management has to control the dissipated power for each micro component. In addition to the control within micro components,

the TISSs of the individual micro components can be put into fine-grained, power-efficient degraded modes of operation by selectively deactivating the dispatching logic (i.e., phase comparators and period controllers in the Burst Dispatcher) of a subset of the supported periods of the periodic control system.

Particularly in the TTSoC architecture, integrated resource management supports "power-aware routing" of encapsulated communication channels through the TTNoC. It is possible to circumvent specific switching components, i.e., Fragment Switches, so that traffic is concentrated at a subset of switching components in the TTNoC. Consequently, idle switching components remain in a power-saving idle state, while we increase the energy efficiency of switching components with increased traffic. Additionally, each Fragment Switch can be totally powered down, i.e., put into a *sleep state*, to further increase power saving if no traffic passes by that given Fragment Switch.

In addition to "power-aware routing", the activation and deactivation of pairs of period controllers and phase comparators in the Burst Dispatcher of a TISS entails another potential to cut down on power consumption.

8.2 Resource Management Strategies

An application subsystem provides a well specified service, which might change over time. Consider, for example, the multimedia subsystem of a car. The user is allowed to activate different audio or video streams, e.g., incoming video phone call, navigation, jam information. Besides these major service classes, which are provided by the multimedia subsystem, the QoS parameters can vary due to the available resources, e.g. depletion of battery. We call these major service classes the *primary modes* of an application subsystem. Each primary mode possesses a set of degradation levels with respect to QoS parameters.

On the one hand, the change between primary modes is initiated by the application subsystem (i.e., the micro component that realizes the application subsystem) itself and not by the integrated resource management of the TTSoC architecture. A switch between primary modes represents a substantial change in the service of an application subsystem. It may change the communication pattern between cooperating micro components or (in-)activates a subset of the micro components of an application subsystem.

On the other hand, different degradation levels represent the same primary mode but with a different QoS level. Continuing the example from above, degradation levels of the primary mode "video phone call" can be considered as different frame sizes and rates of the audio/video stream in the multimedia system. Also, the inactivation of the video stream can be considered as a degraded mode. Each degradation level is characterized by resource demands (i.e., computation, communication, and power) of the individual micro components realizing the application subsystem.

Primary modes and their degradation levels manifest in different configurations for the memories of TISSs. For instance, communication activities of a primary

mode (or one of its degradation levels) are described within an application section of the Time-Triggered Communication Schedule. Also, a primary mode might use other routes for its encapsulated communication channels, thus this primary mode requires an update of the routing information in the Routing Information Memory.

The process, how the configuration of the TISSs' memories is distributed for the purpose of integrated resource management, is the *on-the-fly reconfiguration*. The calculation of the configurations, i.e., to derive values for data words in the memories from the semantics of integrated resource management, can be carried out in two different strategies.

8.2.1 Static Resource Management

The subject of the first resource management strategy is to maintain a set of degradation levels of all application subsystems in a specific primary mode, for which the sum of the demanded resources is available. Since the specification of the primary modes and their degradation levels is done at design time of the application subsystems, it can be verified off-line that all possibly simultaneously occurring primary modes do not exceed the overall available resources.

We call such a preparation of primary modes and degradation levels at design time a *static resource management*. The current implementation of the TTSoC architecture supports this mode of integrated resource management, yet.

8.2.2 Dynamic Resource Management

The more advanced resource management strategy is *dynamic resource management*. Here, primary modes and their degradation levels dictate upper and lower bounds of a given resource, which is allowed to be assigned to a given application subsystem.

A *resource request* by a micro component states the amount of a given resource the requesting micro component itself intends to reserve or release. For instance, a micro component request to increase the bandwidth for an encapsulated communication channel, or it wants to establish a new encapsulated communication channel with some other micro component. Another example, the micro component is currently idle and does not require that much power, therefore it requests that the integrated resource management withdraws power.

As a result, micro components execute a *negotation protocol* with the integrated resource management in order to communicate resource requests from the local point of view. It is the responsibility of the integrated resource management to assure that the resource request does not exceed any bound of the current primary mode respectively degradation level.

Besides resource requests, micro components explicitly select the current primary mode and degradation level for their application subsystems.

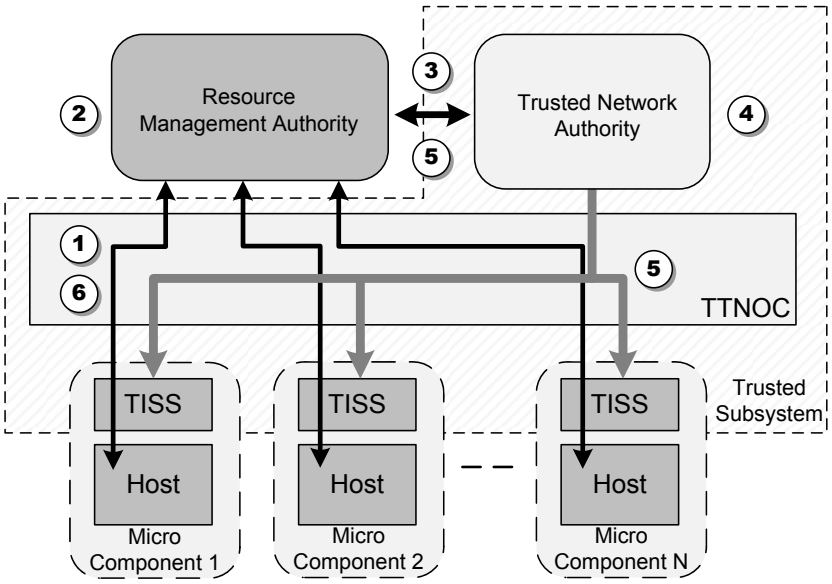


Figure 8.1: Schematic representation of integrated resource management

8.3 Sequence of Interaction

Figure 8.1 illustrates the communication activities, which belong to the process of integrated resource management, between entities of the TTSoC architecture involved in the integrated resource management. This communication is divided into 6 phases:

1. In the TTSoC architecture, integrated resource management starts with a *reconfiguration request* from any micro component. This reconfiguration request could have been triggered, for instance, by a user input that leads to a switch of the primary mode of the application subsystem the micro component belongs to.

2. The RMA processes the reconfiguration request and performs a reallocation of the available resources, e.g., generate a new Time-Triggered Communication Schedule. According to the resource management strategy, a reconfiguration request indicates either directly the aimed primary mode to which the application subsystem should switch (static resource management), or it contains a resource request to modify the amount of some allocated resource, e.g., an increase in bandwidth for a particular encapsulated communication channel (dynamic resource management). In the latter case, the RMA has to evaluate, whether that resource request does not exceed the available amount of the affected resource in the particular primary mode of the application subsystem.

3. At the current stage of development, primary modes and degradation levels of all application subsystems are determined off-line. Thus, the RMA should always be able to find a feasible configuration for the memories in the TISSs.

This generated proposal for configurations is disseminated by the RMA to the TNA in order to perform correctness checks.

4. In this phase of the integrated resource management the TNA acts as a "guard" for the reconfiguration activity of the RMA. Primary, the TNA checks the proposed Time-Triggered Communication Schedule, routing information for encapsulated communication channels etc. for all TISSs . The TNA assures that no conflicts and congestion of encapsulated communication channels occur in the proposed configurations from the RMA. Besides this, additional constraints of the proposed configurations, e.g., resource allocations with respect to bandwidth for individual application subsystems, phase alignment of individual pulsed data streams and so forth, are evaluated. The TNA does not distinguish between resource management strategies, but it always performs the same generic validation algorithm on the proposed configurations.

5. Based on the results of the prior examination, the TNA carries out two actions. Firstly, it informs the RMA, whether the proposed configurations have been accepted. Secondly, if the TNA considers the proposed configurations as correct, the TISSs of all micro components (involved in the affected application subsystems) are updated. Otherwise, those TISSs remain unchanged.

6. In case of accepted configurations, in the last phase the TNA disseminates the configuration for the memories in the TISSs of affected micro components. This last phase corresponds to the process of on-the-fly reconfiguration.

Besides reconfiguration requests from micro components, the RMA is enabled to initiate a reconfiguration on its own. For instance, the RMA reacts on a thermal sensor, which indicates that the overall power dissipation of the TTSoC is too high. However, we impose the restriction that the RMA can only affect the degradation levels of any application subsystem, but not its primary modes.

8.4 On-the-fly Reconfiguration

The integrated resource management relies on the communication service of the TTSoC architecture to establish the communication required during the 6 phases of interaction. The messages exchanged via the associated encapsulated communication channels include the necessary information, e.g., reconfiguration request, notifications etc. As a result, integrated resource management operates with messages[2] and can therefore be counted to the application-layer in the TTSoC architecture. In other words, the TISSs and the TTNoC (i.e., the TSS) are not aware of the semantics of the integrated resource management.

However, there is a special treatment necessary in a TISS in order to redirect configuration data, which has been sent by the TNA to be set into the Time-Triggered

[2]Note that the layout of these messages is beyond the scope of this thesis.

Communication Schedule, Burst Configuration Memory, etc. of the corresponding physical memories. This special treatment embodies the on-the-fly reconfiguration that takes place during phase 6.

The semantics of on-the-fly reconfiguration in the TISS must be hard-wired. That is, the TISS contains special control logic to achieve on-the-fly reconfiguration. Due to the existence of special purpose circuitry in the TISS the information exchanged between TNA and TISS during the on-the-fly reconfiguration must obey a specific *protocol*. The component in the TISS, which redirects configuration data into the physical memories and realizes this specific *protocol of on-the-fly reconfiguration*, is the *Configurator*, as illustrated in Figure 7.1.

This section explains the protocol of the on-the-fly reconfiguration and informs about its semantics.

8.4.1 Configuration Bursts

The on-the-fly reconfiguration takes usage of the communication service of the TTSoC architecture. There is one encapsulated communication channel established between the TNA and each micro component. Through each encapsulated communication channel the TNA distributes configuration data for the physical memories in the receiving TISS. Hence, the TNA is the source of each encapsulated communication channel and each TISS is the sink of the given encapsulated communication channel.

As usual, the encapsulated communication channels convey messages, which are made up of fragments, which in turn are processed by bursts. The same applies to all encapsulated communication channels that are associated with the on-the-fly reconfiguration. The configuration data for the physical memories of the TISS are transported within bursts – the so-called *configuration bursts*. As communication activity within an encapsulated communication channel is aligned to the periodic control system, these bursts are also periodic and repeat according to a given period of the periodic control system.

Moreover, the configuration bursts also manifest in the Time-Triggered Communication Schedule. From the point of view of the TNA, they are ordinary bursts that correspond to a fragment of a message in an output port. From the point of view of the TISS, the bursts are associated with the special port 127 in the current implementation. As the configuration bursts are linked to this special purpose port, the dispatching of configuration burst triggers the special treatment realized by the Configurator. Figure 8.2 gives an example of a sequence of configuration bursts arriving at different micro components.

Figure 8.2 illustrates, how the configuration data that is to be moved into the physical memories of the TISS of a given micro component is spread across different configuration bursts. As configuration bursts are dispatched like ordinary bursts of application messages, the feature of *pulse interleaving* (see section 4.5) is also available. For instance, micro component A receives two distinct configuration bursts,

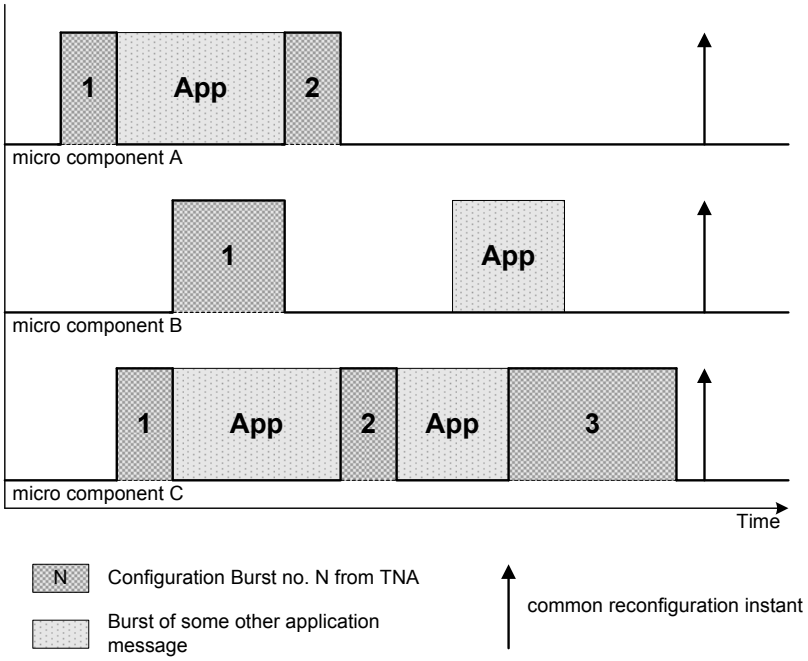


Figure 8.2: Incoming configuration bursts at micro components

because the first one has been aborted and interleaved with a burst of some application message of another encapsulated communication channel. On the contrary, the configuration data of micro component B can be transported within a single configuration bursts, as the burst of an application message does not interfere with that configuration burst.

Like bursts of application messages, configuration bursts are operated within one period. That is, the configuration bursts that refer to a given on-the-fly reconfiguration never span over more than one period. This is a necessary condition in order to establish the *reconfiguration instant*. The reconfiguration instant marks the instant across each TISS, when the actual reconfiguration is launched. According to section 7.6, reconfiguration causes an initialization of the TISS. The reconfiguration instant is realized as a burst that does not transmit data. In fact, the reconfiguration instants manifests as a "virtual fragment" for the special purpose port 127 in the Time-Triggered Communication Schedule, which has been introduced in the context of consistent delivery order in section 4.7.2. Like the watchdog and dissemination service, the reconfiguration instant leverages the concept of "virtual fragments". Therefore, the associated burst is dispatched like ordinary bursts, and therefore is synchronized to the global time base. As a result, it is possible to assign a common instant for the reconfiguration instant at each TISS, so the initialization of the TISSs is triggered simultaneously among the whole system.

Note that configuration bursts are *sporadic*, i.e., messages with event semantics. The reason is that the system might remain in a given primary mode for some time. In such a case no micro component issues an reconfiguration request, therefore the se-

quence of interactions of integrated resource management is not executed. Hence, no on-the-fly reconfiguration takes place, and the TNA does not transmit configuration bursts.

8.4.2 Protocol of Reconfiguration

The configuration bursts convey configuration data from the TNA to the physical memories of a TISS. The Configurator of each TISS grabs the incoming data flits of a configuration burst and moves the configuration data into a given physical memory. The information, which configuration data has to be put to which physical memory at which address, must be included in the configuration burst itself. In fact, configuration bursts contain two entities:

1. Some *meta-information* that tells the Configurator how to process the incoming configuration data.
2. The payload data, which contains the real configuration data that is to be moved (according to the meta-information) into the physical memory of the Time-Triggered Communication Schedule, Burst Configuration Memory, Routing Information Memory, etc.

The combination of meta-information and payload data results in the *protocol* of the on-the-fly reconfiguration. This protocol is applied between the TNA and the Configurator of each TISS. All configuration bursts are structured according to this protocol.

Figure 8.3 shows, how this protocol determines a structure in a configuration burst. Thereafter, a configuration burst is organized into several blocks. Each block begins with a *control flit*, which contains meta-information. Then, the control flit is followed by a sequence of data flits carrying payload data. For a given block, the control flit instructs the Configurator, where to place the configuration data contained in the payload. After the last data flit of the payload of the last block, the configuration burst ends with a *terminal flit*. When the Configurator indentifies a terminal flit, it is aware of the end of the configuration burst.

The protocol of reconfiguration defines that a configuration burst must contain at least one block with an introducing control flit followed by payload data, which in turn must include at least one data flit of configuration data, and must end with exactly one terminal flit.

In the following we examine the layout of control and terminal flits.

Layout of the Control Flit

A control flit is embedded into a data flit, which is transported over the data bus of lanes in the TTNoC. Figure 8.4 shows that a control flit does not occupy the whole

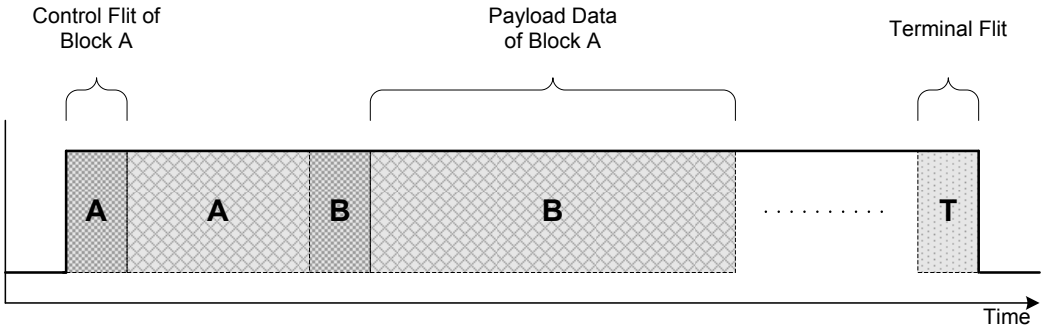


Figure 8.3: Protocol of reconfiguration in a configuration burst

width of the data lane, which is 32 bit in the current implementation. Thus, there are some bits left as "reserved" bits for future purpose. Besides this, the control flit contains 3 fields plus 1 particular control bit at index 20. For the control flit, the control bit at index 20 must be fixed at a value of 0. According to the protocol of on-the-fly reconfiguration, this indicates that the current flit, which contains meta-information, is a control flit. In the following we describe the 3 fields of the control flit.

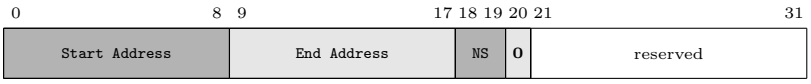


Figure 8.4: Layout of the control flit

`Name Space (NS)` As the TISS includes several physical memories that house configuration data, the control flit contains this field to specify, for which physical memory the configuration data in the payload of the current block is indented. `Name Space` is similar to `OCPS_MAddrSpace` of the Control Interface of the UNI, which introduces a notion of name spaces. However, `Name Space` refers to memories that can not be accessed through the Control Interface. As this field is 2 bit wide, it identifies 4 memories in the TISS. Table 8.1 lists the mapping of values of `Name Space` to the physical memory in the TISS.

`Name Space`	Physical Memory
00	Time-Triggered Communication Schedule
01	Routing Information Memory
10	Burst Configuration Memory
11	Register File and others

Table 8.1: Mapping of name spaces to physical memories in the TISS

`Start Address` This field specifies the start address in the physical memory identified by `Name Space`. According to the name spaces visible in the protocol of

on-the-fly reconfiguration, `Start Address` must be dimensioned in such a way that it can address the whole address space of all memories. In other words, its width equals the width of the address bus of the largest physical memory. In the current implementation, this are the memories of Time-Triggered Communication Schedule and Burst Configuration Memorywith 512 data words each, and therefore 9 bit width of the address bus . Consequently, `Start Address` has a width of 9 bit.

`End Address` Similar to `Start Address`, but this field denotes the end address in the physical memory identified by `Name Space`.

`Start Address` and `End Address` cover a continuous address space in the physical memory given by `Name Space`. Actually, `Start Address` corresponds to a physical address, where the very first data flit of configuration data of the subsequent payload has to be placed. Obviously, the data flits of the payload are written to consecutive addresses. As a consequence, a given block (in a configuration burst) conveys a continuous data chunk for a given physical memory in the TISS. If we want to update several regions in the same memory, we simple install an additional block in the configuration burst, which goes with the same `Name Space`, but with the appropriate `Start Address` and `End Address`.

The combination of `Start Address` and `End Address` determines, how many data flits of configuration data the payload of the current block in the configuration burst must possess. It is essential that the payload is made up an exact amount of data flits, given in equation 8.1.

$$\#\ \text{flits} = \texttt{End Address} - \texttt{Start Address} + 1 \tag{8.1}$$

The reason is that the protocol of on-the-fly reconfiguration does not define error conditions, e.g., the number of flits specified by the corresponding fields mismatches the real number of data flits in the payload. Otherwise, the protocol of on-the-fly reconfiguration would be messed up, and the Configurator would get irritated. In the end, this would leave the Configurator as well as the TISS in an undefined state.

Furthermore, we follow from equation 8.1 that the payload in a block in the configuration burst must span over at least one data flit, particularly when `Start Address` = `End Address`. Also we imply the reasonable condition that `End Address` >= `Start Address`.

However, as the configuration bursts originate from the TNA, we can rely on the fact that the configuration bursts are designed properly with reference to the protocol of reconfiguration with its constraints, and no violations of that protocol ever occur.

Layout of the Terminal Flit

The terminal flit marks the end of a configuration burst. As we learn from Figure 8.5, the terminal flit contains only one special field that is relevant for the protocol of

on-the-fly reconfiguration. For the terminal flit, the special control bit at index 20 must be fixed to a value of 1.

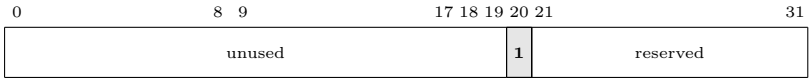


Figure 8.5: Layout of the terminal flit

Actually, the Configurator, which implements the protocol of on-the-fly reconfiguration in the TISS, is not aware of the number of blocks and their length in the current configuration burst. Therefore, the terminal flit serves to purpose to notify the Configurator that there will not follow any blocks of configuration data in the current configuration burst. As we see in section 8.4.3, this information coded in the terminal flit is required in order to let the Configurator finish its operation and go back to an idle state. In this context, it is essential that the configuration bursts and their blocks are properly designed according to the constraint of equation 8.1. The Configurator keeps track of the number of data flits in each payload. Thus, the proper number of data flits is required so that the Configurator can distinguish between configuration data and flits that contain meta-information.

8.4.3 The Configurator

The Configurator is the component in the TISS that redirects the configuration data in the payloads of configuration bursts from the TTNoC interface to the respective physical memories. It consumes the incoming data flits at the TTNoC interface, so that they are not written into the Port Memory through the Port Interface. So, the usual operation of processing bursts by the Port Manager is circumvented, and a special treatment is applied.

Connections of the Configurator

As illustrated in Figure 7.1, the Configurator occupies the write port of the dual-ported RAMs in the TISS, which are not accessible to the host through the Control Interface. Table 8.1 lists the mapping of name spaces in control flits. This implies that the Configurator is attached to the write ports of these physical memories. Each write port comprises an address bus, data bus, and a "write enable" signal. There is no byte masking involved, consequently the Configurator only writes complete data words into the memories, which equal the width of lanes of the TTNoC.

Moreover, the Configurator has special signals linked to some fields of the TISS-side Register File, so that the TNA has a means to assign values to these fields.

- The Configurator possesses direct access to `Host Mode (HM)` in the Register File (see section 5.2.4).
- The `Watchdog Period` also resides in the Register File and is controlled by the TNA via the Configurator.

Additionally, the Configurator sets particular control signals within the TISS:

- `PeriodEna`, which is relevant in the Burst Dispatcher. It activates and deactivates single periods for the dispatching (see section 7.3.3). Note that the value for `PeriodEna` in the associated data flit must be aligned to the most significant bit, if the number of supported periods of the periodic control system is less than the width of data flits. For instance, in the current implementation we can choose between 16 and 32 supported periods. In case of 16 periods, `PeriodEna(15)` resides at index 31 and goes down to `PeriodEna(0)` at index 16, while the bits 15 down to 0 of the data flit are not interpreted by the Configurator.
- override values for the value of the counter vector in the clock component, which embodies the global time base in the TISS. Hence, the Configurator has access to the clock component, therefore it is able to set a given value to the counter vector, whereas it bypasses the increment on each macro tick of that counter vector. As a result, the TNA has the proper means to control the (locally replicated) global time base in each TISS via the on-the-fly reconfiguration. Such a mechanism is essential to realize an external clock synchronization with a clock outside the TTSoC architecture.

  As the time format of the global time base is 64 bit and the width of the data bus of lanes in the TTNoC is configured as 32 bit, the new value for the global time base must be split into two data flits. The Configurator assembles these data flits and sets the value to the counter vector in the clock component in an atomic write operation.

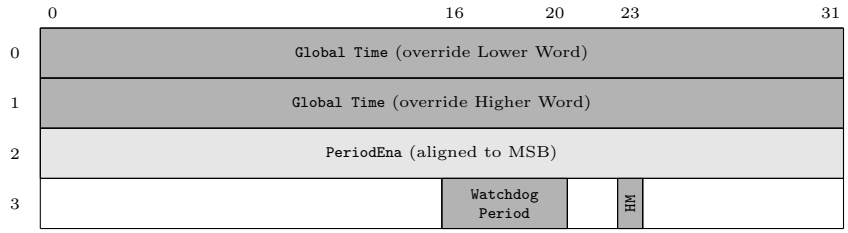


Figure 8.6: Entities of name space 11 visible to the TNA

Figure 8.6 shows, at which addresses in the name space 11 the specific entities are visible to the TNA. The Configurator decodes these addresses in that name space, selects the corresponding signal towards that entity, and sets the value as given in the associated data flit of the configuration burst.

State Machine of the Configurator

Like most components in the TISS, the Configurator is driven by a state machine, which is depicted in Figure 8.7.

The state machine of the Configurator includes a `Reset` state in order to handle the reset of the target hardware. Upon inactivity, the state machine remains in the `Idle` state. The cyclic processing of the state machine is triggered, when the Burst Dispatcher indicates a burst that refers to the reserved port 127. Afterwards the state machine enters the `RX_Wait` state.

The Configurator processes incoming data that is associated with a receive operation. Like the Memory Digger, the Configurator is dependent on the notification from the Receive Window Detector, which informs about the arrival of data at the incoming lane of the TTNoC interface. During the `RX_Wait` state, the Configurator waits for the notification signals from the Receive Window Detector (`RXValid`, `RXMiss`, `RXLate`). If there is no reconfiguration request in progress and no configuration data arrives, the Receive Window Detector will discover the `Miss` of the receive window. In this case, the Configurator aborts operation and goes back to the `Idle` state. Otherwise (if configuration data arrives), the Receive Window Detector drives `RXValid` = 1 and the state machine continues in the `Setup` state.

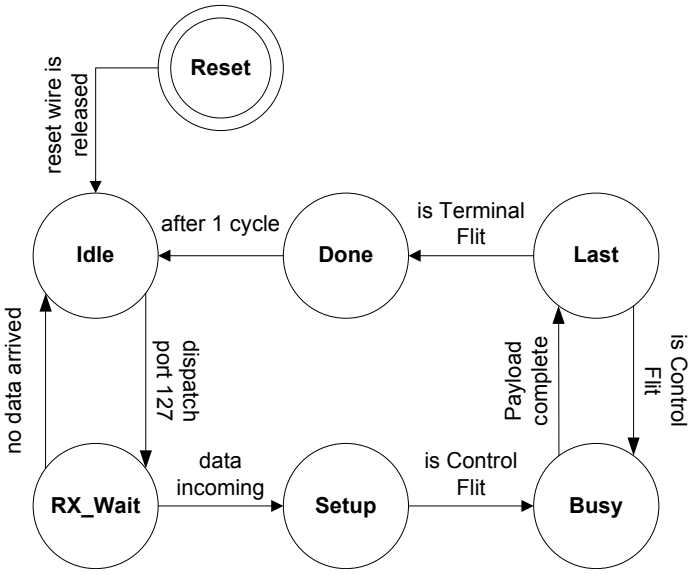


Figure 8.7: State machine of the Configurator

According to the protocol of reconfiguration, the very first data flit arriving at the TTNoC interface must be a control flit. This very first control flit is processed in the `Setup` state. The Configurator extracts all the fields in the control flit (`Start Address`, `End Address`, `Name Space`) and stores them in internal registers. Afterwards (after one clock cycle of system operation frequency), the state machine proceeds in the `Busy` state.

During the `Busy` state the Configurator redirects the configuration data from the TTNoC interface to this memory that has been identified by `Name Space` right before. Therefore, it drives the "write enable" signal to 1 and maintains the address bus as well the data bus of the memory associated with `Name Space`. The data

issued at the data bus is the same as the data bus of the incoming lane of the TTNoC interface, where the data flits of the current payload are available. The internal address register, which has been assigned with `Start Address`, drives the corresponding address bus. Additionally, it is incremented by one on each cycle of the system operation frequency, as long as the state machine is in the `Busy` state. If that internal address register equals `End Address`, all data flits of the payload of the current block in the configuration burst have been processed. So, the state machine enters the `Last` state.

According to the protocol of reconfiguration, in this state there must arrive another data flit with meta-information. If the "last flit" is a terminal flit, the next state is `Done`. Otherwise, if the configuration burst contains more than one block, this "last flit" is a control flit. Consequently, the Configurator has to continue to process another block in the configuration burst. It performs the same initialization of internal registers like in the `Setup` state and goes back to the `Busy` state. The state machine loops between `Busy` and `Last` state, until all blocks in the configuration burst have been processed, i.e., when the terminal flit has arrived during the `Last` state.

The `Done` state is the last state before the state machine completes one cycle of operation. This state is solely used for "housekeeping", for instance reset the internal registers. Moreover, the Configurator "remembers" the current configuration burst. The Configurator sets an internal register named `Pending`, which indicates that a configuration burst has just been processed and the memories of the TISS are equipped with new configuration data. As a consequence, `Pending` indicates the necessity for a reconfiguration instant to be indicated.

As introduced in section 8.4.1, the reconfiguration instant is realized as a burst of a "virtual fragment" that is synchronized among all TISSs. As the TNA can only supply one configuration burst for one TISS at a time, the Configurator has to postpone the triggering of the reconfiguration instant until all TISSs are equipped with new configuration data. This event correlates with the "virtual fragment", which is simultaneously dispatched among all TISSs. Therefore, if `Pending` $= 1$ and a subsequent burst referring to port 127 is dispatched, the Configurator triggers the reconfiguration instant and resets `Pending`. As a result, the reconfiguration instant occurs at each TISS simultaneously, and each TISS performs the procedure of initialization, as introduced in section 7.6.

Chapter 9

Prototypes & Results

In this chapter we present the results of working prototypes of the TTSoC architecture. We show the design of the prototypes and inform, how such an instance of the TTSoC architecture is mapped onto prototype hardware. Furthermore, we include statistics, which kind and how many resources the entities of the TTSoC architecture (e.g., a Fragment Switch, a TISS) occupy on the prototype hardware.

9.1 FPGA-based Prototypes

In the near future, we envision that the TTSoC architecture will incorporate ASIC technology. Then, the implementation of the TTSoC architecture will exist in lithographic masks in order to manifest in a silicon die. However, developing lithographic masks involves infrastructure and is expensive. Also, it is riskier, because design bugs can not that easily be corrected, once the production of silicon has been launched.

For this reason, the development and prototyping of the TTSoC architecture focuses on FPGAs. FPGAs are particularly convenient, as they are far more inexpensive to purchase and do not demand for infrastructure such as vacuum-clean laboratories. Due to their technology based on SRAM cells, they are flexible, and it is easier to alter designs and restart test series than on ASIC technology. Moreover, FPGA vendors provide complete development kits off-the-shelf, which entails professional support from the vendor as well as a user community.

Nevertheless, it must be mentioned that FPGAs have some drawbacks compared to ASICs. Generally, a given circuitry implemented in LUTs of an FPGA demands for up to 35 times more area, and is between 3 - 5 times slower than an ASIC implementation. Moreover, an FPGA consumes about 14 times more dynamic power than an equivalent ASIC on average [KR07].

9.1.1 Supported FPGAs

So far, the TTSoC architecture has successfully been ported to 3 different families of FPGAs. The following Table 9.1 shows the families of FPGAs and the specific devices of each family, which run instances of the TTSoC architecture.

FPGA Family	Device	Vendor Information
Altera Cyclone II™ Series	EP2C35 EP2C70	`http://www.altera.com/products/devices/`
Altera Stratix II™ Series	EP2S60	
Altera Stratix III™ Series	EP3SL150	

Table 9.1: FPGA families and devices running the TTSoC architecture

9.1.2 Resource Usage

As a result of the porting to different FPGA families, we present a summary in Table 9.3 and Table 9.4 about which kind of resources and how many of each resource in the FPGA are occupied by specific entities in the TTSoC architecture. These tables list the following resources that FPGAs provide:

logic elements (LE) These are the basic cells in the FPGA, which are used to embody circuitry. For instance, logic elements realize a comparator or a multiplexer. In general, logic elements are regarded as asynchronous, i.e., they are not driven by a clock signal.

logic registers (Reg.) A logic register is a basic cell in the FPGA, which holds the level of a signal. It is common practice that in synchronous designs such logic registers store the results of circuitry implemented in logic elements. This implies that a logic register is driven by a clock.

on-chip memory (Mem.) Even though, logic elements and logic registers are based on SRAM technology, state-of-the-art FPGAs are equipped with dedicated SRAM cells. This special purpose cells are intended to instantiate a RAM component in the design that is loaded into the FPGA. While the technological foundation of SRAM technology is transparent for logic elements and logic registers, the designer deliberately assigns on-chip memory to a RAM component in his/her hardware design.

Tuning the Current Implementation

The current implementation is dimensioned in such a way that it is unlikely that a target application built upon the TTSoC architecture will ever exceed the supported features, for instance the number of supported ports, or the size of the memories in

the TISS. Consequently, the current implementation is fit for various dimensions of target applications. Moreover, the current implementation denotes an upper bound of resource usage. Such an upper bound is helpful when we come about to draw conclusions about the feasibility of the TTSoC architecture (see section 10.4), as an instance of the TTSoC architecture to be deployed in the field will be smaller.

In fact, the current implementation can be stripped down. Like the parameters of the periodic control system, the TISS is configurable. It is possible to reduce the number of supported ports and cut down on data words in physical memories of the TISS. While the current implementation is a maximum configuration, we present the resource usage of a stripped down variation, which reflects a more "realistic"[1] of an instance of the TTSoC architecture. Table 9.2 compares the parameters of the current implementation and the realistic variation.

Parameter	current	realistic
Number of supported Ports	128	32
Data words in the Port Configuration Memory	128	32
Data words in the Port Synchronization Memory	128	32
Data words in the Time-Triggered Communication Schedule	512	128
Data words in the Burst Configuration Memory	512	128
Data words in the Routing Information Memory	128	32

Table 9.2: Parameters of current implementation and realistic variation

Interpretation of Resource Usage

Table 9.3 shows the resource usage of the entities of the TTSoC architecture, which have been introduced in the previous chapters. This is for the current implementation, and considers 16 and 32 supported periods in the periodic control system. Similarly, Table 9.4 deals with the realistic variation. The tables state the resource usage for all FPGA families, to which the TTSoC architecture has been ported. The synthesis and placement & routing tool used for these evaluations is Altera Quartus II™ version 8.0 SP1[2]. We can make a series of observations from the resource usage listed in Table 9.3 and Table 9.4.

Firstly, we can compare the resource demand among the different families of FPGAs. While the amount of reserved registers and on-chip memories remains approximately constant for all FPGA families, the number of logic elements deviates considerably. The main factor is the size of a Look-Up Table (LUT) in the specific FPGA. This parameter affects the number of logic elements it takes to realize a given basic function. For instance, the Cyclone II has LUTs with 4 input signals and 1 output, whereas the Stratix II features LUTs with 6 inputs and 1 output. If we want to build the basic function of a 4-to-1 multiplexer, this will take 1 LUT on

[1] At least what we expect to be a realistic dimensioning for deployment in industrial applications.
[2] `http://www.altera.com/products/software/sfw-index.jsp`

	Cyclone II			Stratix II			Stratix III		
	LE	Reg.	Mem.	LE	Reg.	Mem.	LE	Reg.	Mem.
Fragment Switch (4 interconnects)	582	160	0	302	160	0	236	181	0
Trusted Interface Subsystem	2704† / 4213‡	2271† / 3546‡	53248	1348† / 1901‡	2281† / 3579‡	53248	1482† / 2175‡	2342† / 3656‡	53248
Burst Dispatcher	1410† / 2883‡	1290† / 2556‡	0	585† / 1151‡	1306† / 2588‡	0	697† / 1397‡	1324† / 2622‡	0
Phase Comparator	18	15	0	9	16	0	11	16	0
Period Controller	66	61	0	10	54	0	15	62	0
Vector Coder	17† / 29‡	0	0	14† / 29‡	0	0	10† / 29‡	0	0
Multiplexers	381† / 799‡	39	0	211† / 453‡	39	0	214† / 463‡	39	0
Clock Component	67	66	0	56	66	0	57	67	0
Configurator	313	290	0	53	290	0	58	291	0
Port Manager	284	118	0	237	116	0	270	122	0
Routing Processor	27	15	0	23	15	0	27	16	0
Receive Window Detector	31	10	0	22	9	0	30	10	0
Address Calculator	53	0	0	79	0	0	80	0	0
Memory Digger	79	48	0	40	48	0	48	49	0
Time Stamper	34	20	0	17	19	0	24	21	0
Port Synchronization Controller	60	25	0	56	25	0	61	26	0
Register File	297	225	0	191	223	0	212	225	0
Time-Triggered Communication Schedule	0	0	25600	0	0	25600	0	0	25600
Burst Configuration Memory	0	0	15360	0	0	15360	0	0	15360
Port Configuration Memory	0	0	4096	0	0	4096	0	0	4096
Port Synchronization Memory	0	0	4096	0	0	4096	0	0	4096
Routing Information Memory	0	0	4096	0	0	4096	0	0	4096

Table 9.3: Resource usage of entities of the TTSoC architecture (current implementation)

†16 periods, ‡32 periods

Fragment Switch (4 interconnects)	Cyclone II			Stratix II			Stratix III		
	LE	Reg.	Mem.	LE	Reg.	Mem.	LE	Reg.	Mem.
	582	160	0	302	160	0	236	181	0
Trusted Interface Subsystem	2523† 3864‡	2107† 3261‡	12288	1288† 1823‡	2117† 3286‡	12288	1423† 2072‡	2179† 3365‡	12288
Burst Dispatcher	1264† 2585‡	1154† 2292‡	0	541† 1071‡	1170† 2324‡	0	649† 1301‡	1188† 2358‡	0
Phase Comparator	18	15	0	11	16	0	13	16	0
Period Controller	54	53	0	10	53	0	15	54	0
Vector Coder	17† 29‡	0	0	13† 30‡	0	0	11† 25‡	0	0
Multiplexers	301† 631‡	31	0	168† 374‡	31	0	171† 367‡	31	0
Clock Component	67	66	0	56	67	0	57	68	0
Configurator	294	272	0	47	272	0	56	273	0
Port Manager	271	115	0	233	113	0	268	120	0
Routing Processor	15	13	0	22	13	0	25	14	0
Receive Window Detector	30	9	0	23	9	0	31	10	0
Address Calculator	53	0	0	78	0	0	80	0	0
Memory Digger	79	48	0	38	48	0	47	49	0
Time Stamper	34	20	0	16	18	0	24	21	0
Port Synchronization Controller	60	25	0	56	25	0	61	26	0
Register File	290	205	0	187	205	0	212	206	0
Time-Triggered Communication Schedule	0	0	5376	0	0	5376	0	0	5376
Burst Configuration Memory	0	0	3840	0	0	3840	0	0	3840
Port Configuration Memory	0	0	1024	0	0	1024	0	0	1024
Port Synchronization Memory	0	0	1024	0	0	1024	0	0	1024
Routing Information Memory	0	0	1024	0	0	1024	0	0	1024

†16 periods, ‡32 periods

Table 9.4: Resource usage of entities of the TTSoC architecture (realistic variation)

the Stratix II: 4 of the input signals are occupied by the multiplexer's data input signals, and the remaining 2 inputs can be used for the selectors of the multiplexer. The Cyclone II has to cascade 2 LUTs, as 2 of the 4 input signals have already been taken by the multiplexer selectors, and only 2 other input signals are left for the multiplexer's data input. To sum up, the internal structure of the FPGA affects the number of used logic elements. However, this is no result from the implementation of the TTSoC architecture, but a technological issue.

Secondly, if we focus on a given FPGA family, we notice a ratio of resource usage among the different entities of the TTSoC architecture. It is evident from Table 9.3 and Table 9.4 that the Burst Dispatcher imputes the major part of resource usage on the TISS. Depending on the number of supported periods in the periodic control system, the Burst Dispatcher claims about 50 % (16 periods) up to about 70 % (32 periods) of logic elements and registers for all FPGA families.

In comparison, the Configurator and the Port Manager each consume resources in the magnitude around 10 %. **Apparently, the resource usage of entities does not reflect the architectural complexity and mightiness of features of the TTSoC architecture, which those entities realize.** For instance, the Port Manager is in charge of handling multiple encapsulated communication channels, it realizes the port synchronization and deals with different semantics of ports. Also, the Configurator implements an application-level protocol, which entails some complexity. Though, Port Manager and Configurator make up a fraction of the total resource usage of the TISS. Actually, the task of dispatching in the Burst Dispatcher with its phase comparators and period controllers, which seems to be so primitive, is crucial when it comes to the number of logic elements and registers.

In this context, we can say that the support for multiple encapsulated communication channels with all their port synchronization and semantics approximately causes the same usage of logic elements and registers as the ability of on-the-fly reconfiguration.

Within the Burst Dispatcher, the multiplexers for the address bus and the payload output from the period controllers represents the biggest single component, i.e., nearly a third of the Burst Dispatcher. While a pair of period controller and phase comparator is relatively tiny, the quantity of pairs leads to the biggest part of resource usage.

Resource Estimation for ASIC

Even though the current implementation of the TTSoC architecture focuses on FPGA technology, we have conducted experiments to find out, how many resources (e.g., area, power) the current implementation would require for ASIC technology, and which timing constraints (i.e., maximum frequency of the system operation frequency) are feasible.

We investigate a Fragment Switch and a TISS like in the previous section. We use the Synopsis Design Compiler (version Y-2006.06) to synthesize[3] both entities with the *Oklahoma State University's FreePDK* 45 nm open source standard cell library[4] as target library.

Table 9.5 gives information about the maximum frequency, estimated area (transistor number), and estimated power consumption (dynamic plus static power at the given maximum frequency) of Fragment Switch and TISS. Note that the Fragment Switch is the default configuration with 4 interconnects at 32 bit data bus width in lane of the TTNoC, whereas the TISS has been synthesized with 16 periods in the current implementation (not the realistic variation).

Parameter	Fragment Switch	TISS
maximum f_{sys}	$1,5$ GHz	1 GHz
estimated power consumption	$\approx 10,5$ mW	$\approx 62,2$ mW
area consumption (cell units)	$6206,02$	$56644,51$
estimated transistor number	≈ 25 k	$\approx 225 - 230$ k

Table 9.5: Results of the synthesis with a 45 nm library

The results of these experiments are rough estimations, so that we can draw conclusion about the costs of the TTSoC architecture when brought to silicon. Actually, the values for the TISS do not incorporate memory cells, but just logic cells. Furthermore, the value for area are not the number of transistors, but "cell units" or "gates" based on the smallest cell in this library. If we want to estimate the transistor number of Fragment Switch and TISS, we have to guess to number of transistors for that smallest, basic cell and multiply by the number given in Table 9.5. Usually, the basic cell is the NAND and NOR gate, which can be implemented by 4 transistors in CMOS technology. According to this estimation, a Fragment Switch would take about 25.000 transistors, whereas a TISS reserves about $225.000 - 230.000$ transistors in ASIC technology.

9.2 Prototype Hardware

For each FPGA family we use specific prototype hardware. The following section introduces prototype hardware used for the current implementation.

[3] We can just present the result of synthesis, as a placement & routing tool for ASIC has not been available.

[4] `http://avatar.ecen.okstate.edu/projects/scells/`

TTTech MPSoC Development Kit

The MPSoC Development Kit is customized hardware, which has been manufactured by TTTech[5]. It is the primary prototype hardware for the current implementation of the TTSoC architecture.

The MPSoC Development Kit consists of several PCB devices equipped with FPGAs of the Altera Cyclone II™ series and different CPUs.

Mainboard The basis of this prototype set-up is a Mainboard that provides 9 Powerlink™ [TTT06] extension slots, on which the other PCB devices can be mounted as *add-on boards*. Furthermore, the PCB devices can be stacked vertically beginning at the Powerlink™ extension slots on the Mainboard, as illustrated in Figure 9.1. Besides the Powerlink™ extension slots, the Mainboard posseses one Cyclone II EP2C70 FPGA.

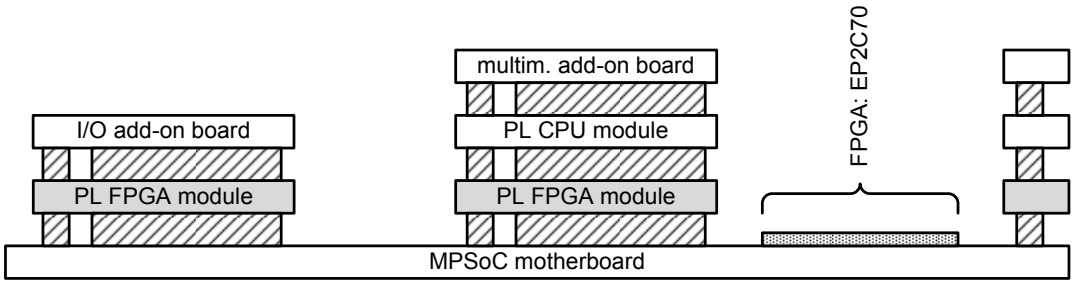


Figure 9.1: Stacking add-on boards on the MPSoC Development Kit

CPU Board A CPU Board is exclusively used to realize the application computer of a host in the TTSoC architecture. As its name suggest, these add-on boards contain CPUs in a stand-alone package. The MPSoC Development Kit entails different variations of CPU Boards with a specific processor each. The following assembly of CPU Boards is available:

- Freescale MPC555, 4 MByte Flash memory, 512 kByte SRAM
- Freescale MC9S12XDP512, 2 MByte Flash memory, 512 kByte SRAM
- Infineon TriCore TC1796, 4 MByte external Flash memory, 4 MByte internal Flash memory, 1 MByte SRAM
- Infineon C167, 1 MByte Flash memory, 512 kByte SRAM

FPGA Board An FPGA Board is assembled with an Altera Cyclone II™ EP2C35. This FPGA is free to take particular entities of the TTSoC architecture. On the one hand, an FPGA Board can be used to hold a Front-End in order to wrap the physical signals of the UNI to an interface of a CPU in a CPU Board.

[5]TTTech Computer Technik AG, `http://www.tttech.com`

On the other hand, an FPGA Board can also contain a Front-End plus a "soft-core" CPU (e.g., a LEON3 SPARC V8 Processor core[6], an Altera Nios II™ Embedded Processor[7]). In the latter case, the FPGA Board embodies a complete host in the TTSoC architecture.

Basic I/O Board The Basic I/O Board provides interfaces to the external environment of a TTSoC design. Usually, a Basic I/O Board will be stacked upon an FPGA Board, so that the host in the FPGA Board can operate the physical interfaces of the Basic I/O Board. It features a serial RS232, CAN, LIN, TTP/A, Ethernet, and GPIO.

Multimedia Board Similar to the Basic I/O Board, but the interfaces can be regarded as multimedia devices. For instance, a Multimedia Board is equipped with an AC97 compatible audio device. Also, a colour touch screen LCD display can be driven by the Basic I/O Board. Moreover, the Multimedia Board contains an USB controller that can function as an USB host as well as an USB device.

Note that the MPSoC Development Kit does not provide a single FPGA to hold a whole design of the TTSoC architecture. Instead of this, all entities are spread across several FPGAs of FPGA Boards or CPUs of CPU Boards. Thus, the MPSoC Development Kit emulates an SoC. The reason for this hardware partitioning is that no single FPGA, which would have been big enough to house a complete TTSoC design, had been available at that time, when the development of the TTSoC architecture had been started.

Section 9.3 describes a design example of the TTSoC architecture, which has been built around the MPSoC Development Kit.

Altera Nios II™ Development Kit

The Altera Nios II™ Development Kit[8] is an commercial off-the-self product, which is intended for designing and prototyping of a wide range of embedded applications. Altera supplies the soft-core Nios II Embedded Processor, a library of IP-components that drive the on-board hardware devices, software drivers, the light-weight embedded real-time operating system μC/OS-II[9], and a comprehensive tool chain with this development kit. The Nios II™ Development Kit is assembled with a Stratix II™ EP2S60 FPGA, 16 MByte DDR SDRAM, 1 MByte SSRAM, 16 MByte Flash memory, 1 serial RS-232 interface, a 10/100 Ethernet device, and a character LCD display. For details about this developer kit have a look at the vendor documentation.

[6] http://www.gaisler.com/cms/index.php?option=com_content&task=section&id=4&Itemid=33

[7] http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[8] http://www.altera.com/products/devkits/altera/kit-niosii-2S60.html

[9] http://www.micrium.com/products/uc-products.html#ucosii

We use the Nios II™ Development Kit to surplus the MPSoC Development Kit. As the MPSoC Development Kit features physically displaces FPGAs and just emulates a physical SoC, we have had the desire to experiment with prototype hardware that is equipped with a single FPGA. In this single FPGA we set up an instance of the TTSoC architecture. Certainly, this instance must be smaller than it would be possible on the MPSoC Development Kit to fit into the EP2S60 FPGA. Anyhow, this experiment gives as indications, how the TTSoC architecture behaves as a "real" SoC. The main interest is to have evidence about the performance (i.e., maximum system operation frequency) of the TTSoC architecture.

We have built an instance of the TTSoC architecture with 4 micro components, which each consists of Nios II CPUs, a Front-End, Port Memory, some peripherals (e.g., JTAG UART for debugging, memory controllers for DDR SDRAM and Flash memory), and a TISS. One single Fragment Switch (with 4 interconnects) embodies a minimal TTNoC that connects all micro components. The micro components take the roles of the TNA, RMA, and 2 general purpose micro components.

The frequency of the macro tick f_{mt} is fixed at 2^{-20} $sec \approx 1,04$ MHz. The Stratix II FPGAs provides a maximum frequency for the system operation frequency f_{sys} that drives the TSS of about 133 MHz, but we use 100 MHz to achieve a stable design.

Altera Stratix III™ Development Kit

The Altera Stratix III™ Development Kit[10] is another commercial off-the-self product. It serves the same purpose as the Nios II™ Development Kit. The vendor includes the same set of software / soft-core components. However, the hardware assembly of the board is more advanced. The basis of the Stratix III™ Development Kit is a Stratix III EP3SL150 FPGA. Besides this, the development kit features a GBit Ethernet device, several DDR2 SDRAM chips and an UDIMM slot, Flash memory, and other peripherals like an OLED graphics display, and USB interfaces. For details look up the data sheet at the vendor's website.

With the EP3SL150 FPGA we possess a hardware platform, which offers enough resources to come close to the scale of the MPSoC Development Kit. As the MPSoC Development Kit has been our main prototype hardware, we use the Stratix III™ Development Kit to gain results on a state-of-the-art FPGA series. Actually, we run the same instance of the TTSoC architecture like on the Nios II™ Development Kit. However, the performance estimation are more impressive. While we fix the frequency of the macro tick f_{mt} at 2^{-20} $sec \approx 1,04$ MHz, the TSS achieves a maximum frequency for the system operation frequency f_{sys} of about 220 MHz. We have the design operating at 200 MHz to assure the stability of the FPGA design.

[10] `http://www.altera.com/products/devkits/altera/kit-siii-host.html`

9.3 Design Example

The MPSoC Development Kit is the main prototype hardware. We run the design example, which is introduced in this section, on this hardware. The design example is the instance of the TTSoC architecture, which we use to construct an demonstration application [OFSH08]. Figure 9.2 graphically describes the essence of this design example.

9.3.1 Structure of the Design Example

The design example on the MPSoC Development Kit features 9 hosts. The RMA as well as the Gateway occupy one host each. The DU reserves another one (at slot 0). The 6 remaining hosts are free for the DASs of the demonstration application. The TNA is a stand-alone unit and is not realized on a host.

In total, the design example features 10 TISSs, which are connected by a TTNoC consisting of 6 Fragment Switches. The Fragment Switches are arranged into an 2×3 mesh topology in this design example.

Moreover, we learn from Figure 9.2 that the TSS (the 6 Fragment Switches, the 10 TISSs, and the TNA) resides in the EP2C70 FPGA of the Mainboard. The 9 hosts are implemented in the EP2C35 FPGAs of the FPGA Boards mounted on the Powerlink^{TM} extension slots of the Mainboard. Basic I/O Boards and Multimedia Boards, which are required by the demonstration application, are not mentioned in this section.

Each host utilizes one Nios II Embedded Processors as its application computer. Besides this, a host possess memory controllers to operate the memory chips of the FPGA Boards, and other peripherals such as a JTAG UART for debugging. All peripherals are interconnected by the Altera Avalon Memory-Mapped Interface [Cor08]. Considering the entities of the TTSoC architecture, a host must be equipped with a Port Memory (16 KByte on-chip memory) and a Front-End suitable for the Avalon Memory-Mapped Interface. Besides this, the Gateway is a host that features an additional Time-Triggered Ethernet (TTE) Controller [KAGS05].

In fact, the design of the TNA is similar to a host, except for the number of memory controllers, as there a not as many memory chips available on the Mainboard as on an FPGA Board.

9.3.2 The Need for Serialization

The design example physically separates hosts from the TSS. As illustrated in Figure 9.2, the cut line of this physical separation is the UNI. We know from section 5.1.1 and section 5.2.1 that the Port Interface and Control Interface demand for 181 physical wires. Even though Powerlink^{TM} provides about 140 pins, only 61 pins are available per slot on the Mainboard. Unfortunately, this design restriction entailed

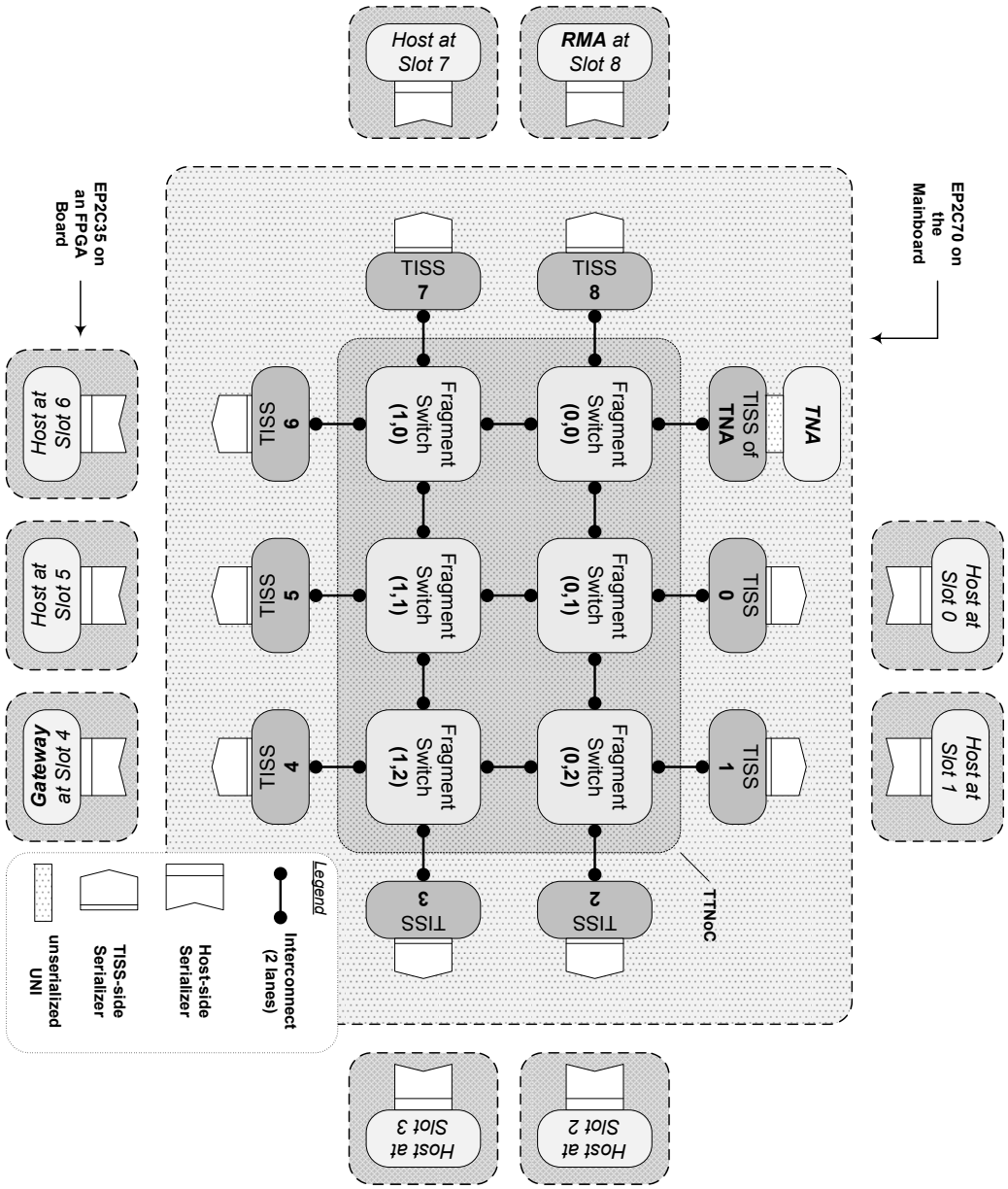


Figure 9.2: Partitioning of the prototype design on the MPSoC Development Kit

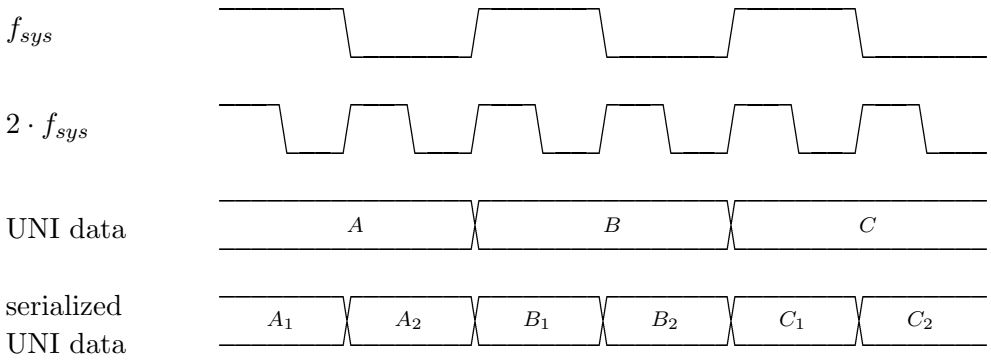


Figure 9.3: Serializing the UNI

by the MPSoC Development Kit leads to the necessity to serialize the physical signals of the UNI over the usable Powerlink^{TM} wires.

For the purpose of serialization, all buses like the address buses `OCPM_MAddr` and `OCPS_MAddr` are split into two halves. Moreover, data buses of the Port Interface as well as the Control Interface are combined into a tri-stated bus. In fact, `OCPM_MData` and `OCMP_SData` share the same wires, respectively `OCPS_MData` and `OCPS_SData`. The serialized wires of the Powerlink^{TM} extension slots operate at the double system operation frequency. Thus, during one clock cycle of the system operation frequency the accelerated serialization clock makes two clock cycles. This allows to transport the first and the second half of the serialized UNI signals within one clock cycle of the system operation frequency. Figure 9.3 visualizes the concept of serialization for the MPSoC Development Kit.

Note that for the TTSoC architecture serialization is transparent. The TISSs nor the hosts are aware of the fact that all data exchanged via the UNI is serialized. However, serialization extends the latency Δ^+ by one clock cycle, as the data flit to be written into the Port Memory takes that clock cycle to pass through the serialization. For send operation, the request of a given data word from the Port Memory imposes one clock cycle like for read operations. Additionally, the response from the Port Memory also travels one clock cycle longer due to serialization. As a consequence, Δ^- increases by 2 clock cycles.

9.3.3 Drawbacks of the MPSoC Development Kit

Even though (from the aspect of implementation), serialization is transparent, it impairs the achievable maximum frequency of the TSS. The reason is the long wiring from the Mainboard's EP2C70 FPGA to the EP2C35 FPGA of the FPGA Boards via the Powerlink^{TM} extension slots. The capacities of the wires and the signal runtimes on these wires restricts the maximum frequency. We are forced to operate the serialized clock at about 32 MHz in order to have reliable off-chip communication.

This value has been determined by experiments. As the system operation frequency is the half of the serialized clock, this implies a maximum frequency of the system operation frequency of $f_{sys} \approx 16$ MHz.

To achieve better performance estimations about the current implementation of the TTSoC architecture, we call on the Nios II™ Development Kit and Stratix III™ Development Kit, which allow to have a single FPGA solution. Actually, as described in section 9.2 and section 9.2, these alternative prototype hardware configurations outperform the MPSoC Development Kit by a multiple.

Chapter 10

Conclusion

In this last chapter, we present the conclusions of the implementation of the TTSoC architecture. Each section deals with one specific aspect of the "lessons learned".

10.1 Power Awareness

We have learned from earlier chapters that the TTSoC architecture incorporates several mechanisms in its design in order to address issues of power saving and energy efficiency. Let us summarize these power-aware features:

- Fragment Switches automatically remain in a power saving idle mode in case of inactivity. Additionally, the integrated resource management can explicitly powered down each Fragment Switch.
- The integrated resource management can harness these properties of Fragment Switches to establish "power-aware routing", whereas encapsulated communication channel are concentrated on a given subset of Fragment Switches, while the Fragment Switches excluded from traffic can idle or be power down.
- The TNA has the ability to activate and deactivate pairs of period controllers and phase comparators in the Burst Dispatcher of each TISS, if there are no communication activities involved in the associated periods.
- The field `HostMode` in the Register File of a TISS let the host know, whether the services realized in that host are currently needed by some application subsystem, or not. If not, it is in the sphere of control of the host to enter any power saving mode.
- The TTNoC allows native multi-casting without the need of additional control logic and further semantics. So, we obtain multi-casting at no additional costs with respect to area (i.e., logic elements and registers in FPGA technology) and power consumption.

- We can regard the non-existence of buffers during the transportation of data as a power-aware design choice. No buffers mean no further registers or RAM cells, which would consume power. Additionally, the absence of explicit buffers avoids accessing these elements by means of power consuming read and write operations. So, the omission of buffers contributes to lower power consumptions from the structural, i.e., absence of buffers, as well as the dynamic, i.e., read / write operations, point of view.

- The implementation of the TTSoC architecture shows that specific circuitry is reused for other functions than originally intended.

  For instance, we harness the circuitry of dispatching (in the Burst Dispatcher) to realize "virtual fragments", reconfiguration instants, and to realize the watchdog and dissemination service. So, we leverage the control logic of dispatching of communication activities for message ordering, on-the-fly reconfiguration, and supplemental services. Another example for re-usage of circuitry is the fact that the TTNoC conveys application data, reconfiguration data (i.e., configuration bursts) as well as routing information over the same physical wiring.

  Such re-usage not only demonstrates the elegance of design and implementation of the TTSoC architecture, but it yields in further savings of power consumption, as there is no distinct circuitry for each of these functions necessary.

The features listed above concern an architectural scope. They illustrate, how the TTSoC architecture is able to address power issues from a high-level respectively architectural point of view. In fact, the concrete method (of power saving) that is combined with any of the architectural concepts depends on the capabilities of target technology.

For example, state-of-the-art FPGAs do not well support advanced low power methods like *power gating* or *voltage and frequency scaling* [KFA+07,RSG03]. FPGA technology is able to implement *clock gating* to reduce the amount of dynamic power consumption. But it can not totally turn off the power supply towards parts of the design loaded on the FPGA (e.g., entities of the TTSoC architecture like a Fragment Switch), thus also diminishing static power consumption, which would be possible by means of power gating. Moreover, it is hardly possible[1] to spontaneously drop or lift the supply voltage among power distribution networks within the FPGA, or to steadily tune the frequency of clock distribution networks.

Nevertheless, such lower power methods have been proven to work on ASIC technology, whereas prominent examples are SpeedStep™ by Intel or Power Now!™ and Cool'n'Quiet™ by AMD. Even though, we are momentarily tied down to clock gating as the only means of power saving on our FPGA based prototype hardware, due to its power-aware features the TTSoC architecture is prepared to harness the full repertoire of low power methods once ported to ASIC technology.

[1] considering the state-of-the-art of mainstream FPGAs by the time of writing this thesis

10.2 Growth of the TISS

It is evident from Table 9.3 and Table 9.4 that the TISS grows into two distinct dimensions:

1. logic elements and registers
2. on-chip memory

Each of these dimensions is influenced by one parameter of the TTSoC architecture.

number of supported periods in the periodic control system directly affects the first dimension (logic elements and registers)

number of ports is the single factor that controls the demand for on-chip memory. Actually, the 5 memories (i.e., Time-Triggered Communication Schedule, Burst Configuration Memory, Port Configuration Memory, Port Synchronization Memory, and Routing Information Memory) are the only components in the TISS, which claim on-chip memory.

We can identify a slight interdependency between number of ports (number of supported encapsulated communication channels) and logic elements. If we compare the current implementation and the realistic variation, we learn that the less ports we have, the smaller the Burst Dispatcher becomes. Apparently, a reduction of ports not only cuts down on records in the respective memories. Additionally, some fields in the entries of the Time-Triggered Communication Schedule, which refer to records of other memories (e.g., `RIRef`, `PortRef`, or `BurstRef`), tend to shrink, if there are fewer records that can be addressed. As a consequence, the multiplexers in the Burst Dispatcher become smaller, because the affected fields also belong to the payload output of period controllers and therefore flow through the multiplexers. We can explains the deviations in other components such as the Port Manager owing to smaller reference fields, however the fluctuations in logic elements and registers are of minor scale in those components.

From the other side, modifying the number of supported periods in the periodic control system solely affects the Burst Dispatcher. The memories do not change their demand for on-chip memory, as well as the other components in the TISS, e.g., Port Manager, Configurator, show now fluctuations in logic elements and registers that would be worth mentioning.

All in all, **we trivially identify a linear growth of the resource usage of the TISS in both dimensions**. The more ports and encapsulated communication channels we have, the more on-chip memory we have to spend on the respective memories. The more periods we want to utilize in the periodic control system, the more logic elements and registers we have to spend on pairs of period controllers and phase comparators in the Burst Dispatcher.

Actually, that parallel hardware in the Burst Dispatcher causes the major part of resource usage in the TISS, and therefore bears the main responsibility for the resource usage of a TISS. Each period of the control system gets dedicated hardware (i.e., a pair of period controller and phase comparator) assigned, which operates simultaneously. Anyhow, due to design constraints it is not possible for the TISS to dispatch more than one communication activity at a given instant of time. As a consequence, let us question the design choice of parallel hardware in the Burst Dispatcher, whether it is necessary to supply parallel hardware as long as we can not harness the immanent parallelism.

The alternative to parallel dispatching is a serialized determination of instants of communication activities. Assume one mighty, fictitious Burst Dispatcher with one pair of a phase comparator and period controller. This single pair is able to indicate the arrival of instant of bursts, which belong to messages of all supported periods. The entries in the Time-Triggered Communication Schedule still reflect the (temporal) sequence of communication activities. As we just have one comparator unit in this Burst Dispatcher, we just have one global period p_G left. In order to model the periodicity of periods, which are shorter than this global period, the Time-Triggered Communication Schedule must include a given communication activity (of the shorter period) multiple times. According to the ratio between periods $\mathcal{R}_p$ (see equation 4.1), a burst b_P in period p_P must occur

$$\mathcal{R}_p(G, P) = 2^{\delta \cdot |G-P|}$$

times in the Time-Triggered Communication Schedule. For instance, if we fix p_G to period #15 like in Figure 4.2 (period delta $\delta = 1$) and assign period #4 to p_P, the burst b_P would be present $2^{11} = 2048$ times in the Time-Triggered Communication Schedule. In comparison to this alternative design, in the ordinary way with parallel hardware we would need an additional pair of phase comparators and period controller for period p_P besides other periods, however just one single entry in the Time-Triggered Communication Schedule, which corresponding circular, linked list is dedicated to period p_P.

The approach of serialized dispatching brings in a "simple" Burst Dispatcher, which size remains constant despite the support of several periods. Thus, it has an order of $O(1)$. However, the "unrolling" of periodicity leads to exponential growth of the Time-Triggered Communication Schedule, as $O(\mathcal{R}_p) = O(e^n)$. In contrast to such an exponential growth, the approach of parallel dispatching entails a linear growth $O(n)$ of pairs of period controllers and phase comparators as well as memory of the Time-Triggered Communication Schedule.

To sum up, parallel hardware embodies the best practice to achieve dispatching of communication activities in multiple periods due to the argumentation in terms of complexity theory.

10.3 Timing Issues

Dispatching of communication activities not only causes costs with respect to resources of a given target technology. We learn from section 7.5 that it also imposes a temporal latency on the operation of the communication subsystem. The variables Δ^+ and Δ^- model this latency for receive and send operations of bursts. Even though, this latency is predictable and deterministic, we should reflect the consequences of that latency.

10.3.1 About Latency

The amount of Δ^+ and Δ^- has two causes that originate from features of the TTSoC architecture.

1. The support for multiple periods in the periodic control system requires parallel hardware, which operation is concurrent. However, the parallelism must be resolved to a single entity. To be more precise, the Burst Dispatcher maintains several dispatching units, however it has to multiplex information concerning the current burst and give a single chunk of information to the remaining TISS, because the TISS can only operate one burst at a time. As we see in section 7.3.2, this single process takes 3 clock cycles of the system operation frequency.

2. We allow a certain degree of freedom to the host. For instance, the host manages the layout of the Port Memory on its own. Additionally, the host decides the semantics of messages (i.e., state or event semantics), implicit or explicit synchronization and so forth. Consequently, we need a Port Configuration Memory, where host exchanges information about its decisions with the TISS.

   The TISS must incorporate such information for each burst. This is the reason, why the TISS (i.e., the Port Manager) has to load information corresponding to the port (to which the fragment of the current burst belong to) for each execution of a burst. Then, the TISS processes the information during address calculation in the Address Calculator, port synchronization in the Port Synchronization Controller etc.

Certainly, this degree of freedom and the periodic control system demand for additional time to be realized. However, we do not understand such latency as a penalty, but as a necessary expense of an intended feature of the TTSoC architecture. In fact, such a mighty periodic control system aids an application designer, as it easily allows to model periodic behaviour. Furthermore, keeping some aspects of application subsystems local to the architectural units, where the application subsystem is realized, contributes to encapsulation and therefore complexity reduction.

We are the opinion that it is more valuable for a system architecture (such as the TTSoC architecture) to assist the application designer to cope with architectural

complexity than to insist on implementation characteristics like latency. Considering the vast benefits of complexity reduction on an application-level scale, the aspect that it takes a few clock cycles before send or receive operations take place is neglectable.

10.3.2 Justification for Bursts

We have a latency of **at least 7 clock cycles** of the system operation frequency for receive as well as send operations. In this context, we question the efficiency of dispatching and setting-up of communication activities.

As the TTSoC architecture is intended for several fields of target applications, let us investigate some scenarios. For instance, in control applications it is common to exchange state information between sensors and actuators. Usually, such information consists of a relatively small data word. In the TTSoC architecture we could transmit such data in one single data flit.

If we understand the latency as overhead that is necessary to send and receive data, the efficiency E for send as well as receive operations is as follows:

$$E = \frac{1}{\Delta^{+,-}} \leq \frac{1}{7} \approx 14\%$$

Hence, in order to transmit a single data flit in the TTSoC architecture, we just achieve an efficiency of about 14 % and below. Honestly, this is disillusioning.

During the development of the TTSoC architecture we have foreseen such unpleasant results considering efficiency and overhead of transactions. In fact, Δ^+ and Δ^- embody "lower bounds", which can not be further reduced. There is only one means to overcome such concerns: to introduce bursts. Bursts feature the transmission of data in each clock cycle of the system operation frequency. As result, they demand for one single dispatching and set-up, then this single overhead is better utilized. So, the efficiency E is modified to incorporate the number of data flits, which is given by the fields `End Offset` $-$ `Start Offset` in the Burst Configuration Memory, processed during a burst:

$$E = \frac{\texttt{End Offset} - \texttt{Start Offset}}{\Delta^{+,-}}$$

Thus, the application of bursts particularly pays off for `End Offset` $-$ `Start Offset` $\gg \Delta^{+,-}$. For instance, this is the case for multimedia application with high bandwidth demands, so that the bursts are dimensioned to convey large quantities of data flits.

To sum up, latency embodies overhead that impairs the efficiency of send and receive operations in the TTSoC architecture. For very small burst, we can not fade out this overhead. However, for large burst the overhead becomes negligible.

	LE	Reg.	Mem.
Nios II/s factory design	6.312	5.104	1.647.744
Nios II/f factory design	7.843	6.472	1.686.336
Nios II/s CPU	1.131	893	575.488
Nios II/f CPU	2.349	2286	614.080
Altera DDR2 Controller IP-core	1.708	1.282	1.152
Fragment Switch (4 interconnects)	**236**	**181**	**0**
TISS (16 periods)	**1.423**	**2.179**	**12.288**
LEON III factory design	11.483	4.895	291.840
LEON III CPU	5.739	1.944	250.368
GR Ethernet MAC IP-core	2.331	1.104	20.992
AMBA AHB Controller	393	30	0
AMBA APB Controller	195	89	0

Table 10.1: Comparing resource usage of exemplary designs (Stratix III)

10.4 Feasibility of the TTSoC Architecture

So far, we have seen how to build the TTSoC architecture. Also, we have learned which resources of target technology such as FPGA or ASIC it takes. In fact, several features of the TTSoC architecture like support for multiple periods in the periodic control system, or the existence of multiple encapsulated communication channels demand for some costs with respect to these resources.

From the local point of view, it seems to be alarming that, e.g., the Burst Dispatcher occupies the half of a TISS. However, we have to take under consideration, how the TTSoC architecture puts up from the point of view of a whole system. To be more precise, we are interested in the answer of the questions, whether it is feasible to realize the TTSoC architecture in a competitive way.

In order to compare the TTSoC architecture we present the results of some experiments. We have investigated the resource usage of exemplary micro controller designs, which could be used to implement a host of the TTSoC architecture. The results are listed in Table 10.1. These experiments have been conducted on an Altera Stratix III^{TM} FPGA of the Stratix III^{TM} Development Kit.

The Nios II designs are factory design examples, which are distributed as part of the Stratix III^{TM} Development Kit's tutorials and developer documentation. Same applies to the SPARC LEON III design, which is a factory design example supplied by the GRLIB IP Library[2]. Both designs represent minimal designs of hosts with some extend of potential and performance. However, we expect "real" hosts to be even more capable and powerful, for instance hosts with multiple CPU cores, AES encryption IP-cores, TMR-voting engines etc.

[2] `http://www.gaisler.com/cms/index.php?option=com_content&task=section&id=13&Itemid=125`

When we incorporate the numbers of Table 9.4 of the realistic variation of the TTSoC architecture, we find that a TISS's resource demands are in the magnitude of an Ethernet Controller (i.e., the GR Ethernet MAC IP-component), or approximately of an DDR2 Controller, or even a light-weight embedded soft-core processor like the Nios II. A Fragment Switch is competitive with the AMBA AHP Controller.

Actually, even in such minimal design examples for hosts like the LEON III design example, a TISS would occupy about 12 % of logic elements, 44 % of registers, and 4 % of on-chip memory. The bigger and more powerful a host becomes, the better this ratio gets in favour of the TTSoC architecture. Taking this expectation into account, the TTSoC architecture will produce an overhead of resource usage of target technology in the magnitude of 10 % and below on FPGA technology. Thus, to provide a given system design with the ability to take part in the TTSoC architecture, the costs (with respect to resource usage on FPGAs) are manageable.

If it comes to absolute number in "real money"' about the costs of the TTSoC architecture, we refer to the estimations of number of basic cells in section 9.1.2. According to the 2007 semiconductor roadmap [Wil07b], in the year 2008 a basic cell (i.e., "gate") including SRAM cells entailes costs of $9,4$ $\mu cent$ in production on CMOS technology. By the time of the year 2013, the costs will have dropped to 1.7 $\mu cent$. So, the TISS of the current implementation with its $\approx 56k$ gates would cost about $0,005$ \$, while a Fragment Switch can be estimated up to $0,0005$ \$. If we consider several TISSs and include Fragment Switches like in the design example of section 9.3, based on the estimation above, a complete TSS would cost about $0,05$ \$. Note that within the next 5 years, this value can be expected to drop by a factor > 5, so that a TSS produces costs of about $0,01$ \$ then.

To conclude, the low overhead of resource usage on a given target technology is more than compensated by all the mighty features of the TTSoC architecture (e.g., encapsulation and error containment, periodicity of application-level messages, complexity reduction, flexibility by means of integrated resource management with on-the-fly reconfiguration, predictability and determinism). Consequently, we have shown that the TTSoC architecture is feasible, indeed, and that its monetary costs are so low that it is promising to equip system designs in the field of mixed-criticality distributed embedded (hard) real-time systems with the TTSoC architecture in order to benefit from its features and architectural concepts.

10.5 Outlook

Even though, the current implementation embodies a working prototype, the TTSoC architecture is still in its infancy. During the implementation we have revealed much potential for future work to be undertaken. This future work ranges from minor implementation optimizations up to integration of mighty high-level services.

Optimizations There is some margin left in the layout of the Time-Triggered Communication Schedule and Burst Configuration Memory. If we move most fields

in the entry of the Time-Triggered Communication Schedule into the Burst Configuration Memory so that only `Next`, `Instant`, and `BurstRef` remain, the size of the multiplexers in the Burst Dispatcher is considerably reduced. We estimate that this optimization would save about 40 % of logic elements and registers in the Burst Dispatcher. However, this measure imposes a penalty on the latencies Δ^+ and Δ^-. The processing of the Port Manager has to be stalled until all information, which (usually) is simultaneously available at the Burst Dispatcher's interface, has been fetched from the Burst Configuration Memory after the indication of the burst by means of `InfoValid` = 1. This is the reason, why this optimization has not been included in the current implementation. If we have had such a layout of entries in the respective memories, the latencies would have become too critical considering $\mathcal{R}_f$ on the MPSoC Development Kit, which is the main prototype hardware of the current implementation. Therefore, we have decided to postpone this optimization until we replace the MPSoC Development Kit with more performant hardware, which provides better values of $\mathcal{R}_f$ so that increased latencies can be neglected.

Dynamic resource management Another example of future work is the dynamic resource management. Although dynamic resource management is fully specified and parts like the on-the-fly reconfiguration are completely implemented, the current version of the RMA's and TNA's embedded software just provides static resource management. The features currently included of the predefined primary modes and degradation levels are sufficient to validate the core services, but they do not harness the full mightiness of integrated resource management in the TTSoC architecture. For example, there is no implementation of a distributed consensus algorithm between micro components and RMA, which negotiates resource allocations. The TNA has the ability to selectively power down Fragment Switches and disable phase comparators and period controllers in the Burst Dispatcher, but there is currently no algorithm implemented, which establishes a reasonable and comprehensive power management. So, the dynamic resource management lies idle, and the implementation of such mighty features is subject of further development.

Dynamic macro tick scaling We conceive *dynamic macro tick scaling* to be implemented in future revisions of the TTSoC architecture. By now, the macro tick is hard-wired at design time, and their is no mechanism to increase or decrease the macro tick during live operation. In contrast to the current implementation, dynamic macro tick scaling introduces a supplemental feature to the TTSoC architecture, which enables to alter the macro tick at run-time.

There are two modification necessary to achieve that dynamic macro tick scaling. Firstly, in the clock component of each TISS we have to replace the constant bit pattern, which is added to the counter vector of the time format of the global time base on each rising edge of the macro tick, with a register. Then, this "increment" register contains a variable bit pattern, which corresponds to the increment of the counter vector on each rising edge. For instance, the default bit pattern is all bits 0

except the bit at index 10, which denotes the macro tick of 2^{-20} $sec \Leftrightarrow 1,04$ MHz. With dynamic macro tick scaling some authorized entity, e.g., the TNA, can change the bit pattern in the "increment" register to some other value, say, all bits 0 except the bit at index 7. Then, on each increment of the counter vector we measure the time elapsed according to a macro tick of 2^{-23} $sec \Leftrightarrow 8,39$ MHz.

Secondly, the frequency associated with the macro tick must be adapted at runtime, so that the counter vector is incremented at intervals that match the setting of the "increment" register. For this purpose, we consider a single "macro tick generator". Unlike the current implementation, which obtains the macro tick from an external clock source, the "macro tick generator" produces the proper frequency from a reference clock. The same authorized entity, e.g., the TNA , which is also in charge of maintaining the "increment" registers in each TISS, configures that "macro tick generator" internally for the system in order to match the produced macro tick frequency with the settings of the "increment" registers.

Dynamic macro tick scaling adds another degree of flexibility to the periodic control system in the TTSoC architecture. We can combine dynamic macro tick scaling with the integrated resource management to change the macro tick dynamically with changing resource demands of the system. This contributes to a better energy efficiency, if the macro tick reflects the timeliness of the current primary modes (and degradation levels) of the application subsystems. Moreover, dynamic macro tick scaling facilitates the generality of a given instance of the TTSoC architecture. For example, it is possible to deploy a given instance of the TTSoC architecture in large quantities without regarding details of the intended target application. Anyway, dynamic macro tick scaling supports the adaption of the overall system to the actual requirements of the target domain in the field. Thus, the costs of designing instances of the TTSoC architecture are reduced, as we re-use a given design several times.

Transaction-level debugging The TTSoC architecture currently includes the concept of diagnosis. This feature resides at architectural level. However, system integrators, who build applications upon the TTSoC architecture, might be grateful, if they had a mechanism to visualize communication activities below the architectural scope. That is, "transaction-level" in the TTSoC architecture means to transportation of fragments in bursts. For instance, it would be useful to monitor the traffic on a given interconnect or TTNoC interface for the purpose of debugging. Therefore, transaction-level debugging contributes to manageability of the TTSoC architecture during development, and is worth to be considering as "feature request" for the TTSoC architecture.

Robustness Finally, we could increase the robustness of the TTSoC architecture against environmental distortions. In fact, the TTSoC architecture is designed to be composable and to tolerate faulty micro components due to error containment and encapsulation. However, these architectural concepts do not consider the unpredictability of operational conditions that originate from the real physical world.

For instance, cosmic radiation particles might cause bit flips on signals of the data bus of a lane in the TTNoC. If there is routing information transported over that lane, the bit flip causes a misinterpretation of a switching opcode so that a Fragment Switch decides to forward flits to the wrong interconnects. As a consequence, the route of an encapsulated communication channel is not set up properly. By design, we assume that the TSS is free of failures. However, if failures due to unforeseen external stimuli occur, the TTSoC architecture does not possess the ability to cope with unspecified error states. A possible solution of this problem, which establishes robustness against radiation induced bit flips, is a signal coding with multiple rails. The feature of robustness is necessary to harden the TTSoC architecture against environmental distortions. So, it is possible to recover from erroneous states that have not been covered in the fault hypothesis of the TTSoC architecture.

List of Acronyms

AHB	...	Advanced High-Performance Bus
ALG	...	Asynchronous Latency Guarantee
AMBA	...	Advanced Microcontroller Bus Architecture
APB	...	Advanced Peripheral Bus
BE	...	Best Effort
DAS	...	Distributed Application Subsystem
DMA	...	Direct Memory Access
DSM	...	Deep Submicron
DTL	...	Device Transaction Level
DU	...	Diagnostic Unit
GALS	...	Globally-Asynchronous Locally Synchronous
GB	...	Guaranteed Bandwidth
GS	...	Guaranteed Service
GT	...	Guaranteed Throughput
IP	...	Intellectual Property
LC	...	Looped Container
LUT	...	Look-Up Table
NA	...	Network Adapter
NBW	...	Non-Blocking Write
NI	...	Network Interface
NoC	...	Network-on-Chip
OCP	...	Open Core Protocol
PCA	...	Proximity Congestion Awareness
QoS	...	Quality of Service
RMA	...	Resource Management Authority
RNI	...	Resource Network Interface

SoC ... System-on-Chip

TDMA ... Time Division Multiple Access
TDN ... Temporally Disjoint Network
TISS ... Trusted Interface Subsystem
TMR ... Triple Modular Redundancy
TNA ... Trusted Network Authority
TSS ... Trusted Subsystem
TTA ... Time-Triggered Architecture
TTE ... Time-Triggered Ethernet
TTNoC ... Time-Triggered Network-on-Chip
TTSoC ... Time-Triggered System-on-Chip

UNI ... Uniform Network Interface

VC ... Virtual Circuit
Vc ... Virtual Channel

Bibliography

[AC03] TTTech Computertechnik AG and VERTEL Corp. Smart Transducers Interface. Specification formal/03-01-01, Object Management Group (OMG), January 2003. Available at `http://www.omg.org/technology/documents/formal/smarttrans.htm`.

[ARM99] ARM Limited. *AMBA 2 AHB Specification*. ARM Limited, Cambridge, England, UK, 1.0 edition, 1999. Available at `http://www.arm.com/products/solutions/amba2overview.html`.

[ARM01] ARM Limited. *AMBA 2 Multi-Layer AHB Specification*. ARM Limited, Cambridge, England, UK, 1.0 edition, 2001. Available at `http://www.arm.com/products/solutions/amba2overview.html`.

[ARM04] ARM Limited. *AMBA 3 AXI Specification*. ARM Limited, Cambridge, England, UK, 1.0 edition, 2004. Available at `http://www.arm.com/products/solutions/AMBA3AXI.html`.

[Ass05] OCP International Partnership Association. *Open Core Protocol Specification 2.1*, 2005. Available at `http://www.ocpip.org`.

[BB01] Ian Broster and Alan Burns. The Babbling Idiot in Event-triggered Real-time Systems. In *Proceedings of the Work-In-Progress Session, 22nd IEEE Real-Time Systems Symposium (RTSS), YCS 337*, pages 25 – 28. Department of Computer Science, University of York, 2001.

[BB04] Davide Bertozzi and Luca Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine*, 4(2):18 – 31, April/June 2004.

[Bje05] Tobias Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, Kgs. Lyngby, Denkmark, 2005.

[BM02] Luca Benini and Giovanni De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 38(1):70 – 78, January 2002.

[BM06] Tobias Bjerregaard and Shankar Mahadevan. A Survey of Research and Practices of Network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1), March 2006. Article no. 1.

[BMOS05] Tobias Bjerregaard, Shankar Mahadevan, Rasmus Grøndahl Olsen, and Jens Sparsø. An OCP Compliant Network Adapter for GALS-based SoC Design Using the MANGO Network-on-Chip. In *Proceedings of the International Symposium on System-on-Chip*, pages 171 – 174, November 2005.

[Bos91] Robert Bosch Gmbh, Stuttgart, Germany. *CAN Specification*, 2.0 edition, 1991. Available at `http://www.can-cia.de/index.php?id=164`.

[BS04] Tobias Bjerregaard and Jens Sparsø. Virtual channel designs for guaranteeing bandwidth in asynchronous network-on-chip. In *Proceedings of the Norchip Conference*, pages 269 – 272, November 2004.

[BS05a] Tobias Bjerregaard and Jens Sparsø. A Router Architecture for Connection-Oriented Service Guarantees in the MANGO Clockless Network-on-Chip. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1226 – 1231, March 2005.

[BS05b] Tobias Bjerregaard and Jens Sparsø. A Scheduling Discipline for Latency and Bandwidth Guarantees in Asynchronous Network-on-Chip. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 34 – 43, March 2005.

[BS06a] Tobias Bjerregaard and Jens Sparsø. Implementation of guaranteed services in the MANGO clockless network-on-chip. *IEE Proceedings in Computers and digital techniques*, 153(4):217 – 229, July 2006.

[BS06b] Tobias Bjerregaard and Jens Sparsø. Packetizing OCP Transactions in the MANGO Network-on-Chip. In *Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 657 – 664, August 2006.

[Car96] B. Carpenter. Architectural Principles of the Internet. Request for Comments RFC 1958, Network Working Group, Internet Engineering Task Force (IETF), June 1996. Available at `http://www.ietf.org/rfc/rfc1958.txt`.

[Cor08] Altera Corporation. *Avalon Interface Specification*, chapter Avalon Memory-Mapped Interfaces. Altera Corporation, San Jose, CA, USA, 1.1 edition, October 2008. Available at `http://www.altera.com/literature/manual/mnl_avalon_spec.pdf`.

[CSG98] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Series in Computer Architecture and Design. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1st edition, August 1998.

[Dal92] William Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 36(8):194 – 205, 1992.

[DT03] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.

[FCS01] Paul Feltovich, Richard Coulson, and Rand Spiro. Learners' Misunderstanding of Important and Difficult Concepts. *Smart Machines in Education*, pages 354 – 380, 2001. AAAI Press.

[FR92] Uriel Feige and Prabhakar Raghavan. Exact analysis of hot-potato routing. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 553 – 562, October 1992.

[GDG+05] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago González Pestana, Andrei Rădulescu, and Edwin Rijpkema. A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1182 – 1187, Washington, DC, USA, March 2005. IEEE Computer Society.

[GDR05] Kees Goossens, John Dielissen, and Andrei Rădulescu. The Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414 – 421, Sept./Oct. 2005.

[Gel01] Patrick P. Gelsinger. Microprocessors for the New Millenium, Challenges, Opportunities, and New Frontiers. In *Proceedings of the Solid State Circuit Conference (ISSCC)*, pages 22 – 25. IEEE Press, May 2001.

[GG00] Pierre Guerrier and Alain Greiner. A Generic Architecture for On-Chip Packet-Switched Interconnections. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 250 – 256, March 2000.

[GIJ+03] Marie-Claude Gaudel, Valérie Issarny, Cliff Jones, Hermann Kopetz, Eric Marsden, Nick Moffat, Michael Paulitsch, David Powell, Brian Randell, Alexander Romanovsky, Robert Stroud, and François Taiani. Final Version of DSoS Conceptual Model. Technical Report CS-TR-782, University of Newcastle, Newcastle, UK, April 2003. Available at `http://www.cs.ncl.ac.uk/research/pubs/trs/papers/782.pdf`.

[Gro05] IEEE Ethernet Working Group. Carrier sense multiple access with collision detection (CSMA/CD) access method and pyhsical layer specifications. Standard IEEE 802.3-2005, IEEE, December 2005. Available at `http://grouper.ieee.org/groups/802/3/index.html`.

[GvMPW02] Kees Goossens, Jef van Meerbergen, Ad Peeters, and Paul Wielage. Networks on Silicon: Combining Best-Effort and Guaranteed Services. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 423 – 425, March 2002.

[HCG07] Andreas Hansson, Martijn Coenen, and Kees Goossens. Undisrupted Quality-Of-Service during Reconfiguration of Multiple Applications in Networks on Chip. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 954–959, April 2007.

[HG07] Andreas Hansson and Kees Goossens. Trade-offs in the Configuration of a Network on Chip for Multiple Use-Cases. In *Proceedings of the International Symposium on Networks on Chip (NOCS)*, pages 233 – 242, May 2007.

[HGR07] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A Unified Approach to Mapping and Routing on a Network on Chip for both Best-Effort and Guaranteed Service Traffic. *VLSI Design*, 2007, May 2007. Article ID 68432, 16 pages.

[Hoe04] Carl Hoefer. Causality and Determinism: Tension, or Outright Conflict? *Revista de filosofía*, 29(2):99 – 115, November 2004.

[HP06] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 4th edition, 2006.

[Hub08] Bernhard Huber. *Resource Management in an Integrated Time-Triggered Architecture*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2008.

[KAGS05] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The Time-Triggered Ethernet (TTE) Design. In *Proceedings of 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 22 – 33, Seattle, Washington, USA, May 2005.

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.

[KFA+07] Michael Keating, David Flynn, Rob Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual For System-on-Chip Design*.

Series on Integrated Circuits and Systems. Springer Verlag, New York, NY, USA, 1st edition, July 2007.

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP – A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14 – 23, January 1994.

[KJS+02] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johnny Öberg, Kari Tiensyrjä, and Ahmed Hemani. A Network on Chip Architecture and Design Methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 105 – 112, April 2002.

[KO87] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933 – 940, 1987.

[KO02] Hermann Kopetz and Roman Obermaisser. Temporal Composability. *Computing & Control Engineering Journal*, 13(4):156 – 162, August 2002.

[Kop92] Hermann Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 460 – 467, Yokohama, Japan, June 1992.

[Kop97] Hermann Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Bosten, MA, USA, 1st edition, April 1997.

[Kop06] Hermann Kopetz. Pulsed Data Streams. In *IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES)*, pages 105 – 124, Braga, Portugal, October 2006. Springer.

[Kop08a] Hermann Kopetz. The Complexity Challenge in Embedded System Design. In *The 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*, pages 3 – 12, Orlando, Florida, USA, May 2008.

[Kop08b] Hermann Kopetz. The Rationale for Time-Triggered Ethernet. In *The 29th IEEE Real-time System Symposium (RTSS)*, Barcelona, Spain, December 2008.

[KR07] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 203 – 215. IEEE Circuits and Systems Society, February 2007.

[Lei85] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892 – 901, October 1985.

[LJ06] Guang Liang and Axel Jantsch. Adaptive Power Management for the On-Chip Communication Network. In *Proceedings of 9th EUROMICRO Conference on Digital System Design*, pages 649 – 656, August 2006.

[Lyn97] Nancy A. Lynch. *Distributed Algorithms*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 1st edition, April 1997.

[MB06] Giovanni De Micheli and Luca Benini. *Networks on Chips: Technology and Tools*. Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann Publishers, August 2006.

[MNT02] Mikael Millberg, Erland Nilsson, and Rikard Thid. The Nostrum Protocol Stack and Suggested Services Provided by the Nostrum Backbone. Technical Report TRITA-IMIT-LECSR02:01, LECS, IMIT, KTH, Stockholm, Sweden, November 2002. Available at `http://www.imit.kth.se/info/FOFU/NOC/docs1/Reports-2002/millberg.pdf`.

[MNT+04] Mikael Millberg, Erland Nilsson, Rikard Thid, Shashi Kumar, and Axel Jantsch. The Nostrum Backbone – a Communication Protocol Stack for Networks on Chip. In *Proceedings of the 17th International Conference on VLSI Design*, pages 693 – 696, January 2004.

[MNTJ04] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed Bandwidth using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 890 – 895, February 2004.

[MT89] Mihajlo D. Mesarovic and Yasuhiko Takahara. *Abstract system theory*. Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany, January 1989.

[MVK+99] Jens Muttersbach, Thomas Villinger, Hubert Kaeslin, Norbert Felber, and Wolfgang Fichtner. Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-CHIP Systems. In *Proceedings of the 12th Annual IEEE International ASIC/SOC Conference*, pages 317 – 321, September 1999.

[NMÖJ03] Erland Nilsson, Mikael Millberg, Johnny Öberg, and Axel Jantsch. Load distribution with the Proximity Congestion Awareness in a Network on Chip. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1126 – 1127, March 2003.

[OFSH08] Roman Obermaisser, Bernhard Frömel, Christian El Salloum, and Bernhard Huber. Integrating Safety and Multimedia Subsystems on a Time-Triggered System-on-a-Chip. In *Proceedings of the 6th IEEE International Conference on Industrial Informatics (INDIN)*, pages 270 – 275, Daejeon, Korea, July 2008.

[OKSH07] Roman Obermaisser, Hermann Kopetz, Christian El Salloum, and Bernhard Huber. Error Containment in the Time-Triggered System-On-a-Chip Architecture. In *Proceedings of the International Embedded Systems Symposium (IESS)*, San Diego, CA, USA, June 2007.

[OPHS06] Roman Obermaisser, Philipp Peti, Bernhard Huber, and Christian El Salloum. DECOS: An Integrated Time-Triggered Architecture. *E & I Journal*, 3:83 – 95, March 2006. (Journal of the Austrian professional institution for electrical and information engineering).

[Owe04] John Owens. GPUs: Engines for Future High-Performance Computing. Technical Report A551334, Lincoln Labs, University of California, Davis, CA, USA, September 2004.

[Phi02] Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification*. Philips Semiconductors, Eindhoven, The Netherlands, 2.2 edition, July 2002. Available at `http://www.nxp.com`.

[PJ06] Sandro Penolazzi and Axel Jantsch. A High Level Power Model for the Nostrum NoC. In *Proceedings of 9th EUROMICRO Conference on Digital System Design*, pages 673 – 676, August 2006.

[PK08] Christian Paukovits and Hermann Kopetz. Concepts of Switching in the Time-Triggered Network-on-Chip. In *The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 120 – 129, Kaohsiung, Taiwan, August 2008.

[PMH98] B. Pauli, A. Meyna, and P. Heitmann. Reliability of Electronic Components and Control Units in Motor Vehicle Applications. In *VDI Bericht 1415, Electronic Systems for Vehicles*, pages 1009 – 1024. Verein Deutscher Ingenieure, 1998.

[Pol93] Stefan Poledna. Replica Determinism in Distributed Real-Time Systems: A Brief Survey. Research Report 6/1993, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 1993.

[Pol94] Stefan Poledna. Replica Determinism in Distributed Real-Time Systems: A Brief Survey. *Real-Time Systems*, 6:289 – 316, 1994.

[POS+07] Harald Paulitsch, Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. A Diagnostic Unit for the Time-Triggered System-on-a-Chip Architecture. *DATE07 Workshop on Di-*

agnostic Services in Network-on-Chips – Test, Debug, and On-Line Monitoring, April 2007.

[RDG+04] Andrei Rădulescu, John Dielissen, Kees Goossens, Edwin Rijpkema, and Paul Wielage. An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, volume 2, pages 878 – 883, Washington, DC, USA, February 2004. IEEE Computer Society.

[RDP+05] Andrei Rădulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(1):4 – 17, January 2005.

[RGR+08] Edwin Rijpkema, Kees Goossens, Andrei Rădulescu, John Dielissen, Jef van Meerbergen, Paul Wielage, and Erwin Waterlander. Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip. In Rudy Lauwereins and Jan Madsen, editors, *Design Automation, and Test in Europe DATE. The Most Influential Papers of 10 Years*, Circuits & Systems, chapter 2 (Networks on Chip). Springer, January 2008.

[Ros90] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI.* Prentice Hall, January 1990.

[RSG03] Vijay Raghunathan, Mani B. Srivastava, and Rajesh K. Gupta. A survey of techniques for energy efficient on-chip communication. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 900 – 905, June 2003.

[SAC+05] Stergios Stergiou, Federico Angiolini, Salvatore Carta, Luigi Raffo, Davide Bertozzi, and Giovanni De Micheli. ×pipes lite: A Synthsis Oriented Design Library For Networks on Chips. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1188 – 1193, March 2005.

[Sal08] Christian El Salloum. *Interface Design in the Time-Triggered System-on-Chip Architecture.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2008.

[Sch93] Werner Schütz. *The Testability of Distributed Real-Time Systems*, volume 245 of *The Springer International Series in Engineering and Computer Science.* Springer Verlag, 1st edition, August 1993.

[Sif05] Joseph Sifakis. A Framework for Component-based Construction. In *Proceedings of 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM05)*, pages 293 – 300, September 2005.

[SOH+07] Christian El Salloum, Roman Obermaisser, Bernhard Huber, Harald Paulitsch, and Hermann Kopetz. A Time-Triggered System-on-a-Chip Architecture with integrated support for diagnosis. *DATE07 Workshop on Diagnostic Services in Network-on-Chips – Test, Debug, and On-Line Monitoring*, April 2007.

[SOHK08] Christian El Salloum, Roman Obermaisser, Bernhard Huber, and Hermann Kopetz. The Time-Triggered System-on-a-Chip Architecture. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, Cambridge, UK, June 2008.

[SWH95] Neeraj Suri, Chris J. Walter, and Michelle M. Hugue. *Advances In Ultra-Dependable Distributed Systems*, chapter 1. IEEE Computer Society Press, Los Alamitos, CA, USA, January 1995.

[TS06] Andrew Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition, 2006.

[TTT06] TTTech Computer Technik AG. *PLCB1 bus specification*. TTTech Computer Technik AG, Vienna, Austria, 1.0.02 edition, May 2006. Available at `http://www.tttech.com`.

[UK03] Osman S. Unsal and Israel Koren. System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, 91(7):1055 – 1069, 2003.

[WG02] Paul Wielage and Kees Goossens. Networks on silicon: blessing or nightmare? In *Proceedings of the Euromicro Symposium on Digital System Design*, pages 196 – 200, September 2002.

[Wil05] Linda Wilson. International Technology Roadmap for Semiconductors (ITRS) — Executive Summary. Technical report, Semiconductor Industry Association, 2005. Available at `http://www.itrs.net/Links/2005ITRS/Home2005.htm`.

[Wil07a] Linda Wilson. International Technology Roadmap for Semiconductors (ITRS) — Design. Technical report, Semiconductor Industry Association, 2007. Available at `http://www.itrs.net/Links/2007ITRS/Home2007.htm`.

[Wil07b] Linda Wilson. International Technology Roadmap for Semiconductors (ITRS) — Executive Summary. Technical report, Semiconductor Industry Association, 2007. Available at `http://www.itrs.net/Links/2007ITRS/Home2007.htm`.

[Wol04] Wayne Wolf. The Future of Multiprocessors Systems on Chips. In *Proceedings of the 41st annual conference on Design Automation*, pages 681 – 685, San Francisco, CA, USA, 2004. IEEE Press.

Index

Curriculum Vitae

Christian Peter Paukovits

June 15^{th}, 1982	Born in Oberwart (Austria)
September 1988 – June 1992	Primary School in Stadtschlaining (Austria)
September 1992 – June 1996	Secondary School in Stadtschlaining (Austria)
September 1996 – June 2001	Secondary Technical School for Information Technology and Management in Pinkafeld (Austria)
July 2001 – February 2002	Military Service in Baden (Austria)
March 2002 – September 2004	Bachelor Studies of Computer Science at the Vienna University of Technology with distinction
October 2004 – September 2006	Master Studies of Computer Science at the Vienna University of Technology with distinction
since October 2006	PhD Studies at the Vienna University of Technology
since January 2007	Research Assistant at the Institute of Computer Engineering, Vienna University of Technology