DISSERTATION

Interface Design in 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

eingereicht an der Technischen Universität Wien,
Fakultät für Technische Naturwissenschaften und Informatik

von

Christian El Salloum
Matr.-Nr. 9625650
Laimgrubengasse 12A/12, A-1060 Wien

Wien, im Dezember 2007 .....

Interface Design in the Time-Triggered System-on-Chip Architecture

The *Time-Triggered System-on-a-Chip* (TTSoC) architecture provides an integrated execution environment for the component-based development of many different types of embedded applications (e.g., automotive, avionics, consumer electronics). At the core of this architecture is a time-triggered *Network-on-a-Chip* (NoC) for the predictable interconnection of IP cores.

This thesis contributes to the TTSoC architecture by designing a *Uniform Network Interface* (UNI) that supports the integration of multiple heterogeneous IP cores—belonging to different criticality-classes and application domains—into a single SoC. The UNI is implemented by a dedicated hardware component called the *Trusted Interface Subsystem* (TISS), which is replicated for each IP core. The TISS controls the IP core's access to the time-triggered NoC and provides encapsulation mechanisms that prevent any unintended interference between IP cores, which is a major requirement for the integration of mixed-criticality subsystems.

Exploiting the inherent fault isolation and determinism of the architecture, we introduce a novel concept for fault-tolerance based on the replication of entire IP cores organized in *Triple Modular Redundancy* (TMR) configurations. With respect to TMR we have investigated two different approaches. While *on-chip TMR* realizes the replicas in the same SoC to increase the reliability of services residing on a single chip, *off-chip TMR* instantiates the replicas on different SoCs interconnected by a fault-tolerant off-chip network, as it is required for ultra-dependable systems.

Complementing the architectural framework, we have introduced a novel naming scheme tailored to the unique challenges of large embedded systems based on multi-processor SoCs. The naming scheme supports independent development of application subsystems by providing a dedicated, independent, and domain-specific namespace for each application subsystem and facilitates dynamic resource management by decoupling the logical and the physical system structure.

Interface-Design in der Time-triggered SoC Architektur

Die *Time-Triggered System-on-a-Chip* (TTSoC) Architektur unterstützt die komponentenbasierte Entwicklung von eingebetteten Systemen in den unterschiedlichsten Bereichen wie zum Beispiel in der Automobilindustrie, der Luft- und Raumfahrt oder der Unterhaltungselektronik. Den Kern der Architektur bildet ein zeitgesteuertes *Network-on-a-Chip* (NoC), welches für eine deterministische Kommunikation zwischen den IP-Cores eines SoCs sorgt.

Im Rahmen dieser Arbeit wurde für die TTSoC Architektur ein *uniformes Netzwerkinterface* (UNI) definiert, welches die Integration von heterogenen IP-Cores unterschiedlicher Kritikalitätsklassen in einem SoC ermöglicht. Das UNI wird durch eine dedizierte Hardwarekomponente, das *Trusted Interface Subsystem* (TISS), implementiert. Das TISS ist in jedem IP-Core repliziert, kontrolliert die Zugriffe des IP-Cores auf das zeitgesteuerte NoC und verhindert somit jegliche Form von unbeabsichtigter Interaktion zwischen den IP-Cores.

Weiters beschreibt diese Arbeit ein neuartiges Fehlertoleranzkonzept, welches auf der inhärenten Fehlerisolation und dem Determinismus der Architektur aufbaut. Es basiert auf der Replikation von kompletten IP-Cores und deren Anordnung in einer *Triple Modular Redundancy* (TMR) Konfiguration. Hierbei werden zwei unterschiedliche Ansätze beschrieben. Bei "*on-chip TMR*" befinden sich die replizierten Komponenten auf demselben SoC, um die Zuverlässigkeit eines einzelnen Chips zu erhöhen. "*Off-chip TMR*" unterstützt die Konstruktion von Systemen höchster Kritikalitätsklasse, indem die replizierten IP-Cores auf unterschiedlichen SoC instanziiert werden, die mittels eines fehlertoleranten Netzwerks verbunden sind.

Weiters wird ein neuartiges Namensschema eingeführt, welches auf die speziellen Herausforderungen von SoC-basierenden eingebetteten Systemen abgestimmt ist. Das Namensschema unterstützt die unabhängige Entwicklung von Subsystemen, indem für jedes Subsystem ein dedizierter und domänenspezifischer Namensraum vergeben wird. Weiters zeichnet sich das Namensschema durch die Entkopplung von logischer und physikalischer Systemstruktur aus. Diese Eigenschaft ist für das dynamische Management von Systemressourcen und die damit verbundene dynamische Rekonfiguration von großem Vorteil.

Danksagung

Diese Arbeit entstand im Rahmen meiner Forschungs- und Lehrtätigkeit am Institut für Technische Informatik, Abteilung für Echtzeitsysteme, an der Technischen Universität Wien.

Besonders danken möchte ich dem Betreuer meiner Dissertation, Prof. Dr. Hermann Kopetz, der mir die Forschungstätigkeit am Institut für Technische Informatik ermöglicht hat. Er unterstützte meine Arbeit durch wertvolle Anregungen und stimulierende Diskussionen und prägte so meinen wissenschaftlichen Werdegang.

Weiters danke ich Prof. Dr. Wolfgang Kastner, dem Zweitbegutachter dieser Dissertation, für das Interesse an meiner Arbeit und für die konstruktiven Verbesserungsvorschläge.

Auch meinen Arbeitskollegen möchte ich meine Dankbarkeit zum Ausdruck bringen, für ihre Freundschaft, das angenehme Arbeitsklima und die fruchtbringende Zusammenarbeit. Die folgenden Freunde und Kollegen (in alphabetischer Reihenfolge) haben durch das sorgfältige Korrekturlesen von früheren Versionen dieser Arbeit wesentlich zur Qualität beigetragen: Wilfried Elmenreich, Bernhard Huber, Stefan Kral, Roman Obermaisser, Christian Paukovits, Harald Paulitsch und Armin Wasicek.

Besonders bedanken möchte ich mich bei meiner lieben Freundin Judith. Auch in Zeiten intensievster Arbeit konnte ich immer auf ihr Verständnis und ihre Unterstützung zählen.

Zu guter Letzt möchte ich vor allem meinen Eltern Renate und Habib für den emotionalen Rückhalt danken, auf den ich während meines bisherigen Lebens immer vertrauen konnte.

Contents

Chapter 1

Introduction

1.1 Motivation

During the past forty years, the semiconductor industry has made tremendous progress that has enabled the construction of chips approaching a billion of transistors on a single die. These spectacular improvements and the associated reduction of the cost per transistor have enabled new applications of embedded systems with ever increasing functionality. A representative example is the automotive industry, where a modern luxury car contains more than 70 *Electronic Control Units* (ECUs) and up to 10 millions lines of code [MGRG+04]. In-vehicle electronics is already the strongest innovation driver in the automotive industry and accounts for up to 35% of the total value of a car. Considering future applications, like steer-by-wire, we can expect this trend to continue.

Together with the increased functionality, the design complexity is also rising at an overwhelming rate. The *International Technology Roadmap for Semiconductors* (ITRS) considers design complexity and designer's productivity as key challenges on the way to giga-scale SoCs [ITR05]. This challenge can only be tackled by lifting the design process to higher levels of abstraction.

A major problem of traditional design approaches for SoCs is that the *Intellectual Property* (IP) cores of a processor are designed with bus-specific interfaces exposing the implementation-specific details of the interconnect directly to the core [BM06]. An example for this approach is AMBA AHB [ARM99] where the interfaces of the attached cores include signals that are connected to bus-internal components like the bus arbiter. Thus, the cores and the interconnect are tightly coupled.

When the level of abstraction with respect to system integration began to rise, it became necessary to reuse pre-designed cores, possibly provided by third party

suppliers, across multiple architectures with different on-chip communication infrastructures.

The *Time-Triggered System-on-a-Chip* (TTSoC) architecture [Kop05] is a novel system architecture that offers a component-based design methodology for managing the complexity of chips with billions of transistors. By thorough decoupling of the computational components from the communication infrastructure, the design of a computational component can abstract from the implementation of the interconnect, which facilitates the rapid development of multi-core SoCs by using pre-verified functional cores.

For this purpose, the time-triggered SoC architecture provides an architectural framework that supports the side-effect-free composition of component services, based solely on the interface specifications, to form larger systems-of-systems.

As a fundamental concept, we introduce the notion of a *micro component* which is a self-contained computational unit that provides its functionality over a *message-based* interface that is defined in the value domain, as well as, in the temporal domain. The clear separation of the processing within a micro component from the interactions among the micro components leads to a communication-centric model that is highly appropriate for many applications.

Micro components are interconnected through a predictable and deterministic time-triggered NoC, with dedicated time slots assigned to each micro component. The time-triggered NoC prevents any unintended and unwanted interference between micro components employing dedicated encapsulation mechanisms, which reduces system complexity because the behavior of interfering subsystems is more difficult to understand and to reason about than the behavior of cleanly encapsulated subsystems.

1.2 Contribution of this Thesis

A major contribution of this thesis is the design of an NoC interface that supports the integration of multiple heterogeneous IP cores—belonging to different criticality-classes and application domains—into a single SoC. We call this interface the *Uniform Network Interface* (UNI). It provides a set of *core platform services* that facilitate the development of distributed real-time applications and separate the application functionality from the underlying platform technology to reduce design complexity and to enable design reuse. This approach corresponds to the concept of platform-based design as described in [SV02].

The specification of the UNI is complemented by a novel naming scheme tailored to the unique challenges of large embedded systems based on multiprocessor SoCs. The naming scheme facilitates dynamic resource management

by establishing *location transparency* for logical system entities and by decoupling the logical and the physical system structure. Further benefits of the proposed solution are the coexistence of a *uniform namespace* spanning over multiple SoCs for system integration and multiple independent, *domain-specific namespaces* enabling autonomous subsystem development.

The UNI is implemented by a dedicated hardware component called the *Trusted Interface Subsystem* (TISS), which is replicated for each IP core. The TISS controls the IP core's access to the time-triggered NoC and provides encapsulation mechanisms that prevent any unintended interference between IP cores, which is a major requirement for the integration of mixed-criticality subsystems.

The core platform services of the TISS can be adapted, refined, and extended by a *frontend*, which is a hardware element that is stacked on the TISS and translates the UNI to the interface of the attached IP core. The ability to customize the UNI is of particular importance for the support of multiple heterogeneous application domains and for the integration of legacy components.

The configuration of the TISS, which includes, among other parameters, the time-triggered message schedule, can be dynamically adapted to support *integrated resource management*, which is an important cornerstone for the construction of power-aware and area-efficient systems. The design of the TISS guarantees that the temporally deterministic system behavior and the encapsulation properties are retained despite dynamically triggered reconfiguration activities.

In addition to the design of the UNI, the proposed naming scheme, and the TISS, this thesis contributes to the time-triggered SoC architecture by introducing a novel concept for fault-tolerance. The proposed concept exploits the inherent fault isolation and determinism of the architecture and is based on the replication of entire IP cores that are grouped into *Fault-Tolerant Units* (FTUs) according to the TMR scheme.

With respect to TMR we have investigated two different approaches: *on-chip TMR* and *off-chip TMR*. In the case of on-chip TMR, the replicated IP cores of an FTU are realized in the same SoC in order to increase the reliability of services residing on a single chip.

Off-chip TMR is employed for ultra-dependable systems, as a single chip cannot be expected to achieve the required reliability which is typically in the order of 10^{-9} failures per hour [Kop97]. In this case, the replicated IP cores of an FTU are located on different SoCs that are interconnected by a fault-tolerant off-chip network (e.g., TTP [KG94] or FlexRay [Fle05]).

1.3 Structure of this Thesis

Chapter 2 outlines the background and the motivation that led to the development of the integrated TTSoC architecture. The chapter starts with a comparison of the *federated* and the *integrated* architecture paradigm, then motivates the paradigm shift to multi-core architectures, and concludes with an elaboration of the role of complexity and complexity management in the construction of future embedded devices.

Chapter 3 gives an overview of state-of-the-art NoC interfaces and interconnects.

Chapter 4 describes the overall structure and the component model of the TTSoC architecture, and introduces the concept of *on-chip TMR* and *off-chip TMR* as the architecture's basic fault-tolerance mechanisms.

The naming scheme is presented in Chapter 5, while the TISS and the *core platform services* are described in Chapter 6.

Chapter 7 describes the design and implementation of a prototype that demonstrates the feasibility of the proposed architecture. It covers a detailed memory layout of the UNI, the design of the architectural elements, the description of the prototype hardware, and the implementation results.

Chapter 8 concludes the thesis by summarizing the main contributions.

Chapter 2

Background and Basic Concepts

This chapter outlines the background and the motivation that lead to the development of the integrated Time-Triggered System-on-a-Chip architecture. The chapter starts by comparing the *federated* to the *integrated* architecture paradigm, motivates the paradigm shift to multi-core architectures, and concludes with an elaboration of the role of complexity and complexity management in future embedded devices.

2.1 Federated vs. Integrated Architectures

One can distinguish two extreme classes of architecture paradigms, namely *federated* and *integrated* architectures [OPHES06]. According to the federated paradigm, each application subsystem is realized on a dedicated computer system, while in integrated architectures multiple application subsystems are integrated within a single, possibly physically distributed, computer system. Since both paradigms have specific advantages and disadvantages, most of the existing real-time systems are positioned between these two extremes.

Federated architectures have been usually preferred for ultra-dependable systems since the physical separation of application subsystems facilitates error containment, independent development, and complexity management. The major advantages of federated architectures are listed below.

Fault and Error Containment. Since each application subsystem is realized on a dedicated computer system, a physical fault in any hardware component can only affect a single application subsystem. Thus, fault containment is trivially achieved in federated architectures whereas in an integrated architecture a hardware fault in any of the system components

has a potential impact on every application subsystem that shares that component. Furthermore, the limited interactions between the different application subsystems in a federated system facilitate the establishment of error containment, i.e., the prevention of error propagation from one application subsystem to another one.

Independent Development and IP Protection. The federated paradigm enables independent development of application subsystems by different vendors since each application subsystem is nearly independent, and the interactions between application subsystems are limited to occur via gateways. Thus, the need for communication between the development teams of different application subsystems is kept on a minimum level. Furthermore, the use of heterogeneous technologies (e.g., processors, communication protocols and operating systems) for the different application subsystems poses no problem.

In a federated architecture, the vendor of an application subsystem does not have to reveal internals to the system integrator or to any other vendor due to the clear separation of the application subsystems. This property is relevant for the protection of the vendor's intellectual property.

Complexity Control. Each application subsystem has a given complexity which is inherent to the associated application (i.e., to the problem to be solved). When multiple application subsystems are integrated into a larger system, the complexity can significantly increase, if the architecture does not prevent unintended interference between the application subsystems. The federated approach prevents any unintended interference trivially by assigning a dedicated distributed computer system to each application subsystem, whereas an integrated architecture requires elaborate mechanisms for temporal and spatial partitioning.

Compared to federated architectures, integrated architectures are very attractive due to the potential cost savings achieved by avoiding the duplication of resources, the increased dependability due to the reduction of wires and connectors, and the flexible coordination of different application subsystems.

Hardware Cost Reduction. Integrated architectures enable the multiplexing of hardware resources (e.g., communication networks, sensors, and computational nodes) that are needed by multiple application subsystems. In contrast to this, the federated approach requires to duplicate such resources which is likely to result in an unacceptable overhead. The automotive industry is an example where the federated approach, is about to reach its limits. Modern luxury cars like the BMW 7 [Dei02] series

already contain up to 70 ECUs, which are interconnected via multiple different communication networks (e.g., CAN [Rob91], MOST [MC02]).

Dependability Improvements. In addition to the increase in resource efficiency, the reduction of the number of ECUs leads to systems with fewer connectors and wires. By analyzing field data from the automotive industry it was shown that more than 30% of electrical failures are caused by connector problems [SM98]. Thus, the reduction of the ECU count has a direct impact on the dependability of the overall system.

Furthermore, integrated architectures enable redundancy management in a very flexible way. In a federated architecture, each fault-tolerant application subsystem has its own spare components that it can use exclusively. An application subsystem that has already used up all its spare components and develops an additional fault, can fail even if there are other available spare components in the system (i.e., in other application subsystems). In an integrated architecture, spare components can be made universally available to multiple application subsystems.

Improved Coordination of Application Subsystems. Integrated architectures support the tactic coordination of tightly coupled application subsystems. An example where such a coordination is required is the passive safety mechanism (Pre-Safe) of the Mercedes S-Class [Bir03]. The Pre-Safe systems tightens the seat belts, aligns the seats in a safe position and closes the sun roof when a hazardous driving situation is detected. For this purpose the comfort subsystem uses and correlates the information of existing car dynamics sensors of other application subsystems.

An ideal future system architecture would *combine the complexity management advantages of the federated approach, but would realize the functional integration and hardware benefits of an integrated system* [Ham03]. The challenge is to devise an architecture that supports the integration of different application subsystems—possibly with mixed criticality levels—on a single distributed computer system, while retaining the fault isolation and complexity management properties of the federated architecture paradigm.

The DECOS project [OPHES06] addressed exactly this challenge and laid the foundation for the TTSoC architecture which is the topic of this thesis.

2.2 The Paradigm Shift to Multi-Core

A multi-core processor combines multiple independent cores into a single package. This section outlines the motivation for the transition to multi-core architectures and describes the consequences of that transition.

2.2.1 Power and Area Efficiency

Gordon E. Moore predicted that the total number of devices on a chip would double every generation (18–24 months). Known widely as *Moore's Law*, this prediction made the case up to now and is still a driving force for technological and social change. Figure 2.1 plots the growth of transistor counts for Intel processors over the last decades, starting with the first microprocessor—Intel's 4004—to the most recent microprocessors.

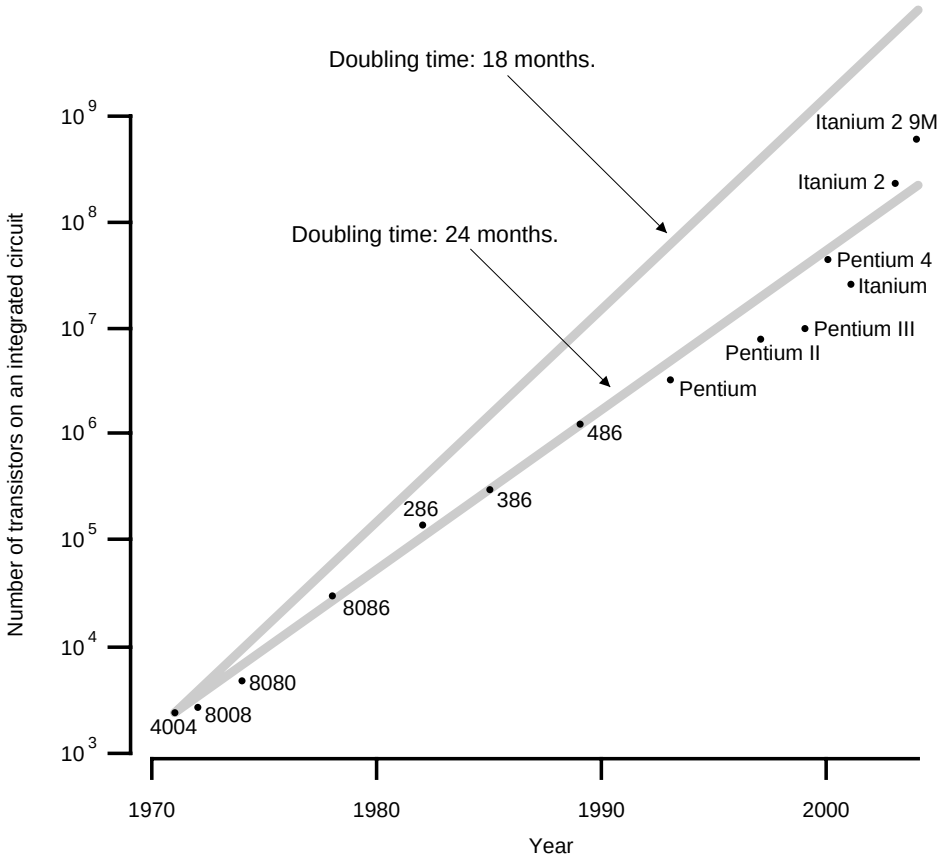

Figure 2.1: Growth of Transistor Counts for Intel Processors

While Moore's Law is still alive, and brings us an integration capacity of billions of transistors today, we have to consider ways to employ these transistors effectively in order to deliver the desired performance. For decades, the increasing number of transistors was used to push the performance of a *single* processing core by developing larger micro-architectures with a higher complexity. Examples are micro-architectures based on super-pipelined designs featuring speculative, super-scalar, and out-of-order execution.

The problem is that performance increases that are exclusively based on ad-

vances in micro-architecture are governed by *Pollack's Rule* [Pol99]. Pollack's Rule states that, in the same process technology, a leading micro-processor consumes twice the area and power over the previous generation microprocessor, compared with a performance increase by a factor of 1.4. In other words, doubling the number transistors in a single processor core results only in 40% additional performance, which means that ever increasing single cores yield diminishing performance in a power and area envelope. In contrast to the single core approach, a multi-core architecture has the potential to provide near linear performance improvement.

Another key challenge in keeping Moore's Law alive is integrated power management. The main issue is that the integration capacity is still increasing according to Moore's law, while the scaling of the supply voltage is slowing down. Thus, the power consumption of a fixed-sized die will be constantly increasing until reaching an unacceptable limit if appropriate counter measures are not taken. Multi-core architectures are very flexible with respect to power management due to the following reasons.

- To save power, each core can be individually turned off if its functionality is not required at the moment.
- Each core can run at its optimized clock frequency and supply voltage. This is especially useful for heterogeneous multi-core architectures.
- The computational load can be balanced among the processor cores in order to distribute the heat uniformly across the die.

2.2.2 Amdahl's Law

As mentioned above, multi-core systems deliver a higher potential compute throughput than single-core systems for the same die size and in the same power envelope. Nevertheless, the potential speedup is limited. *Amdahl's Law*, states that the theoretical maximum speedup achievable by parallelization is limited by the relative size of the non-parallelizable part (i.e., the serial part) of a program. Equation 2.1 shows how the maximum speedup (S) relates to the relative size of the parallelizable part (p), the relative size of the serial part ($s = 1 - p$), and the number of parallel cores (N).

$$S = \frac{s + p}{s + p/n} = \frac{1}{s + (1 - s)/N} \tag{2.1}$$

Figure 2.2 depicts the maximum speedup for different values of s and different numbers of parallel cores. The figure shows that even a small percentage of

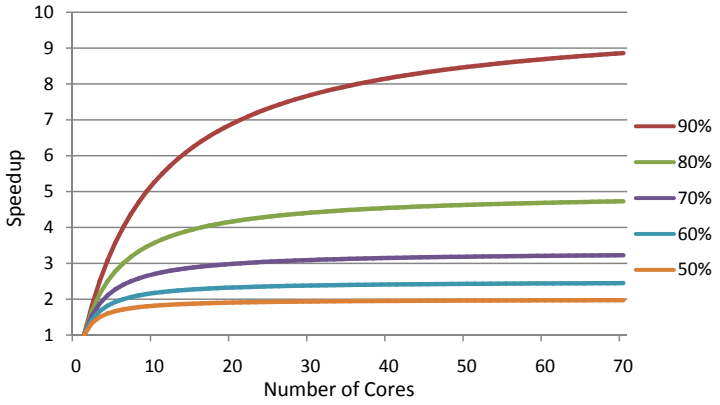

Figure 2.2: Amdahl's Law

non-parallelizable code in a program leads to a saturation of the achievable speedup at a small number of cores.

It must be considered that this limitation is only valid if one tries to parallelize a single application with a single continuous serial part across all the cores in the chip. In the domain of embedded systems, a device typically integrates multiple application subsystems that are inherently parallel. As an example consider the electronic control system in a modern luxury car executing tasks for the power train, the comfort electronics, or for the vehicle dynamics management system. In such a system the real challenge is not to parallelize algorithms solving *one* big problem as it is done for high-performance computing, but to integrate multiple tasks that are inherently parallel without inducing any additional complexity by the architecture (e.g., non-intended interference).

Furthermore, as embedded systems applications often process data flowing to and from *multiple* independent serial parts, instead of a single continuous serial part, a pipelined approach can naturally be employed for parallelization. For example, a multi-media application could utilize dedicated serial parts (e.g., one serial part per core) for decoding, application of different filters, and encoding.

2.2.3 The Memory Bandwidth Gap

The potential speedup of a multi-core processor can only be exploited if the bandwidth between the memory and processor scales together with the processor's performance. An example of a state-of-the-art multi-core processor is the *IBM Cell Broadband Engine* [Hof05] providing an aggregate memory bandwidth of 25.6 GB/s via two 32 bit channels (3.2 Gbit/s per pin).

We can expect that the bandwidth requirements of next generation processors will soon exceed several Tbit/s. Considering that a typical memory I/O circuit consumes about 25 mW/Gbps (or 25 W/Tbps) [Bor07], the traditional

memory subsystem approach is about to reach its limits. The main reason is that traditional buses are transmission lines with LRC effects that require complex and power-consuming signal processing techniques to attain a high data rate. If we were able to shorten the bus length to a few millimeters, the buses would behave more like lumped capacitors instead of transmission lines. Such I/O circuits would consume substantially lower power, in the order of 1-2 mW/Gbps [Bor07].

A potential solution for shortening the bus length, is a three dimensional integration approach, where a thinned memory die is placed between the processor and the package [VHR+07]. In this approach, shown in Figure 2.3, the signals between the I/O pins and the processor are routed through the memory die using silicon vias. The width of the bus can easily be increased to deliver the required bandwidth, since the bus signals do not have to be routed out of the package. At the same time, the signaling speed of the bus can be reduced to further optimize the power efficiency.

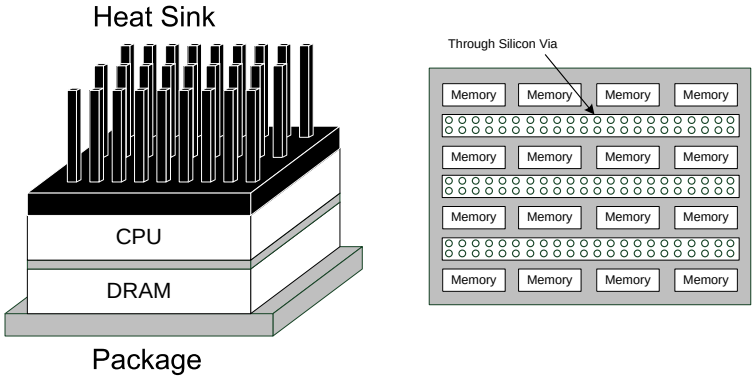

Figure 2.3: 3D Memory Stacking

In a multi-core architecture, the three-dimensional integration approach would allow each core to have its own memory with a dedicated memory bus that is not shared with the other cores. Such an architecture would significantly simplify arbitration and temporal predictability with respect to memory I/O. On the other hand, it must be considered that the traditional shared memory abstraction model will no more be feasible, and alternative inter-core communication mechanisms like message-passing will have to be employed.

2.3 The Role of Complexity

As pointed out in the previous section, Moore's Law is still valid and we can expect the trend of exponentially increasing transistor counts to continue. The

cost to realize a specific logic function in hardware and the energy required for its execution are constantly decreasing.

The tremendous progress of the semiconductor industry enables new applications of embedded systems with ever increasing functionality. A good example is the telecommunication industry. State-of-the-art mobile phones integrate GPS navigation, multi-media players, high-definition cameras, and office applications and provide a high degree of connectivity to other devices by supporting a multitude of communication standards like Bluetooth[1], IrDA[2], USB[3], and Wi-Fi[4] (WLAN).

We have to be aware that the astonishing leap of each generation of embedded system devices does not come for free. Together with the increased functionality, the design complexity is also increasing at an overwhelming rate. The International Technology Roadmap for Semiconductors [ITR05] considers design complexity and designer's productivity as key challenges on the way to giga-scale SoCs. Complexity is a key issue for the following reasons.

Development Cost. It is obvious that the complexity of a design has a direct impact on the *non-recurring engineering* (NRE) cost of a product. Devices with high complexity require a larger number of system designers with a higher level of education during a longer development phase.

Time-to-Market. While the system complexity is exponentially increasing, the time-to-market requirements for embedded system devices are becoming more and more stringent. Again the telecommunication industry is a representative example where the required time-to-market—from the idea to the final product—for the next generation of mobile phones is predicted to be shorter than three months. Since the market is highly competitive, missing the market window by only a few weeks results in a significant financial loss.

It is not possible to conquer the challenge of developing devices with ever increasing complexity within a shorter time interval by simply increasing the size of development teams. In his book *The Mythical Man-Month* [Bro95] Fred Brooks pointed out that the absolute throughput of a development team saturates or even decreases at a given team size due to the increased communication overhead.

The only solution to avoid exponentially increasing design cost and design time is to scale the overall productivity together with Moore's Law,

[1] http://www.bluetooth.com
[2] http://www.irda.org
[3] http://www.usb.org
[4] http://www.wi-fi.org

which means that the productivity with respect to design, verification, and testing has to double at least once per each technology generation. This challenge is commonly denoted as the *productivity challenge*.

Dependability. Complexity has also a significant impact on the dependability of a system. Right now we are in a situation where the number of bugs is growing faster than Moore's law. Verification and testing consumes already a major amount of the total development costs of a product.

Even worse, in spite of a lot of time and money spent on verification and testing, the uncontrolled complexity results in a large number of bugs that remain undetected. This leads to products that are failing at the customer and cause bad reputation of the vendor. A typical example is the automotive industry, where costumers are irritated that their expensive leading-edge luxury cars may turn out less reliable than cheaper series due to the complexity of the in-vehicle electronics.

To make the complexity manageable and to be able to concentrate on the relevant properties of a system, we have to elevate the design process to a higher level of abstraction. The following quote is taken from the International Technology Roadmap for Semiconductors: *"For continued improvement in designer's productivity, an emerging system-level of design, well above RTL (Register Transfer Level), is required. ... Higher levels of abstraction allow many forms of verification to be performed much earlier in the design process, reducing the time to market and lowering costs by discovering problems earlier." [ITR05, p.8]*

In many computer systems, the design complexity does not stem from the actual problem that has to be solved, but is introduced by the architecture used and the development method employed. One example is the integration of multiple distributed application subsystems via a single CAN network. Even if the complexity of each single application subsystem is low, the analysis of the integrated system is difficult due to the emergent complexity induced by the hidden interactions between the application subsystems. In a CAN network, the temporal behavior of one application subsystem with respect to communication is highly dependent on the communication activities of the other application subsystems. Furthermore, a transient hardware fault in the network or in the sender of a message can lead to masquerading effects by modifying the message identifier. Thus, error propagation can occur between two application subsystem even if they are totally independent on the logical level.

Other examples for architecture-induced complexity are multi-tasking operating systems in conjunction with modern pipelined microprocessors and complex caching strategies. In such architectures the temporal behavior is theoretically

deterministic, but cannot be captured in simple models with low cognitive complexity. *"We cannot determine the behavior of the system not because we cannot know "how it works", but because its complexity exceeds our computing or perceptual capacities." [Ger02]*

Concluding, we can summarize that complexity management and designer's productivity are key challenges for the successful development of future embedded devices. These challenges have to be considered already at the architecture level.

Chapter 3

NoC Interfaces and Interconnects

This chapter gives an overview of state-of-the-art NoC interfaces and interconnects. The network interface is the glue between an IP core and the on-chip interconnect. Traditionally, the IP cores of an embedded controller have been designed with bus-specific interfaces exposing the implementation-specific details of the interconnect directly to the core [BM06]. An example for this approach is AMBA AHB [ARM99] where the interfaces of the attached cores include signals that are connected to bus-internal components like the bus arbiter. Thus the cores and the interconnect are tightly coupled.

When the level of abstraction with respect to system integration began to rise, it became good practice to reuse pre-designed cores, possibly provided by third party suppliers, across multiple architectures with different on-chip communication infrastructures. This requirement caused a paradigm shift which is commonly referred to as *core centric* design. State-of-the-art protocols provide interface definitions that abstract from the implementation details of the interconnect. They can be either used with traditional buses or with modern on-chip networks. Examples are AMBA AXI [ARM04] which is the most recent specification of the AMBA protocol family or the *Open Core Protocol* (OCP) [OCP05].

3.1 AMBA

The *Advanced Microcontroller Bus Architecture* (AMBA) specification defines a standard for on-chip communication in high-performance embedded microcontrollers [ARM99][ARM01][ARM04]. It is released by ARM and represents the

current de facto standard for designing microcontrollers. This section gives an overview of the different versions of the AMBA specification, which represent a real-life example for the evolution of on-chip interconnects.

3.1.1 Advanced High-performance Bus - AHB

AMBA AHB [ARM99] is one of the first released specifications of the AMBA protocol family. It acts as a high-performance system *backbone* bus and was designed to interconnect high clock frequency system modules like processors or on-chip memories. Figure 3.1 depicts a typical AMBA AHB-based microcontroller which consists of the AHB backbone bus, on which the CPU and several other high-performance *Direct Memory Access* (DMA) devices reside, and a bridge to an *Advanced Peripheral Bus (APB)* which interconnects low-power peripherals like timer or I/O modules [ARM99]. While the AHB bus features high performance, pipelined operation, burst transfers, and multiple bus masters, the APB bus provides a reduced interface complexity and is optimized for minimal power consumption.

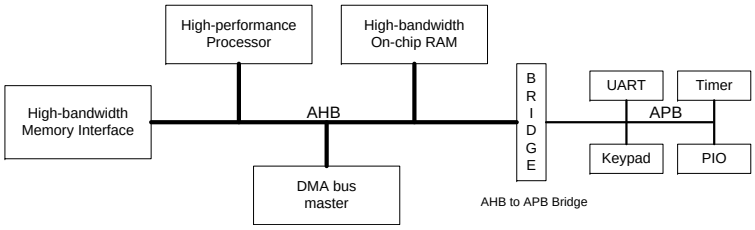

Figure 3.1: Typical AMBA AHB-Based Microcontroller

The following four types of components are differentiated in an AMBA AHB system design, as shown in Figure 3.2.

AHB Master. Bus masters are the initiators of read and write operations. An AHB design may contain one or more bus masters, but at any time at most one master is allowed to actively use the bus.

AHB Slave. A bus slave is associated with a specific address range and responds to read or write operations within that range. In addition to executing the read or write commands, it indicates the status of the previous transfer (e.g., success or failure) to the active master.

AHB Arbiter. The purpose of the bus arbiter is to ensure that at any point in time, only a single bus master is granted access to the bus. Depending on the application requirements, different arbitration protocols (e.g., priority-based arbitration) can be implemented.

AHB Decoder. The AHB decoder selects the slave that is involved in a transfer by decoding the address of the given transfer. Each AHB implementation requires a single centralized decoder.

According to the specification, the AHB protocol is based on a central multiplexer interconnection scheme. Each master connected to the bus indicates an intended transfer by driving its address and control signals. The *arbiter* selects—via the central address and control multiplexer and the write data multiplexer—a single master from which the address, control, and data signals are routed to all the slaves. The *decoder* controls the read data multiplexer which selects signals from the appropriate slave.

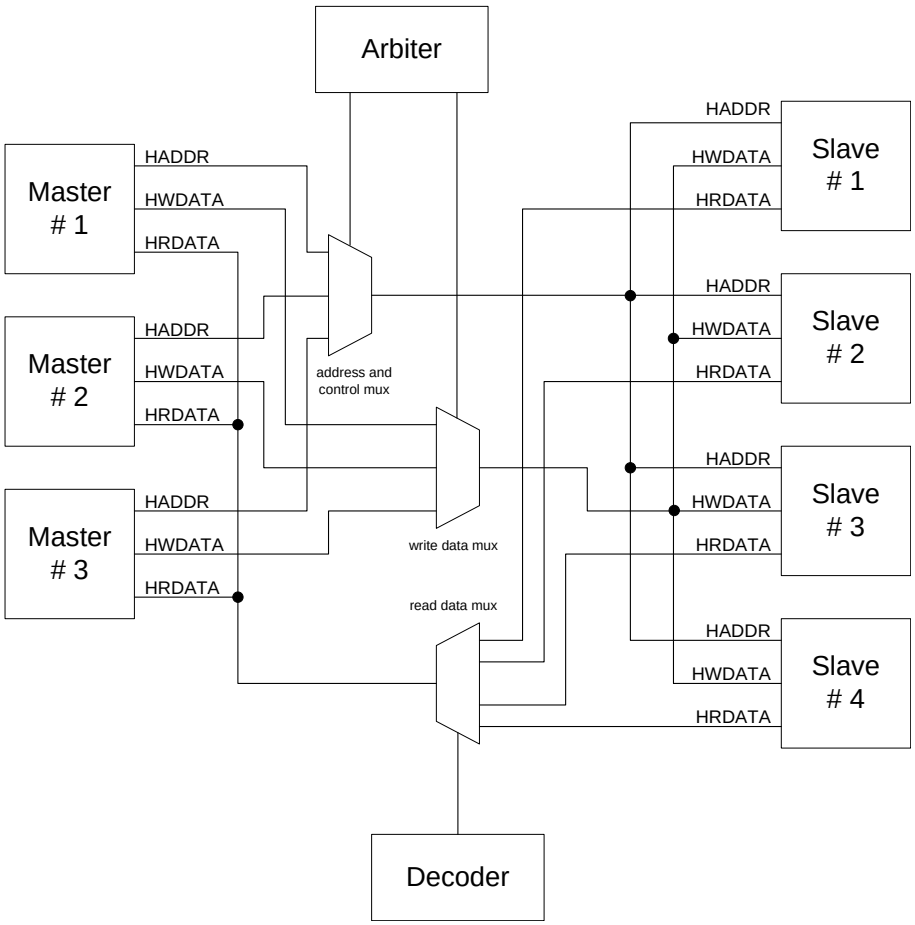

Figure 3.2: AHB Interconnection Scheme

A bus master must be granted access to the bus before it can start a transfer. For this purpose it asserts a request signal to the central arbiter. After the arbiter has granted the access to the master, the master starts the transfer by driving the control and address signals, which contain the information of the

address, direction, and width of the transfer. Furthermore, it indicates whether the current transfer is part of a burst operation or not. AMBA defines a burst operation as one or more data transactions initiated by a single bus master with a consistent width to an incremental address.

3.1.2 Multi-Layer AHB

Multi-layer AHB is an interconnection scheme that overcomes the restriction of using a single shared bus by supporting parallel access path between multiple masters and slaves using an advanced *interconnect matrix* [ARM01]. It is backward compatible to the standard AHB protocol which means that previously designed master and slave modules can be reused without any modification.

Multi-layer AHB supports a wide variety of bus structures. In a full multi-layer structure, each master has a dedicated AHB layer which is connected to every slave via an interconnect matrix (see Figure 3.3).

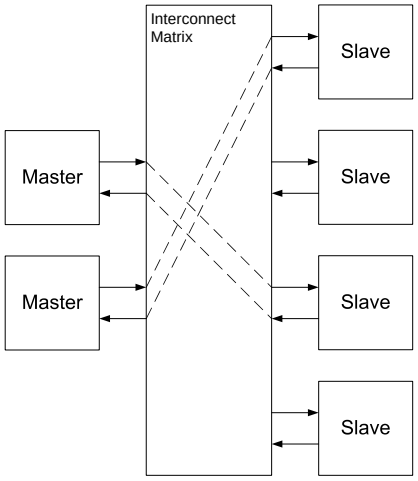

Figure 3.3: AHB Multi-Layer Basic Structure

Within the interconnect matrix, each layer has a dedicated decoder that determines which slave is selected for a given transfer, as depicted in Figure 3.4. If two or more layers require access to the same slave, arbitration becomes necessary. Since each slave port has its own dedicated arbitration logic, an individual arbitration scheme (e.g., round robin or fixed priority) can applied for each slave.

In order to reduce the complexity of the interconnect matrix in systems with a large number of masters and slaves, the following options can be used to optimize the system architecture [ARM01].

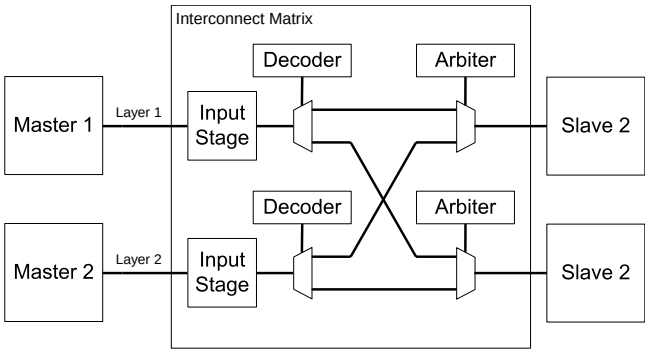

Figure 3.4: AHB Multi-layer Interconnect Matrix

Local Slaves. When a given set of slaves will be exclusively accessed by a single master, or by a group of masters residing on the same layer, they can be made private to the corresponding layer, as shown in Figure 3.5. Private slaves do not add complexity to the interconnect matrix.

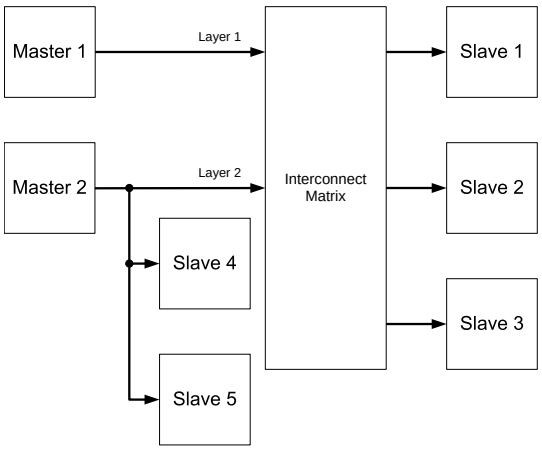

Figure 3.5: AHB Multi-Layer Local Slaves

Multiple Slaves on a Single Slave Port. Multiple low-bandwidth slaves can be combined to appear as a single slave to the interconnect matrix (see Figure 3.6). This concept is particularly attractive if a set of slaves, accessed by a single master during normal system operation, needs to be accessible by *more* than one master in some special circumstances like debugging.

Multiple Masters on a Single Layer. Another way to reduce the cost of the interconnect matrix is to combine multiple masters to share a single layer, as depicted in Figure 3.7. Such an approach can be applied to masters with a low-bandwidth demand that do not require parallel channels.

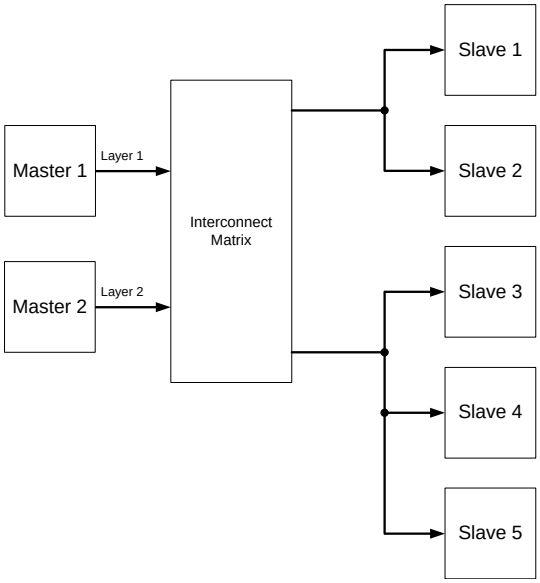

Figure 3.6: AHB Multi-Layer Multiple Slaves on a Single Port

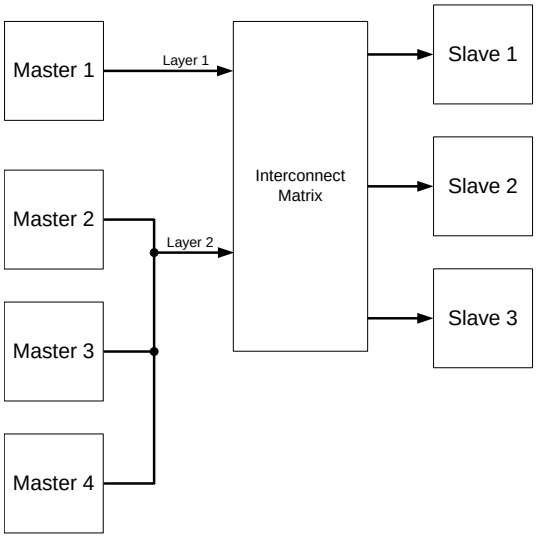

Figure 3.7: AHB Multi-Layer Multiple Masters on a Single Layer

The described optimizations are not restricted to be applied exclusively. They can be combined with each other to connect the different part of a system in an optimal way.

3.1.3 AMBA AXI

AMBA AXI (also called AMBA 3.0) [ARM04] represents the latest generation of the AMBA protocol family. It provides features to maximize resource

efficiency and data throughput like the support for multiple outstanding transactions, out-of-order transaction completion, and efficient burst transactions with only the start address issued.

In AMBA AHB, each transfer consists of an address and a subsequent data phase [BM06]. Transaction pipelining is supported in the following way. The address for the $(i+1)^{th}$ transfer can be driven on the address lines while the data phase of the i^{th} transaction is in progress. Since address sampling is only allowed after the data phase has completed, multiple outstanding transactions are not possible. In contrast to the AHB protocol, AXI supports issuing multiple outstanding transactions and out-of-order transaction completion. For this purpose a master can assign an ID tag to every issued transactions. All transactions with the same ID tag have to be completed in order while transactions with different ID tags can be completed out of order. Out-of-order transactions enable fast-responding slaves to complete ahead of slower slaves, even if the transactions involving the slower slaves have been issued first (i.e. a fast transaction can overtake a slower transaction).

The AMBA AXI architecture introduces the following five independent channels [ARM04].

Read address channel. The read address channel carries all of the required address and control information for a read transaction.

Write address channel. The write address channel carries all of the required address and control information for a write transaction.

Read data channel. The read data channel transports the read data and any read response information (e.g., the completion status of the transaction) from the slave back to the master that initiated the transaction.

Write data channel. The write data channel transports the write data to the slave. The information on this channel is always treated as buffered, so that the master does not have to wait for acknowledgments of previous write transactions to perform new write transactions.

Write response channel. This channel is used by slaves to respond to write transactions (e.g., completion signaling). The completion signal occurs only once for each burst, and not for each transfer within a burst.

Figure 3.8 depicts how the read address and the read data channels are used during a read transaction. Figure 3.9 shows how the write address, the write data, and the write response channels are used during a write transaction.

Each of the channels uses a two-way *VALID* and *READY* handshake mechanism. The channel source uses the *VALID* signal to indicate that new data

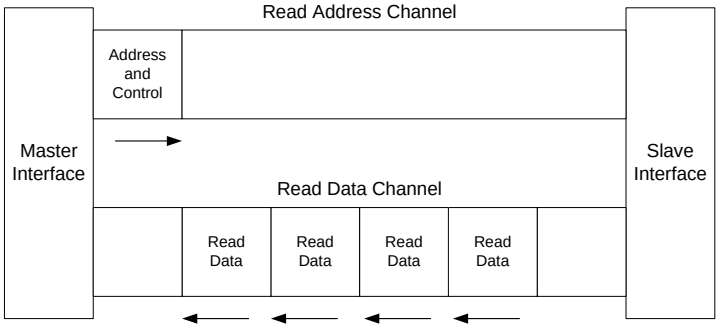

Figure 3.8: AMBA AXI Read Transaction

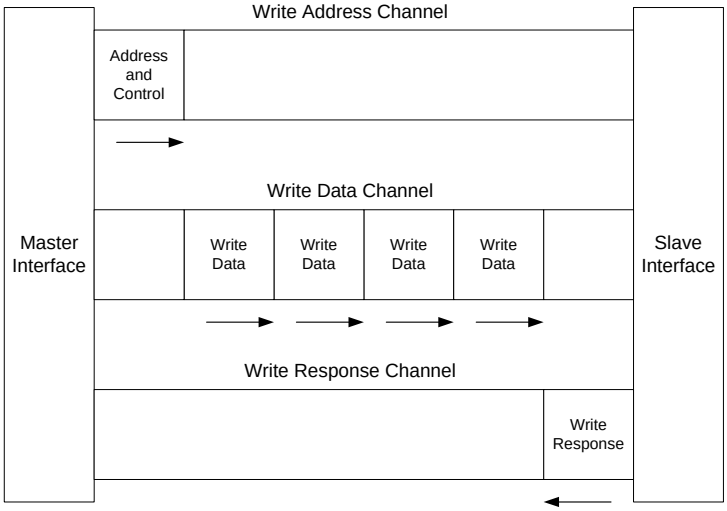

Figure 3.9: AMBA AXI Write Transaction

or control information is available on the channel, while the destination uses the *READY* signal to show that it can accept new data. The read and the write data channel include an additional signal, the *LAST* signal, to indicate the transfer of the final data item of a transaction. This two-way handshake mechanism allows both the master and the slave to control the rate at which data or control information is transfered.

AXI channels are strictly unidirectional, and there is no requirement for a fixed relationship between the various channels. Depending on the implementation of the interconnect, the bandwidth of the individual channels can differ. In most systems one of the following three interconnect approaches is used [PM05]:

- Shared Address and Single Data buses (SASD)
- Shared Address buses and Multiple Data buses (SAMD)
- Multiple Address buses and Multiple Data buses (MAMD)

In the SASD approach, only a single master can be active per channel. This concept is similar to the AMBA AHB protocol.

When SAMD is used, only a single master can be active per address channel, while multiple master and slave pairs can be active on the other channels (e.g., master A can send write data to slave A, while master B is sending write data to slave B). The number of master-slave pairs that can be active at the same time depends on the implementation of the interconnect.

Using the MAMD approach, multiple pairs can be active on all channels at the same time. This approach yields the highest performance with respect to the interconnect. On the other hand, this approach results also in the most complex scenario with respect to system verification, as multiple masters and slaves can be active at any time.

The key advantage of AMBA AXI compared to previous AMBA specifications (e.g., AMBA AHB) lies in the interface definition of the attached cores. In AMBA AHB the master and the slave interfaces include signals that are connected to bus-internal components like the arbiter [BM06]. Since the internal bus architecture is exposed to the interface of the cores, it is not possible to directly connect a master interface to a slave interface. Furthermore, the implementation of the interconnect is tightly coupled to the interface of the cores. Contrary to previous AMBA specifications, ABMA AXI is based on a *point-to-point* interface definition (see Figure 3.10) [BM06].

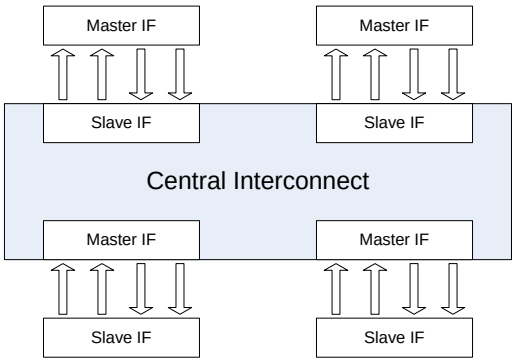

Figure 3.10: AMBA AXI Point-to-Point Master/Slave Interfaces

The protocol provides a single interface specification to describe interfaces:

- between a master and the interconnect
- between a slave and the interconnect
- between a master and a slave

A connection between any two devices consists of a master interface and the symmetrical slave interface [BM06]. The signals between a master interface and a slave interface are not specific to the implementation of the interconnect. Furthermore, the interface through which a master (resp. a slave) is connected to the interconnect matches the interface of the corresponding slave (resp. master). This means that a master and a slave can be connected directly to each other without modification of the interface. Since the interface definition is decoupled from the implementation of the interconnect, a variety of implementations can be used (e.g., Æthereal [GDR05] implements, among other protocols, AXI at its boundaries).

3.2 OCP

The *Open Core Protocol* (OCP) [OCP05] defines a bus-independent point-to-point interface between two communicating IP cores. An IP core can be a simple peripheral core, an on-chip communication subsystem, or a complete high-performance microprocessor. In OCP one of the communicating entities acts as a master and the other one as the slave. Commands can be issued exclusively by the master. The slave responds to the commands of the master, either by accepting data from the master or by presenting requested data to the master. For setting up a peer-to-peer communication, two instances of the OCP are required: one where the first core acts as the master and the second core acts as the slave, and one where the first core acts as the slave and the second core acts as the master. It is determined by the inherent characteristics of an IP core whether it needs the master side, the slave side, or both sides of the OCP (see Figure 3.11).

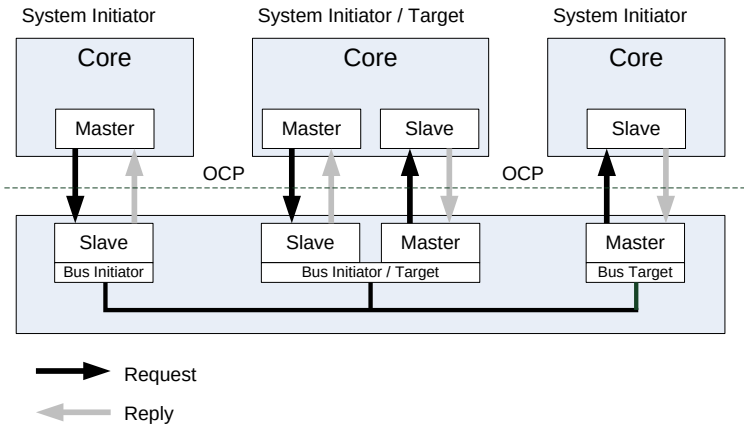

Figure 3.11: OCP Core Interfaces

The OCP supports write and read commands, which enable a master to write or read a specific address to or from the slave. In addition to the read and write commands so-called sideband signaling for signals like interrupts, reset, or errors is supported. The OCP is a strictly synchronous interface, where all signals crossing the interface are driven by a single clock. All signals are strictly uni-directional and point-to-point. The OCP does not specify the interconnecting bus functionality or its implementation. It is the responsibility of the designer of the communication subsystem to handle arbitration and to convert an OCP request by a master into an implementation specific bus transfer and to reconvert it to a legal OCP command at the slave's side. The bus-independent interface allows developers of OCP compliant IP cores to think of them as being directly connected to each other in a point-to-point fashion without an intervening on-chip bus. Thus, the complexity of bus-dependent device selection and arbitration mechanisms is hidden from the IP cores.

3.3 Sonics SiliconBackplane

The Sonics SiliconBackplane uNetwork [Son02] is an on-chip network for the interconnection of multiple IP cores within an SoC. It unifies all on-chip communications by replacing dedicated physical connections between IP cores with logical connections realized on a shared interconnect. Furthermore, it provides guaranteed bandwidth and bounded latency to individual IP cores. The SiliconBackplane Network uses so-called agents (standardized network interfaces) to decouple the implementation of the SiliconBackplane μNetwork from the individual IP cores, enabling the IP cores to be designed independently form the network.

The interface to an agent complies with the Open Core Protocol (OCP) [OCP05]. An IP core can be connected as an OCP master, as an OCP slave, or with both interfaces. Guaranteed bandwidth and bounded latency for individual cores is achieved by a combination of a fixed-latency bus and a *Time Division Multiple Access* (TDMA) bandwidth allocation scheme. The TDMA allocation scheme is based on recurring frames which consist of up to 256 cycles. Each cycle in a frame can be allocated to at most one agent. An agent that is allocated a cycle is permitted to send a message on the bus during that cycle. If an agent does not need its cycle, the cycle is dynamically allocated in a round-robin fashion among the IP cores waiting for access. Thus, a two level arbitration for high priority and lower priority data transfers scheme is realized.

3.4 Æthereal

The *Æthereal* architecture [GDR05] combines guaranteed services—such as guaranteed throughput and bounded latency—with best effort services. Guaranteed services aid in the compositional design of robust SoCs, while best effort services increase the resource efficiency by exploiting the NoC capacity that is left unused by the guaranteed services.

The constituting elements of the architecture are *Network Interfaces* (NIs) and routers which are interconnected by links. The NIs translate the protocol of the attached IP core to the NoC-internal packet-based protocol. The routers transport messages between the NIs.

Æthereal offers a *shared-memory abstraction* to the attached IP modules [RDG+] and employs a transaction-based master/slave protocol. Masters initiate a transaction by issuing a request which consists of a command (e.g., read or write command at a specific address) and optional write data. These commands are received and executed by one or more slave modules which may issue a response back to the master including the status of the command execution and optional data. The transaction based model was chosen to ensure backward compatibility to existing on-chip network protocols like AXI [ARM04] or OCP [OCP05].

In the Æthereal NoC, the signals of an IP core with a standardized interface (e.g., AXI or OCP) are sequentialized in request and response *messages*, which are transported by the NoC in the form of *packets*. The communication services are provided via so-called *connections* which are composed out of multiple unidirectional peer-to-peer *channels*: A typical *peer-to-peer OCP connection* would consist of a request channel and a replay channel; a *multi cast* or *narrow cast* connection can be implemented by a collection of channels, one or two for each master-slave pair; see Figure 3.12, [BM06].

Each channel uses two queues, one at the NI of the sender and one at the NI of the receiver. In order to guarantee message delivery, an end-to-end credit-based flow control mechanism is employed that prevents queue overflows in the NIs (see Figure 3.13, [BM06]). Each channel incorporates a counter (on the sender side of the channel) that tracks the empty buffer space of the remote destination queue [RDG+]. The initial value of counter is the size of the remote queue, and the counter is decremented each time, when data is sent from the source queue to the remote queue. When the receiver consumes data from the remote queue, credits are generated to indicate that more space in the remote queue is available. These credits are sent to the producer of the data where they are added to the counter that tracks the empty buffer space of the remote destination queue. Credits can be sent in dedicated credit packets, or they can

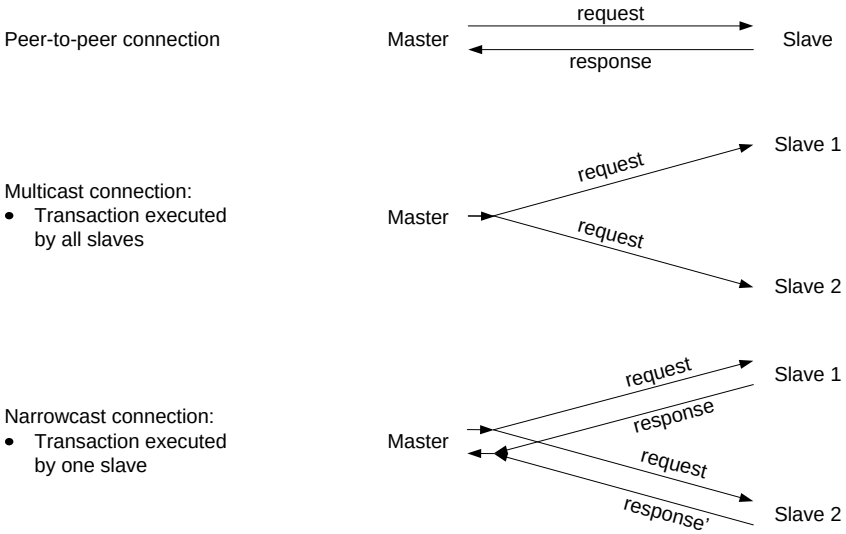

Figure 3.12: Æthereal Different Types of Connections

be piggybacked in the header of the packets for data in the other direction to improve the efficiency of the NoC.

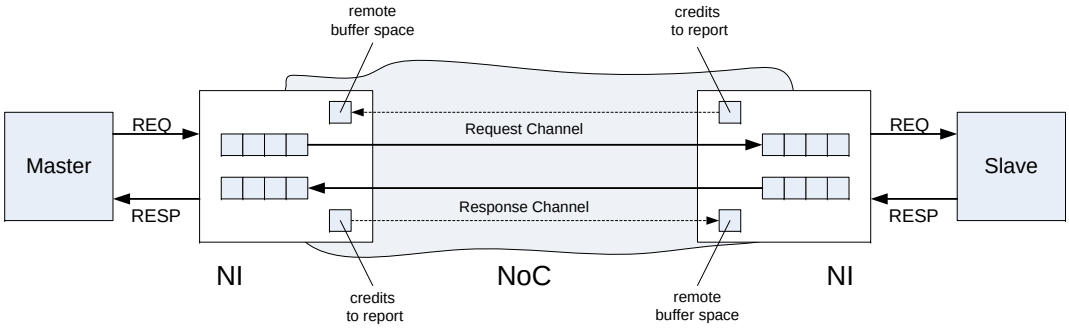

Figure 3.13: Æthereal Credit-based Flow Control

Channels offer two types of service classes: *guaranteed throughput (GT)*, and *best effort (BE)*. GT channels give guarantees on minimum bandwidth and maximum latency by using a TDMA scheme. The TDMA scheme is based on a table with a given number of time slots (e.g., 128 slots). In each slot, a network interface can read and write at most one block of data. Given the duration of a slot, the size of a block that can be transferred within one slot, and the number of slots in the table, a slot corresponds to a given bandwidth B. Therefore, reserving N slots for a channel results in a total bandwidth of NB. The granularity with which the bandwidth of a channel can be reserved equals $1/S$ of the maximum channel bandwidth, where S is the number of slots in the TDMA table.

Within a single channel, temporal ordering of messages is guaranteed. This means that all messages are received in the same order as they were sent. Since the NI treats different channels as different entities, ordering guarantees are only provided for single channels. Across different channels, message reordering is possible.

An *Æthereal* NI consists of a single fixed *kernel* and one or more variable *shells* (see Figure 3.14, [BM06]). The NI shells translate transactions of a specific IP protocol such as AXI or OCP to generic messages that are accepted by the NI kernel. The NI kernel takes these generic messages marshals them into GT or BE packets, schedules them to the routers, and implements end-to-end flow control. The structure of the messages handed over by the NI shells may vary depending on the specific protocol used by the attached IP core.

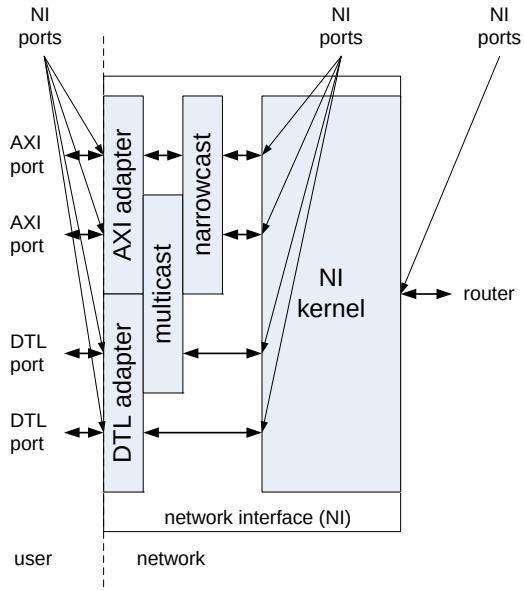

Figure 3.14: Æthereal NI Kernel and Shells

From the point of view of the NI kernel, the message structure is irrelevant, as it regards messages just as pieces of data to be send or received over the NoC. Since the protocol specific functionality is confined by the NI shells, an NI can be instantiated with different NI protocols by attaching the corresponding shells.

Chapter 4

The TTSoC Architecture

The *Time-Triggered System-on-a-Chip* (TTSoC) architecture is a novel system architecture for SoCs that offers a component-based design methodology for managing the complexity of chips with billions of transistors through the consequent decoupling of the computational components from the communication infrastructure. Due to this decoupling, the design of a computational component can abstract from the implementation of the interconnect, which facilitates the rapid development of multi-core SoCs by using pre-verified functional cores. Therefore, the time-triggered SoC architecture provides an architectural framework that supports the side-effect-free composition of component services (based solely on the interface specifications) to form larger systems-of-systems.

This chapter describes the overall system structure and the component model of the TTSoC architecture. Furthermore, it introduces the concept of *on-chip TMR* and *off-chip TMR*, as the architecture's basic fault-tolerance mechanisms.

4.1 Overall System Structure

A central issue in the TTSoC architecture is the provision of standardized, validated, and potentially certified architectural services (e.g., communication, diagnostic, or fault-tolerance services) that facilitate the development of distributed real-time applications. The architectural services separate the application functionality from the underlying platform technology in order to reduce design complexity and to facilitate design reuse, which corresponds to the concept of platform-based design [SV02]. This section describes how the architectural services are organized in the TTSoC architecture.

The TTSoC architecture is intended to be used across multiple heterogeneous application domains, with highly specific and potentially mutually contradict-

ing requirements with respect to the architectural services. Examples for different requirements are the desired properties of communication protocols:

Safety-critical applications (e.g., steer-by-wire systems) that have to deliver a given service within a guaranteed time bound require communication services that are highly deterministic in the temporal domain (e.g., time-triggered protocols). In contrast to safety-critical applications, many non-safety-critical systems typically have to be optimized for average load in order to optimize cost (e.g., the comfort electronic system of car) and thus require a highly flexible protocol that adheres to the event-triggered communication paradigm.

Additional architectural services have to be provided to support the reuse and the integration of legacy applications by emulating the corresponding legacy platform. Examples are emulations of standard communication protocols that are frequently used in specific application domains (e.g., CAN [Rob91] in the automotive sector, ARINC 629 [Moo89] for avionics, ...).

Consequently, the set of provided architectural services has to be *extendable* and *configurable*. Extendability is an important factor since it is not possible to anticipate how applications and their requirements will evolve in the future. Furthermore, it is a strong requirement that a newly added (maybe faulty) service must not compromise the correct operation of existing services in the temporal and in the value domain. Configurability is required to facilitate the construction of resource-efficient systems. Therefore, the architecture should be adaptable in a way, that any particular instantiation contains only those services that are required in the actual system.

In order to support the flexible integration of architectural services for heterogeneous application domains, the TTSoC architecture was designed as a *waistline architecture* (see Figure 4.1), which is based on a generic and minimal set of validated and verified *core platform services*. These services are provided via the so-called UNI and form the *waist* of the waistline architecture. The UNI abstracts from the actual implementation of the core platform services which means that any NoC that implements the UNI is compliant to the TTSoC architecture.

Based on the UNI, a broad range of higher-level services can be realized which are tailored to the requirements of specific application domains and refine or extend the core platform services. Multiple higher-level services (e.g., tailored to multimedia or to safety-critical applications) can coexist in a single system. The TTSoC architecture provides encapsulation mechanism that prevent any unintended interference between high-level services of different application subsystems, which is a key requirement for the integration of application subsystems of different criticality levels.

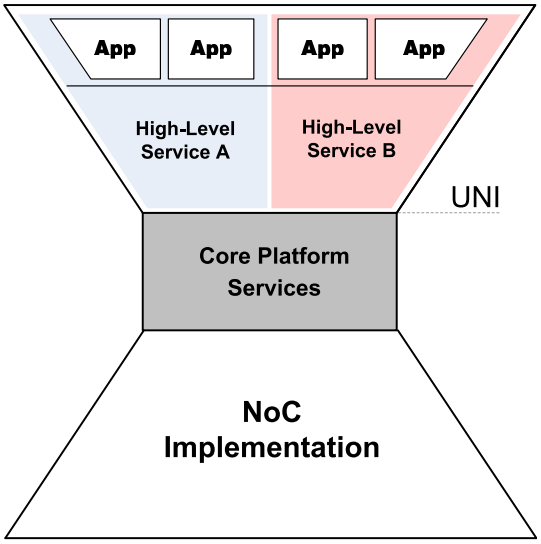

Figure 4.1: Waistline Architecture

4.2 The Component Model

This section gives an overview on the constituting architectural elements of the TTSoC architecture. The TTSoC architecture is built around a deterministic time-triggered NoC which interconnects multiple, possibly heterogeneous IP cores called *micro components* (see Figure 4.2).

A micro component is a self-contained computational unit that provides its functionality over a well defined *message-based* interface. It is composed out of two structural elements, the TISS and the *host*. While the host performs the computations, which are required to deliver the intended service of the micro component, the TISS provides a stable set of core platform services (e.g., predictable transport of messages, global time service, watchdog service) to the host. Furthermore, the TISS acts as a guardian for the NoC by ensuring that a fault within the host of a micro component (e.g., a software fault or a hardware fault) cannot lead to a violation of the micro component's temporal interface specification in a way that the communication between other micro components would be disrupted.

The TTSoC architecture supports integrated resource management by two dedicated architectural elements, the *Resource Management Authority* (RMA) and the *Trusted Network Authority* (TNA). The RMA accepts resource request messages from the micro components and generates, according to internal rules, a resource allocation mapping for the entire *System-on-Chip* (SoC). The TNA checks the resource allocation mapping, provided by the RMA, against a set of predefined constraints (e.g., conflict-freeness of the message schedule or

availability of statically assigned resources for safety-critical application subsystems). If the mapping is valid, the TNA (re-)configures the NoC and the TISSs accordingly.

The TNA, the time-triggered NoC, and the TISSs form the *Trusted Subsystem* (TSS). The TSS constitutes the core of the TTSoC architecture and is assumed to be free of design faults. It has to be certified according the criticality level of the most critical micro component in the SoC.

The TTSoC architecture supports *gateways* to access chip-external networks and to facilitate the construction of distributed systems by the interconnection of multiple SoCs.

Furthermore, the architecture incorporates a dedicated architectural element for diagnosis, the *Diagnostic Unit* (DU).

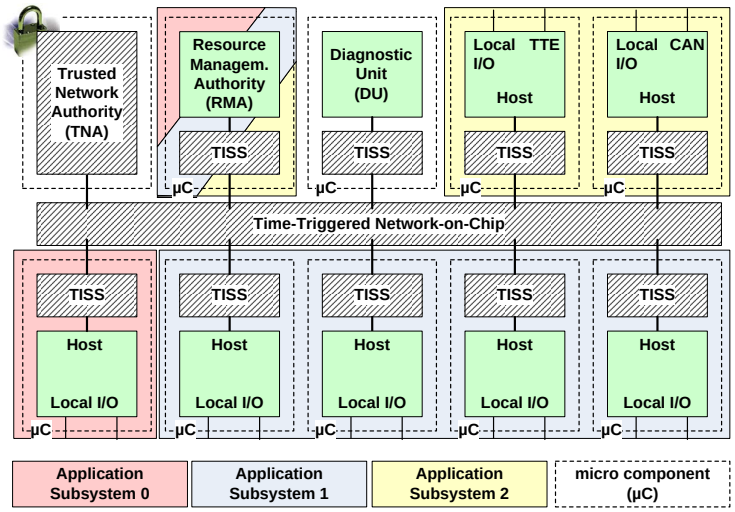

Figure 4.2: Component Model

4.2.1 Micro Components

A major characteristic of the TTSoC architecture is the high level of abstraction with respect to system integration. Existing state-of-the-art NoC architectures like Æthereal or the SiliconBackplane provide a shared memory abstraction to the attached IP cores via a transaction based master/slave protocol as used in OCP or AMBA AXI. Such protocols define low-level signals like address, data, interrupt, reset, or clock signals and are typically employed at the interfaces of processors, memory subsystems, or bus bridges.

The TTSoC architecture raises the level of abstraction by introducing the notion of a *micro component* which is a self contained computational unit (e.g., a

processor or an FPGA with local memory) that provides its functionality over a *message-based* interface which is defined in the value as well as in the temporal domain. Thus, a micro component is more than a processor, or a memory subsystem. It can be viewed as an entire computer that provides a defined part of an application service. On this abstraction level, a message-based interface that defines the exchanged messages at the application level, is much more appropriate than an interface on signal-level as it is used for a processor-memory interconnect.

Encapsulation

A key objective of the TTSoC architecture is to facilitate independent development of application subsystems. This is accomplished by the use of encapsulation mechanisms that prevent any unintended interference between these subsystems. The TTSoC architecture ensures by design (i.e., by the physical separation of the individual micro components), that micro components can interact with each other exclusively by the exchange of messages via the time-triggered NoC; this means that there are no other hidden channels (e.g., implicit interaction via shared memory or interrupts).

Since the time-triggered NoC is a shared resource, it has to be prevented that a design fault (e.g. a software fault) within a given micro component can lead to a violation of the micro component's temporal interface specification in a way that the communication between other micro components can be disrupted. In order to prevent any temporal interference (e.g., delaying messages of other micro components) or spatial interference (e.g., overwriting a message produced by another micro component), a micro component is structured into two architectural elements, the *host* and the TISS (see Figure 4.3).

The host performs the computations that are required to deliver the intended service of a micro component. It can belong to different criticality levels, can be developed by different suppliers, and is generally not assumed to be free of design faults.

Contrary to the host, the TISS, which is a part of the TSS, is assumed to be free of design faults and is certified to the highest criticality level of any host within the SoC. It provides a stable and validated set of core platform services to the host via the UNI, and acts as a guardian for the shared time-triggered NoC by accessing it exclusively at a priori known points in time according to the TDMA scheme. Therefore, each TISS incorporates its own dedicated time-triggered message schedule—the *Message Descriptor List* (MEDL)—which is written exclusively by the TNA and holds the information about the global points in time of all message receptions and transmissions of the respective

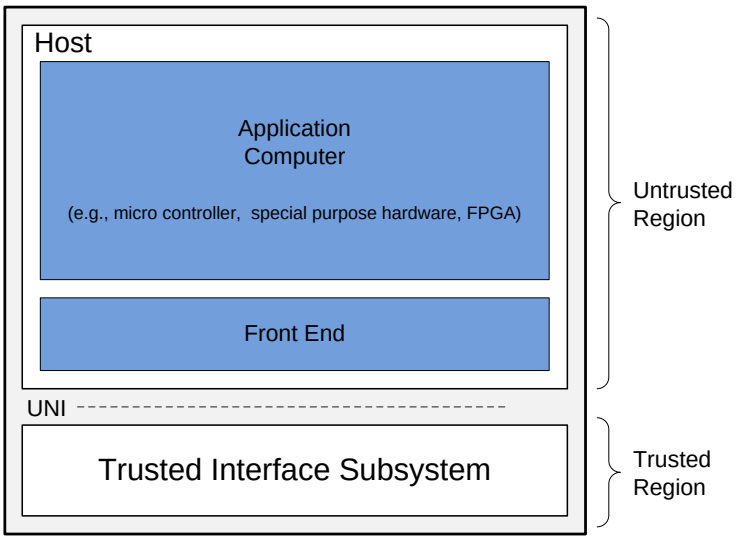

Figure 4.3: Structure of a Micro Component

micro component. The design of the TISS ensures, that the host cannot modify the MEDL and thus, cannot interfere with the correct operation of the communication infrastructure.

The sum of all TISSs and the TNA, establish together a *Fault Containment Region* (FCR) for the TSS. *An FCR is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region [LH94].* In return, the TSS establishes independent FCRs for each host, which means that a host will operate correctly regardless of any fault in any of the other hosts.

Structure of a Host

The host performs the computations that are required to deliver the intended service of the micro component. It is structured into two architectural elements, the *application computer* and the *frontend*. The application computer provides the computational resources of the micro component and controls the micro component's local I/O interfaces (e.g., for sensors or actuators). It can be realized as a general-purpose microcontroller, as a specialized hardware IP block (e.g., an MPEG encoder), or as a *Field Programmable Gate Array* (FPGA). If the application computer supports dynamic resource management, it will typically contain an execution environment, which communicates with the RMA and switches between different operational modes or logical application functions that should be executed on the micro component.

The frontend customizes the core platform services provided at UNI according

to the requirements of the application computer and the application subsystems that are intended to be executed on the micro component. In its simplest version, the frontend is realized as a dual-ported memory providing a temporal firewall interface [Kop97] to the application computer. If required, the frontend can provide higher-level services, which are tailored to needs of specific application domains. Examples are a fault tolerance service which performs majority voting of replicated inputs for failure masking by TMR, or an encryption and decryption service to facilitate secure communication with chip external entities.

The decision to realize the frontend within the host, instead of incorporating it into the TISS, was motivated by the following design drivers.

Openness and Configurability. If domain-specific services are exclusively realized in a higher layer above the TISS (i.e., the frontend), the interface of the TISS itself (i.e., the UNI) can be kept stable and uniform. The uniformity of the UNI enables the instantiation of a single pre-validated implementation of the TISS in multiple micro components, even if the micro components require different domain-specific services. Furthermore, the concept of the frontend allows the addition and removal of domain-specific services without changing the UNI or the TISS.

Since the UNI has an open interface specification, third party suppliers are able to develop domain-specific services for the TTSoC architecture by providing dedicated frontends for their application domain.

Fault Containment. Since third party suppliers should be unrestrictedly allowed to provide additional domain-specific services, these services can not always be assumed to be free of design faults. Therefore, non-interference and fault containment with respect to domain-specific services is a key issue for the TTSoC architecture. As mentioned above, a host is an FCR for design faults. Since the frontend is part of the host, a design fault in the frontend can only affect the local host but cannot disrupt the correct computation or communication of any other micro components. Thus, it is sufficient to certify a frontend according to the criticality level of the micro components in which it will be employed. This property is of major significance for the integration of application subsystems with mixed criticality levels.

Interfaces of a Micro Component

A micro component incorporates the following four interfaces (see Figure 4.4).

Linking Interface. A host provides its real-time services, and accesses the real-time services of other hosts by the exchange of messages across its *Linking Interface* (LIF). These messages have to be fully specified in the value and the time domain in a LIF specification (see Section 5.2.2). According to [KS03] a LIF specification consists of an *operational* specification and a *meta-level* specification. While the operational specification deals with the syntactic and temporal aspects of the messages exchanged across the LIF, the meta-level specification describes the meaning of the information contained in these messages.

The LIF of a host consists of multiple *ports* through which the host sends and receives messages.

TISS CP. The TISS provides a *configuration and planning interface (TISS CP)* through which the TNA configures TISS-internal parameters like the time-triggered message schedule. To prevent any other micro component than the TNA from accessing the TISS CP interface, the architecture provides an encapsulated and protected channel from the TNA to the CP interface of all TISSs. This encapsulated channel can be realized as a separate physical channel, or by dedicated time slots on the main NoC. To prevent that a design fault within the local host can interfere with the temporal control of the time-triggered NoC, it is assured by design that the host can not change the TISS-internal configuration written by the TNA.

HOST CP. Hosts that support dynamic resource management, incorporate a *Host CP* interface through which the parameters of the execution environment of the host can be configured by the RMA. Examples for such parameters are the application functionality that should be executed on the host or performance parameters like clock frequency or power mode.

Local Interfaces. A host can incorporate *local interfaces* through which it accesses peripherals (e.g. sensors, actuators, or gateways to other networks) in its environment.

4.2.2 Time-Triggered NoC

The time-triggered NoC interconnects the micro components within an SoC and supports the predictable transport of periodic and sporadic messages. Furthermore, it performs clock synchronization in order to provide a global time base for all micro components even if they reside in different clock domains.

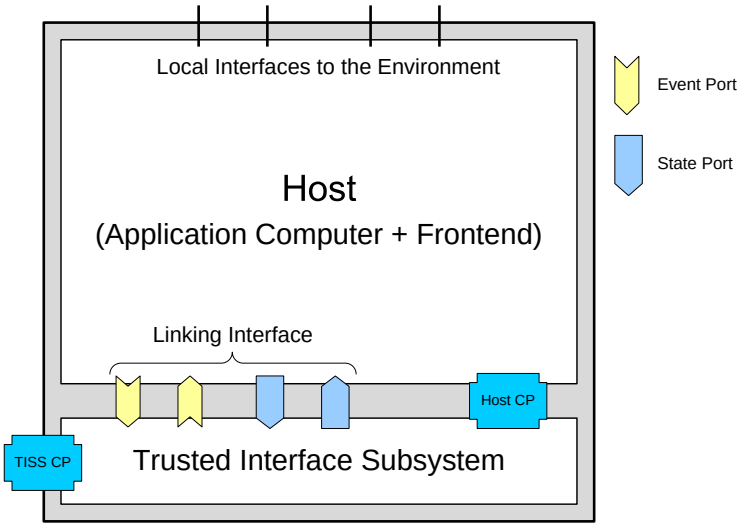

Figure 4.4: Interfaces of a Micro Component

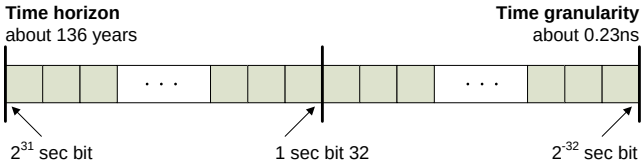

Figure 4.5: Time Format of Time-Triggered NoC

Time Format

A digital time format can be characterized by three parameters: the *granularity*, the *horizon* and the *epoch*. The granularity determines the minimum interval between two adjacent ticks of a clock, i.e., 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 will wrap around. The epoch determines the instant when the measuring of time starts.

The time format of the time-triggered NoC is a binary time-format that is based on the physical second (see Figure 4.5). Fractions of a second are represented as 32 negative powers of two (down to about 230 picoseconds), and full seconds are presented as 32 positive powers of two (up to about 136 years). Thus, each instance can be represent by eight bytes. This time format is closely related to the time-format of the *Global Positioning System* (GPS), which has the same epoch and is also based on the physical second. In case there is no external synchronization [KO87], the epoch starts with the power-up instant.

Predictable Transport of Messages

According to the TDMA scheme, the available bandwidth of the network is divided into conflict-free sending slots, i.e., the sending slots are allocated in way that any contention within the NoC is prevented. Sending slots can be used for the transmission of *periodic time-triggered* messages and for the transmission of *sporadic time-triggered* messages. Contrary to periodic time-triggered messages, sporadic time-triggered messages are only sent—and thus energy is only consumed—whenever the sender has to transmit a new event to the receiver.

The TTSoC architecture and existing TDMA-based SoC architectures like Æthereal and the Sonics SiliconBackplane differ in the way *how* and *for which purpose*, the TDMA scheme is employed. Common to all TDMA-based architectures is, that the TDMA scheme is employed to avoid conflicts on the shared interconnects and to provide encapsulation and timing guarantees for the individual communication channels. The objective of Æthereal and the Sonics SiliconBackplane is to establish resource guarantees with respect to *bandwidth* and *latency*. In contrast to these NoCs, the TTSoC architecture uses the TDMA scheme to schedule periodic send instances of entire *application-level* time-triggered messages. The supported periods are in the range from a few nanoseconds up to milliseconds or seconds. This enables the temporal alignment of the activities within the network and the activation of periodic application tasks in order to reduce the end-to-end latency of the overall application, which is vital for many types of real-time systems.

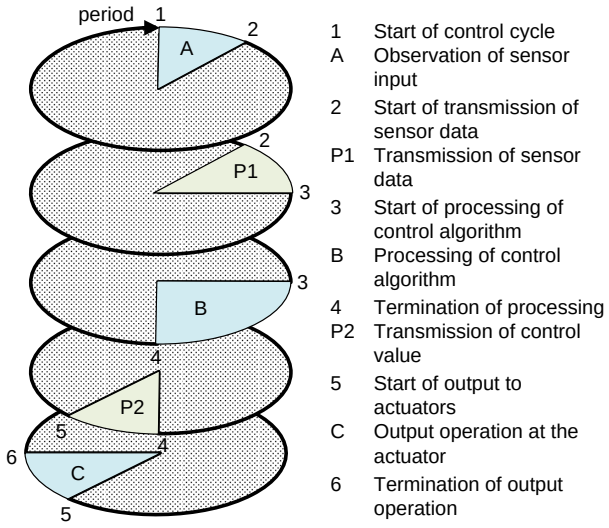

Figure 4.6: Temporal Alignment in Control Loops

As an example, consider a control loop which is realized by three micro com-

ponents, as depicted in Figure 4.6. Micro component `A` acquires the measured value of the controlled variable via locally connected sensors, micro component `B` calculates the value of the manipulated variable according to some kind of control algorithm, and micro component `C` controls the corresponding actuators according to the value of the manipulated variable. The periodic time-triggered message `P1` communicates the measured variable from micro component `A` to micro component `B`, and the periodic time-triggered message `P2` communicates the value of the manipulated value from micro component `B` to micro component `C`. In this example the micro components and the time-triggered messages have the same period. The activities of the three micro components and the two time-triggered messages are temporally aligned such that the end-to-end latency (i.e., the time interval form the instant when the controlled variable is observed to the instant when the value of the manipulated variable is applied to the actuators) is minimized. In fact, the worst case end-to-end latency equals the sum of the worst case execution times of all micro components and the transmission time of the two messages. Such a real-time transaction is denoted as a *phase-aligned transaction* [Kop97].

Contrary to the TTSoC architecture, existing SoC architectures do not support phase-aligned transactions, but only provide guaranteed bandwidth to individual senders without supporting temporal alignment. In architectures only supporting bandwidth guarantees without temporal alignment, one of the following design decisions has to be made.

Over-Dimensioning of Communication Resources. One possible approach lets each core reserve the maximally required bandwidth throughout the entire period of the control cycle to achieve the desired end-to-end latency. This most likely results in a resource-inefficient design since all bandwidth reservations would have to be satisfied all the time, even if the individual cores will never use them at the same time.

Performance Degradation. Another approach is to reserve only that amount of bandwidth that is needed to transmit a message within one period (i.e., a sender will send a message always into the "next round"). In this case, over-dimensioning of resources is avoided, but each sender of a sequential transaction will add a delay of one period to the end-to-end latency. For many real-time systems, this increased end-to-end latency would lead to a degraded system performance (e.g., controller will have a lower quality).

Periodic Reconfiguration. The third possibility, would be to emulate time-triggered messages by periodically performing reconfiguration of the network in order to free and re-allocate communication resources in each

period. If these reconfiguration activities are visible to the system integrator, this approach would significantly contribute to the cognitive complexity of the overall design.

Similarly, in a fault-tolerant system that masks failures by triple-modular redundancy, a high bandwidth communication service is required for short intervals to exchange and vote on the state data of the replicated channels. Therefore, a real-time communication network should take consideration of these pulsed communication requirements and provide appropriate services.

The TTSoC architecture implements time-triggered messages by a novel communication primitive called *pulsed data stream* [Kop06]. A pulsed data stream is a time-triggered periodic unidirectional data stream that transports data in pulses with a defined length (one pulse corresponds to one message) from *one* sender to n a priori identified receivers at a specified phase of every cycle of a periodic control system.

A pulsed data stream consists of periodic *pulses* with a defined *period*, *phase*, and *duration*. The design restricts the pulse periods of a pulsed data stream to negative powers of two of the second, i.e., a period can be 1 second, 1/2 second, 1/4 second, 1/8 second, and so forth. This restriction is introduced in order to reduce the complexity of the NoC, the computation of the time-triggered schedule, and the dispatching of the periodic messages.

There are theoretically at most 32 different periods in the time-triggered NoC—corresponding to the lower 32 bits in the time format. The period of a pulsed data stream can thus be characterized by the corresponding bit of the binary time format. We call this bit the *period bit*.

We define the start of a period as the periodic instant when all bits to the right of the period bit of the corresponding period are zero. The pulse phase, i.e., the offset of the start of the period to the start instant of the pulse, is given by the bits to the right of the period bit.

A pulse consists of at least one *fragment* of variable size. The *duration* of a pulse specifies the time between the start of the transmission of the first fragment and the end of the transmission of the last fragment. Successive fragments of a pulse data stream do not have to be transmitted in a dense sequence on the NoC. This enables the concurrent transmission of multiple pulses—of different pulsed data streams—over a single physical link by temporally interleaving the fragments of the different pulses.

Figure 4.7 depicts the allocation of TDMA slots for an exemplary pulsed data stream, which consists of two fragments. The free TDMA slots between the two fragments can be used by fragments of other pulsed data streams.

The fragmentation of pulses into fragments is not visible at the UNI. Thus, the application subsystems in the hosts have to deal only with periodic messages that have a defined period, phase, length, and transmission time (the transmission time of a message equals the duration of the associated pulse).

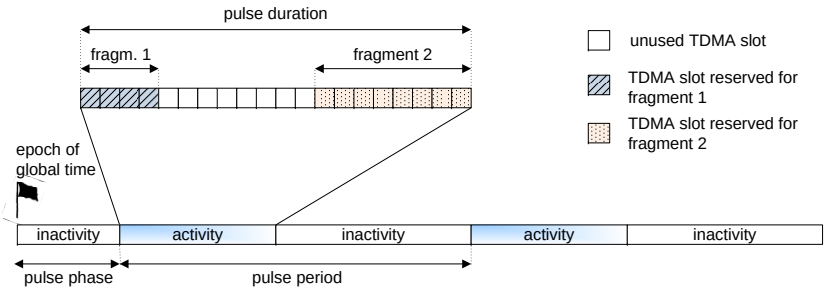

Figure 4.7: Pulsed Data Stream

The proposed NoC works according to the cut-through principle, since the timeliness, complexity, flexibility, and power requirements of an NoC can be drastically reduced if the network does not require any memory for message buffers. This means that no messages are stored within the network (e.g., in switches). If message queuing is required, it is performed exclusively within a micro component, using the micro component's local memory.

4.2.3 Architectural Elements for Resource Management

A major focus of the TTSoC architecture is integrated resource management. The integrated resource management infrastructure facilitates the dynamic allocation of communication resources and power to the individual micro components and the dynamic allocation of micro components to application subsystems.

With respect to resource management, the TTSoC architecture differentiates between two fundamentally different types of application subsystems, *safety-critical* and *non safety-critical* application subsystems. Safety-critical application subsystems have to be certified according to the highest criticality classes (e.g., Class A according to DO-178B). To facilitate validation, verification and certification, the predominant design principles for safety-critical application subsystems are simplicity and determinism. Therefore, also the corresponding resource management mechanisms have to be simple and deterministic. In many cases, safety-critical application subsystems will exclusively rely on statically assigned resources. In contrast, non safety-critical application subsystems do not require certification to the highest criticality classes, and thus can be

designed in a more flexible and resource-efficient way, even if this results in a more complex and indeterministic design.

The TTSoC architecture provides for each of the two classes of application subsystems a dedicated architectural element for integrated resource management, namely the TNA and the RMA [OKESH07]. The RMA computes the resource allocations exclusively for the non safety-critical application subsystems, while the TNA ensures that these resource allocations are not in conflict with the resource allocations for the safety-critical application subsystems. As a part of the TSS (see Figure 4.2), the TNA is assumed to be free of design faults. This assumption is justified by the simple design of the TNA (checking the proposed resource allocations of the RMA for conflicts with safety-critical application subsystems is significantly simpler than the generation of the resource allocations in the RMA). Contrary to the TNA, the RMA is not part of the TSS. The TNA guarantees, that the RMA cannot interfere which the correct operation of the safety-critical subsystems. Thus, the RMA does not have to be certified to the highest criticality levels, and can incorporate arbitrary complex designs and algorithms to facilitate resource-efficient allocation strategies for non-safety-critical application subsystems.

Resource Management Authority (RMA). The RMA is responsible for allocating the available resources to the non safety-critical application subsystems. We consider communication resources, computational resources, and power as the integral resources that have to be managed. In a first step the RMA collects resource allocation requests from the individual micro components. Based on these requests, it calculates an update of the resource allocations exploiting application-specific knowledge (e.g., operational modes of application subsystems and their resource requirements), as well as, system knowledge (e.g., topology of the time-triggered NoC and computational resources of the individual micro components).

The RMA directly accesses the Host CP interfaces of all non safety-critical micro components via the time-triggered NoC, but has no direct access to the configuration parameters that affect resources that are shared with safety-critical application subsystems. The RMA cannot directly update the time-triggered message schedule in the TISSs via the TISS CP interfaces. For resources that are shared with safety-critical subsystems, the RMA can only make reconfiguration proposals to the TNA.

Trusted Network Authority (TNA). The TNA is a guardian for the reconfiguration activities performed by the RMA. Therefore, the TNA checks the proposed time-triggered message schedule for conflicts and ensures that

resource reservations for safety critical application subsystems are not violated. If an erroneous resource schedule is detected, the schedule is rejected and the active schedule is retained. If the new schedule is correct, the TNA updates the configuration accordingly. Since the TNA is part of the TSS it has direct access to the CP interfaces of TISSs.

In addition to protecting static resource reservations for safety-critical application subsystems, the TNA can incorporate certified reconfiguration strategies to dynamically map the functionality of a safety critical application subsystem to a spare micro component in case of a permanent fault of the micro component hosting this functionality.

Thus, the requirement for 100% correctness of the hardware can be relaxated which reduces the cost of manufactoring and testing of SoCs.

4.2.4 Gateways

The TTSoC architecture supports the integration of gateways for accessing off-chip networks like TTP [KG94], FlexRay [Fle05], TTE [KAGS05], and CAN [Bos91] in order to facilitate the interconnection of multiple SoCs for the construction of distributed systems that are suitable for ultra-dependable systems.

Ultra-dependable systems require a maximum failure rate of 10^{-9} critical failures per hour [SWH95], while component failure rates are usually in the order of 10^{-5} to 10^{-6} [PMH98]. Thus, an ultra-dependable system has to be more reliable than any one of its components. This can only be achieved by using fault tolerance mechanisms, based on distributed redundant components, that ensure the continuous operation of the system even in the presence of a bounded number of component failures.

If a time-triggered network like TTP, FlexRay, or TTE is used as an off-chip network, the TDMA scheme of the time-triggered NoC can be synchronized with the TDMA scheme of the off-chip network. The synchronized TDMA schemes facilitate phase alignment of the relayed time-triggered messages on the NoC and the corresponding messages on the time-triggered off-chip network. Consequently, messages can be relayed within a bounded delay with minimum jitter, which depends exclusively on the granularity of the global time base.

Furthermore, the alignment between the time-triggered messages on the NoC and the periodic messages on the time-triggered off-chip network ensures that replicated SoCs perceive each message reception within the same inactivity interval of the global sparse time base [Kop92]. Thus, a consistent temporal order can be established, which is significant for achieving replica determinism [Pol94] as required for active redundancy based on exact voting.

Gateways can provide the SoC with an externally synchronized time base (e.g., GPS), which facilitates the temporal coordination of activities spanning multiple SoCs, like the coordination of actuators that are attached to different SoCs.

4.2.5 Architectural Support for Diagnosis

The TTSoC architecture incorporates a dedicated architectural element for diagnosis, the DU. Diagnosis is done in three distinct phases: *failure/error detection*, *dissemination*, and *analysis*.

Failure Detection. All structural elements of the SoC (i.e., the TISSs, the hosts, the TNA and the DU) perform failure detection in order to indicate faulty and abnormal behavior of micro components.

The TISS incorporates application-independent mechanisms in order to detect failures of the attached host. A watchdog determines whether the host has crashed, and another detection mechanism recognizes violations of message inter-arrival and service times in case of sporadic message transmissions.

The host performs application-specific failure detection. These application-specific failure detection mechanisms are not defined by the architecture, e.g., a host can incorporate a dedicated failure detection mechanism for its local sensors or actuators.

The RMA checks the resource requests from the micro components against predefined quota. Each invalid request is recorded since it might indicate a failure within the requesting micro component.

The DU performs failure detection by means of message classification. For this purpose, the messages of all micro components that have to be diagnosed, are routed to the DU which executes assertions on the syntactic, temporal, and semantic correctness of messages according to the DSoS message classification [GIJ+02]. Thus, the DU performs failure detection at the LIF of the micro components.

Dissemination. Detected failures are reported to the DU via so called *failure indication messages*. A failure indication message includes information concerning the *type* of the occurred failure (e.g., crash failure of a host, illegal resource allocation requests), the *time* of detection w.r.t. to the global time base, and the *location* within the SoC (i.e., the micro component). For the purpose of dissemination, a dedicated time-triggered message is allocated to each architectural element that is capable of delivering failure indications.

Analysis. Based on the gathered failure information, the DU establishes a holistic system view and executes *Out-of-Norm Assertions* (ONAs) to correlate the different failure indication messages in space and time. Correlation in the time domain is possible due to the time stamp (w.r.t. the global time base) that is assigned to each failure detection. Correlation in the space domain exploits the inherent fault isolation mechanisms of the TTSoC architecture which prevents error propagation between micro components. Furthermore, the DU monitors the repeated occurrence of failure indications for the same micro component in order discriminate permanent failures from transient failures.

4.3 Fault Tolerance

The TTSoC architecture employs micro components in TMR configurations in order to tolerate transient faults. TMR is an extension to the concept of triple redundancy which was originally introduced by the computer pioneer John von Neumann [vN56]. The concept of triple redundancy is depicted in Figure 4.8. The three boxes labeled `M` represent identical modules containing any kind of digital equipment (e.g., complete computers or less complex units like adders or single gates). Each of these modules has a single output and is connected to a *majority organ* labeled `V`. The majority organ accepts the input from the three sources and delivers the majority opinion as an output. In this thesis we will call the majority organ a *voter*.

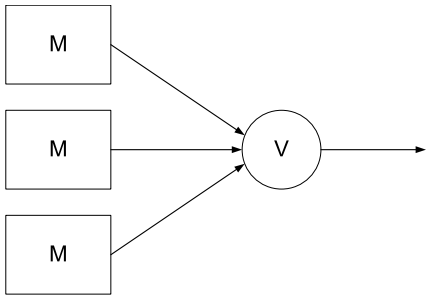

Figure 4.8: Triple Redundancy

Assuming that the three modules fail independently, and the voter does not fail, the reliability of the redundant system R can be described as a function of the reliability of the single modules R_M [LV62]. The reliability R is equal to the sum of the probability that none of the modules fails, and the probability that exactly one of the modules fails. Thus,

$$R = R_M^3 + 3R_M^2(1 - R_M) = 3R_M^2 - 2R_M^3 \tag{4.1}$$

The concept of *Triple Modular Redundancy* (TMR) differs from the original triple redundancy concept by employing three identical voters instead of one

single voter (see Figure 4.9). As long as we assume that the voters are perfect and do not fail, both concepts result in the same reliability of the redundant system [LV62]. When considering the fact, that a voter can fail like any other component in the system, the TMR concept can benefit from the redundant voters.

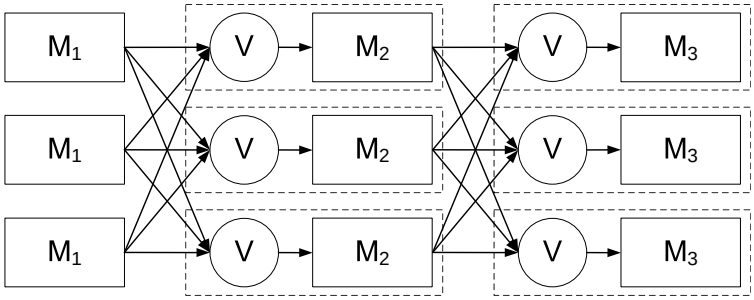

Figure 4.9: Triple Modular Redundancy

We consider the voter at the input of a module and the module itself as a self-contained unit, indicated by the dotted lines in Figure 4.9, which receives the replicated inputs and performs voting by itself without relying on an external voter. We call this behavior *incoming voting* and such a self-contained unit a *replica*. The reliability R_{IV} of a replica with incoming voting is the product of the reliability of the voter R_V and the contained module R_M. Thus,

$$R_{IV} = R_V R_M \tag{4.2}$$

We denote three replicas in a TMR configuration an FTU. Since a replica is a self-contained unit with a single reliability value including the reliability of the internal voter and the internal functional module, the same function used for calculating the reliability in the concept of triple redundancy can be employed for calculating the reliability of an FTU in the TMR concept. Thus, the reliability of an FTU is determined by:

$$\begin{aligned} R_{FTU} &= 3R_{IV}^2 - 2R_{IV}^3 \\ &= 3(R_V R_M)^2 - 2(R_V R_M)^3 \end{aligned} \tag{4.3}$$

With respect to voting, one can differentiate between two kinds of strategies, *exact* voting and *inexact* voting [Kop97].

Exact Voting. Exact voting means that the results of the three replicas are compared bit-by-bit without considering the semantics of the results. Two results are considered equal if they have exactly the same bit pattern. Therefore, exact voting is a generic and application independent concept, which requires only minimal overhead when realized in hardware.

Inexact Voting. In inexact voting, two results are considered equal if they lie within a specific interval determined by the application. In order to reason about the *"sameness"* of the two results, their bit patterns have to be interpreted on the application level. For example, the distance between two composite RGB values can only be calculated, if the structure of the composite value is known. Furthermore, it can be very difficult to define a correctness interval for realistic applications.

Due to the mentioned advantages, exact voting should be preferred over inexact voting. The underlying assumption of exact voting is that the replicas show *replica-deterministic* behavior [Pol96]. Replica determinism requires that all correct replicas produce exactly the same output messages that are at most an interval of d time units apart, as seen by an omniscient outside observer. In a time-triggered system, the replicas are considered to be replica-deterministic if they produce the same output messages at the same global ticks of their local clock [Kop97].

A major source for replica non-determinism can lie in the design and the implementation of the replica itself. Causes for indeterministic behavior at the output of a replica can be non-deterministic program constructs, dynamic scheduling decisions, timeouts, and race conditions in combination with the unavoidable drift of the internal oscillator driving a replica's control signals.

Similarly, replica determinism can be lost, if the replicas observe their input messages in an inconsistent order. Consider the example in Figure 4.10 which depicts the time line of the execution of three replicas (R_1, R_2, and R_3) each receiving three different messages (a, b, and c) from three different senders. We assume that the replicas are event-triggered and process the incoming messages immediately upon reception. We further assume that the sender of Message a has a short transmission latency to Replica R_1, a medium latency to R_2, and a long latency to R_3, that the sender of Message b has a short transmission latency to Replica R_3, a medium latency to R_1 and a long latency to R_2, and that the sender of Message c has a short transmission latency to Replica R_2, a medium latency to R_3, and a long latency to R_1. Due to the different transmission latencies, the replicas do not process the messages in a consistent order. Replica R_1 observes the order a, b, c, while R_2 observes c, a, b, and R_3 observes b, c, a. In general, the operations that are performed upon the reception of a message are not commutative.

Thus, an inconsistent reception order can lead to an inconsistent state in the replicas. To avoid such a situation, *consistent delivery order* has to be guaranteed by the architecture for all messages that are received by the replicas of an FTU.

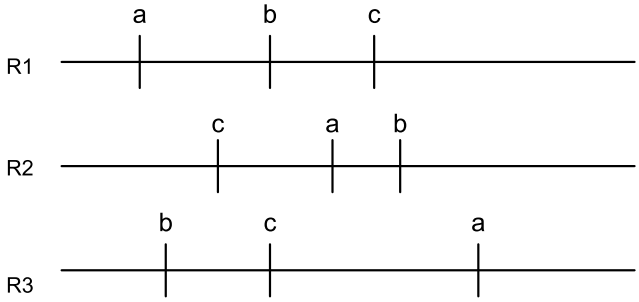

Figure 4.10: Inconsistent Message Order

A TMR-based FTU is a fail-operational component, which means that it continues to deliver a correct service despite the failure of a single replica. Nevertheless, the reliability of an FTU is reduced, if a failed replica remains in an erroneous state. In fact, the reliability of a TMR-based FTU in which one of the three replicas has already failed is actually lower than the reliability of a single replica since the FTU requires both remaining replicas to stay operational. In order to obtain the original reliability of an FTU after the failure of a replica, one of the following actions can be taken.

- In case of a transient fault where the hardware is still operational, a valid state in the replica can be reestablished by adequate recovery mechanisms.
- In case of a permanent hardware fault, the hardware of the replica can be repaired or replaced during the next scheduled maintenance service.
- In case of a permanent hardware fault where a repair action cannot be taken due to temporal or physical constraints (e.g., a faulty IP core within a chip cannot be repaired), the role of the failed replica can be taken over by an equivalent spare component.

In the following, we will explain how the TTSoC architecture realizes TMR-based FTUs at the on-chip and off-chip level. This description will focus on the following categories.

- *Encapsulation.* Common mode failures have to be avoided by *encapsulation* mechanisms that establish a dedicated FCR for each replica. Without encapsulation it cannot be guaranteed that the replicas fail independently, and thus the equations 4.1 and 4.3 are not valid. If a shared communication medium is employed, *error containment* mechanisms (e.g., the bus guardian in TTP [KG94]) are needed to ensure that a fault in a replica cannot disrupt the communication of the other replicas.

- *Replica Determinism.* has to be supported by the architecture to ensure that the replicas of an FTU observe the received messages in a consistent order.
- *Temporal Predictability.* The communication service has to be *temporally predictable* in order to guarantee that the resulting system can meet its deadlines even in peak-load scenarios.
- *Recovery and Repair.* The architecture should provide adequate *recovery* and *repair* mechanisms in order to reestablish the original reliability of an FTU after the failure of a replica.

4.3.1 On-chip TMR

The purpose of on-chip TMR is to increase the reliability of services residing on a single chip.[1] In the case of on-chip TMR, FTUs are constructed by mapping the replicas to individual hosts of micro components residing in the same SoC and communicating via the time-triggered NoC (see Figure 4.11). The following paragraphs describe how the required architectural properties for TMR are met with respect to on-chip TMR.

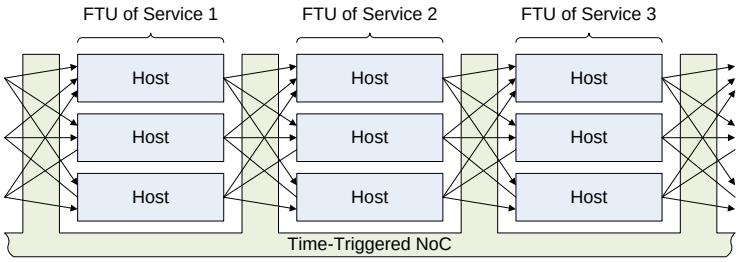

Figure 4.11: On-Chip TMR

Encapsulation. The FCRs for *physical faults* are the individual hosts and the TSS which consists of the NoC, the TISSs, and the TNA. Contrary to an operating system that provides dedicated FCRs for multiple tasks on a single processor by complex memory protection mechanisms and preemptive scheduling strategies, the TTSoC architecture natively provides fault containment by the physical separation of the individual hosts and the TSS.

The independence of the host-FCRs is guaranteed by the fact that hosts can interact with each other exclusively via the exchange of messages over the NoC. There are no other hidden channels (e.g., shared memory) through which a host can interfere with any other host.

[1] For ultra-high reliability, however, off-chip TMR has to be employed (see Section 4.3.2).

The independence of the TSS-FCR is guaranteed by the fact that hosts have no possibility to directly interfere with the operation of the TSS. The TSS transports messages from one host to another host according to a predefined time-triggered message schedule. This schedule can be exclusively configured by the TNA which is itself part of the TSS. Thus, it is guaranteed that a faulty host cannot disrupt the communication among other hosts.

Contrary to fault-tolerant off-chip networks like TTP [KG94]—which are partitioned into multiple FCRs—the TSS is a single atomic FCR. This means that a failure of one of its elements (i.e., the NoC, the TNA, or one of the TISSs) can potentially cause a failure of the entire chip. Since in a typical SoC the die area consumed by the TSS is expected to be relatively small compared to the area consumed by the rest of the chip, we expect the failure rate of the TSS to be relatively low. Therefore, for on-chip TMR we assume that the TSS does not fail during the mission time of the system.

To conclude, the underlying assumptions for on-chip TMR with respect to physical faults are that hosts fail independently and that the TSS does not fail at all during the system's mission time. Considering the fact that a single chip is susceptible to common mode failures caused by disruption of the single power supply, particle induced multi-bit errors, extensive EMI disturbances, or physical damage the assumption coverage for these assumptions will not satisfy the requirements for ultra-high dependable systems. Nevertheless, for many applications with less stringent dependability requirements, on-chip TMR can be a cost-effective alternative for increasing the reliability of systems realized on a single SoC. Furthermore, in safety-critical systems on-chip TMR can be used in conjunction with off-chip TMR to further improve reliability.

With respect to *design faults*, a host constitutes an FCR as long as no piece of the design is used in any other host. If pieces of a design (e.g., library functions or IP cores) are used in a set of hosts, the entire set has to be considered as a single atomic FCR. The TSS is assumed to be free of design faults. It has to be certified at least to the same criticality level as the most critical host in the entire SoC.

Replica Determinism. The TTSoC architecture supports a variety of different network topologies which range from simple shared buses to complex mesh structures featuring multiple channels and concurrent message transfer. In advanced topologies the paths between different sender and receiver pairs can have different length (i.e., they can include a different number of hops), and a message on a short path can be received before a message that has been sent earlier, but over a longer path. Furthermore, multi-cast communication can be temporally asymmetric since also the paths from a single sender to

the individual receivers may have different length. Therefore, messages are potentially received in an inconsistent order as depicted in Figure 4.10.

In the TTSoC architecture, replica determinism is established by exploiting the global time base in conjunction with time-triggered communication and computational schedules. Computational activities are triggered after the last message of a set of input messages has been received by all replicas of an FTU. This instant is a priori known due to the predefined time-triggered schedules. Thus, each replica wakes up at the same global tick and operates on the same set of input messages. The messages in this set are treated as if they had been received simultaneously, which allows neglecting the actual reception order.

Temporal Predictability. The predefined message schedule of the NoC assures that each micro component can use its guaranteed reserved bandwidth independently of the communication activities of the other micro components. Furthermore, the concept of a *pulsed data stream* supports the reservation of a defined bandwidth within a periodically recurring interval, which fits perfectly to TMR in a time-triggered system. A replica of a typical FTU will periodically read the replicated inputs, perform incoming voting, do the application specific processing on the voted input data, and send its output value to the next FTU.

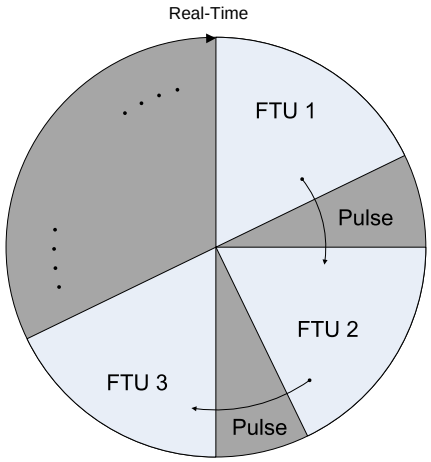

Figure 4.12: Voting on a Circular Time Model

The reactiveness of the overall system can be optimized if the execution of the FTUs and the transmission of the messages are temporally aligned. Figure 4.12 depicts the execution of three FTUs in the periodic time model. The individual replicas of an FTU can execute in parallel since they reside on dedicated micro components. The pulses of the pulsed data streams are perfectly aligned with the execution of the replicas, and thus increase the reactiveness of the system by minimizing the end-to-end latency from the first to the last FTU in the chain.

In addition to minimizing the end-to-end latency, the concept of a pulsed data stream increases the resource efficiency since the communication bandwidth is only reserved for those intervals in which its actually needed.

Recovery and Reintegration. For on-chip TMR, a dedicated component, the *Replica Coordination Unit* (RCU), coordinates the recovery actions of an FTU. The purpose of the RCU is the detection of host failures caused by transient faults and the subsequent reseting of the faulty host. The RCU detects host failures by comparing the redundant computational results in TMR configurations. To accomplish that, the messages of all replicas of an FTU are routed to the RCU where they are compared to each other. If one replica deviates from the other two replicas, the RCU sends a *restart-request message* to the TNA reporting the host on which the erroneous replica resides. Since the TNA has direct access to the TISSs, and the TISSs control the reset lines of the attached hosts, the TNA can restart the corresponding host.

After the restart of a replica, the replica has to build up a valid internal state that is in perfect synchronicity with the state in the other replicas of the FTU. To facilitate state recovery, each replica periodically sends out its internal state via a *history state message*. With the same period, each replica votes over the three history state messages (the own history state message and messages of the other two replicas) and overwrites its internal state with the voted result at the same global tick of its local clock. We call these periodic global ticks the *reintegration points* of an FTU. Since the state of a correct replica after a reintegration point is exclusively determined by its inputs (data inputs plus the history state messages), a replica can be considered as stateless at the reintegration point. Therefore, a restarted component has simply to wait for the next reintegration point to reach a consistent state.

If the replica has failed due to a transient fault, a restart of the corresponding host will be sufficient to reintegrate the replica. The RCU can make use of threshold schemes such as the α-count [BCDGG00] in order to detect both, permanent and intermittent faults. For this purpose, the RCU holds an failure counter for each replica, which is increased each time the replica deviates from the other replicas in an FCR, and which is decreased as time goes on. If the counter reaches a defined upper threshold, the host on which the replica resides is considered to be permanent faulty. In this case, the RCU can send a reconfiguration request to the TNA to remove the faulty host from the FTU and replace it with a spare host.

4.3.2 Off-chip TMR

To achieve ultra-high dependability, the TTSoC architecture supports the construction of FTUs in which the individual replicas are mapped on the hosts of *distinct* SoC components that are interconnected by a fault-tolerant off-chip network like TTP, FlexRay, or Time-triggered Ethernet (see Figure 4.13). Thus, the SoC components form network nodes of a fault tolerant distributed system. Since each replica of an FTU is located on a distinct network node, an FTU will still stay operational despite the failure of an entire node. We will now show how the required architectural properties for TMR are met with respect to off-chip TMR.

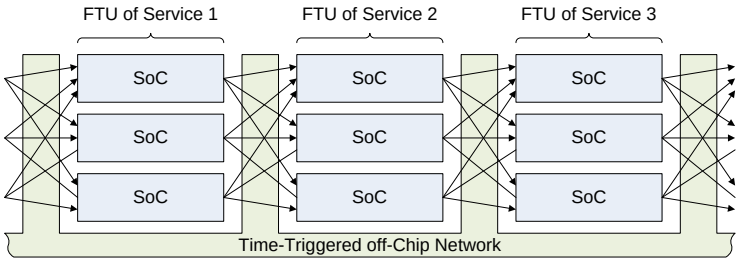

Figure 4.13: Off-Chip TMR

Encapsulation. In ultra-high dependable systems common mode failures have to be considered that can cause an entire chip to fail. Examples are disturbances in the power supply, particle induced multi-bit errors, extensive EMI disturbances, or physical damage of the chip. Thus, single hosts in an SoC can not be regarded as FCRs.

For off-chip TMR we consider an entire SoC as an FCR for physical faults. The coverage of the assumption that the nodes in a distributed system fail independently is much higher than for hosts within a single SoC. Contrary to hosts in an SoC, the network nodes of a distributed system do neither reside on the same die, nor in the same package. They can be physically separated over large distances (e.g., on the opposite sides of a car or an airplane), and can have individual power supplies.

The TTSoC architecture requires that faulty nodes cannot interfere with the correct operation of the off-chip network. Furthermore, the off-chip network has to be able to tolerate internal faults in order to meet the requirements for ultra-high dependability. Therefore, it has to be partitioned into multiple FCRs, which are integrated in a way that the network can deliver a correct communication service despite the failure of a single internal FCR.

An example of an ultra-high dependable network that meets these requirements is TTP [KG94]. TTP provides error containment via so-called bus guardians

which electronically connect each node only during its specified time slot to the shared communication bus. Thus, a node which is violating its temporal specification cannot disrupt the communication among the other nodes. Furthermore, TTP was constructed to tolerate any arbitrary single fault within the network itself, by providing two redundant communication channels (with dedicated bus guardians) forming independent FCRs.

As for on-chip TMR, we consider a host as an FCR for design faults. When pieces of a design are used for a set of hosts, the entire set has to be considered as a single atomic FCR. The off-chip network is considered to be free of design faults (e.g., TTP is certified for the usage in ultra-high dependable systems).

Replica Determinism. As described in Section 4.3.1, replica determinism in the TTSoC architecture is based on a consistent view of the global time. Time-triggered networks like FlexRay, TTP, or TTE provide fault-tolerant clock synchronization to establish a system-wide global time. To facilitate the temporal coordination of hosts residing on different nodes, the *chip-wide* global time in each SoC is synchronized to the *system-wide* global time established by the off-chip network.

Temporal Predictability. Temporal predictability is provided by the combination of the on-chip and the off-chip time-triggered network. Due to the fact that the time in the NoC is synchronized to the system-wide time of the off-chip network, the message schedule of both networks can be aligned which makes the exchange of messages at the on-chip/off-chip gateways temporally predictable.

Recovery and Reintegration. For off-chip TMR there is no central unit that can trigger a recovery action of replicas that are distributed across multiple SoCs. The restart or the migration of a replica can be exclusively triggered by the RCU within the SoC on which the replica resides. The output messages off all three replicas of a distributed FTU have to be routed to the three RCUs of the SoCs where the replicas reside, in order to enable recovery for off-chip TMR. By comparing the output messages, a RCU can decide whether the local replica functions correctly or whether it is affected by a transient or permanent fault. In case of transient fault the replica can be restarted, and in case of a permanent fault of the corresponding host, the replica can be migrated to a spare host on the same SoC.

Thus, from a global point of view, an SoC in a distributed system is a self-checking and self-healing component in which recovery actions are exclusively triggered by the local RCU. The limitation of this approach is that the recovery

actions can only be performed as long as the TSS in the SoC is still operational. The advantage is that a central coordination unit for recovery is not required.

4.3.3 On-chip vs. Off-chip TMR

As described in the previous sections, both options—on-chip and off-chip TMR—satisfy the requirements for TMR with exact voting and temporal predictability. Obvious differences between the two proposed solutions can be observed with respect to performance and end-to-end latency. While on-chip networks can reach bandwidths over 100 GB/s and can have latencies as short as several clock cycles, off-chip networks provide bandwidths in the range of 10 Mbit/s up to 1 Gbit/s with much higher latencies.

Another major difference between the two approaches is the degree of encapsulation. Equation 4.1 and 4.3 assume that the individual replicas fail independently. With respect to the communication channels, the equations are only valid under one of the following assumptions.

- Each outgoing channel is part of the fault-containment region of the replica that drives the channel, and the reliability value of the replica (R_M resp. R_{IV}) includes the reliability of all of its output channels (i.e. channel failures are mapped to the sender, and channels of different senders fail independently).
- Each incoming channel is part of the fault-containment region of the replica that listens on the channel, and the reliability value of the replica (R_M resp. R_{IV}) includes the reliability of its input channels (i.e. channel failures are mapped to the receiver, and channels of different receivers fail independently).
- The communication channels do not fail at all.

In other words, these assumptions require the set of output channels (resp. input channels) of a single sender (resp. receiver) to fail either independently of all other channels or not at all.

For off-chip TMR systems, which are based on a fault-tolerant protocols like TTP, it is known that the coverage of these assumptions meets the requirements for ultra-high dependable systems.

With respect to on-chip TMR the assumption coverage is lower, as the micro components reside on the same die and as the NoC contains no fault tolerance mechanisms against internal faults. Still, for many applications with less stringent dependability requirements on-chip TMR can be a cost-effective alternative.

Tis but thy name that is my enemy.
Thou art thyself, though not a Montague.
What's Montague? it is nor hand, nor foot,
Nor arm, nor face, nor any other part
Belonging to a man. O, be some other name!
What's in a name? that which we call a rose
By any other name would smell as sweet

Shakespeare's Romeo and Juliet

Chapter 5

Naming

Naming is an important aspect in all kinds of computer systems, especially in distributed systems. Nevertheless, this issue is frequently overlooked during the system design phase, which leads to designs with inappropriate high cognitive complexity and negative properties with respect to composability, scalability, and maintainability. This chapter starts by focusing on the most important concepts that have to be considered when designing a naming scheme for a computer system. Subsequent sections deal with the issue of naming in the TTSoC architecture.

5.1 Basic Concepts

A *name* is used to refer to an individual entity within a given context. An entity can be a human, a group of people, a species, a thing, or even an abstract idea, a category or a concept. The process of assigning a name to a given entity is called *naming*. A *naming convention* or *naming scheme* prescribes how to name entities and describes how to refer to named entities [SSA98]. For instance, the archetypical name of a male citizen in the naming convention of the ancient Rome consisted of three parts (*tria nomina*): *praenomen* (given name), *nomen gentile* or *gentilicium* (name of the gens or clan) and *cognomen* (name of a family line within the gens) [Wik07].

The first question to be considered when designing a naming scheme for a distributed system is the following: "*What are the entities that have to be named and what is the appropriate context?*" [1] The answer to this question is not trivial, since the fact whether something is regarded as an entity or as a context depends on the considered level of abstraction. The next key issue

[1] quote taken from a discussion with Hermann Kopetz

is to choose an appropriate name type for the entities at a given abstraction level. The following subsections will focus on important types of names and on the relevant properties that a name can have in a distributed system.

5.1.1 Identifiers and Addresses

A distributed system comprises numerous different entities like hosts, clusters, files, memory regions, processes, network connections, and so on. In order to operate on an entity, it is necessary to access it, via an *access point* which is just another kind of entity in a distributed system [Tv03]. The name of an access point is called an *address* of the entity to which the access point belongs.

An entity can have more than one access point, and thus more than one address. For example, an email account can be viewed as an access point of a person, and the corresponding email address as an address of that person. Of course, a person can have multiple accounts for different purposes (e.g., business and private), and can thus have multiple addresses. The access points and thus the addresses of an entity can change in the course of time (e.g. when people change their job they usually change their business email address).

Another type of name that deserves special attention is an *identifier* or *object identifier (oid)*. An oid is used to distinguish an object from all other objects within a given scope. According to [WdJ92] an oid has to adhere to the following two principles:

Uniqueness Principle: "In any possible state of the world, each relevant object has one and only one oid, which differs from the oid of any other relevant object."

Persistence Principle: "Each relevant object has the same oid across all relevant states of the world. That is, the oid of an object remains invariant under any change of state of the object."

The first principle allows us to pick out an object among other objects even if all objects are in the same state (i.e., even if the objects would be indistinguishable by looking only at their state), since it prevents that any two objects have the same oid at the same time. Furthermore, it facilitates tests for *object equality* and *object difference* by establishing a one-to-one relation between objects and oids (i.e., it assures that an object is referred to by exactly one identifier and thus prevents synonyms). If two oids are equal, they refer to the same object and if they are different they refer to distinct objects.

The second principle guarantees that the *equality* and *difference* of oids does not change in the course of time. If we would allow changes in the oids of

objects, the equality of oids—that have been observed at different points in time—would not necessarily imply that the identified object are identical, nor would different oids imply that the identified objects are distinct.

These two principles might appear obvious, but if we take a closer look we will see that many naming schemes violate at least one of them and therefore cannot be used for oids. In the following we mention two negative examples of non-oids.

- *Mobile phone numbers* do not qualify as oids for persons. The uniqueness principle is violated since a single person can have multiple phone numbers and a single mobile phone can be shared by multiple people. The persistence principle is violated since a person can change or lose its phone number by changing or canceling the contract with the mobile phone provider. Even worse, mobile phone numbers are reused by the providers after a contract has been canceled. (Usually the provider locks phone numbers only for a defined period after a contract has been canceled.)

- *Employee numbers* are very special with respect to oids. They can serve as oids as long as they are only used to identify employee *roles* (e.g., production officer) within a company [WdJ95]. If we would like to use employee numbers to identify the *persons* that play a certain role in a company, we would have to enforce the following conditions. First, nobody can have two jobs with different employee numbers in the same company. And second, whenever a person is re-employed by the same company, the same employee number is given to that person.

5.1.2 Properties of Names

After having elaborated the fundamental differences between addressing and identification, we will now focus on general properties of names and point out how these can either illuminate or confuse system design.

Absolute vs. Relative Names. Names are organized in *namespaces*. An appropriate representation of a namespace is a labeled, directed graph. We call a namespace that is described by a graph with a depth > 1 a *hierarchical* namespace, otherwise the namespace is called *flat*.

A node in the graph that has only outgoing edges and no incoming edges is called a *root node* [Tv03]. A name of an entity can be represented by the sequence of labels of the nodes in a path of this graph. A name is called *absolute* or *fully-qualified*, if the first node in the path is a root node, otherwise

it is called a *relative*. Absolute names denote the same entity in the entire namespace, whereas relative names can have a different meaning in different parts of the namespace. Absolute names are often used when an external reference to a given entity is needed.

The main advantage of a relative name is that they require less centralization with respect to the *naming authority* (we use the term naming authority to denote the part of a system or the group of people that are in charge of assigning names to the entities within a given system). The exclusive use of absolute names requires a centralized naming authority, allocating the names for the entire naming graph [OD83]. Relative names permit decentralizing naming authorities, possibly one for each directory node. The ability to use decentralized naming authorities is a key requirement for every composable architecture.

First of all, it facilitates complexity management by supporting a divide-and-conquer strategy since the names within a given application subsystem can be assigned independently of the names within other subsystems. This property is advantageous for the reuse of subsystems and for the independent development of subsystems by different development teams or third party suppliers. Furthermore, relative names are insensitive to reorganization of the system either during runtime to support dynamic resource management or during the development to incorporate late changes in the design.

Pure vs. Impure Names. Regarding the type of information that is contained in a name one can distinguish *pure* and *impure* names. According to [Nee02] *"a pure name is nothing but a bit-pattern that is an identifier, and is only used for comparing for identity with other such bit-patterns."* This means that a pure name does not contain any information concerning the object it identifies. In contrast, an impure name yields information about the identified object by examination of the name itself.

An example of an impure name is the Internet domain name of our university: *www.tuwien.ac.at*. By examining the name, we can find out that the domain is located in Austria and that it is an academic domain. Furthermore, the word *tuwien* suggests that the domain name represents the University of Technology, Vienna.

Since impure names contain information concerning the objects they identify, they carry commitments that have to be honored in order to retain the validity of the names. Thus, every information that is included within a name becomes an invariant that the named objected must fulfill during its lifetime [SSA98].

Pure names do not commit one to anything and thus, the name remains valid regardless of changes in the state of the identified object. On the other hand,

impure names allow optimizations since one can get the included information immediately without extra stages of indirection.

Location Dependent vs. Location Independent Names. As mentioned above, impure names contain information about the objects to which they refer. A special case of an impure name is a name that includes information about the physical address of the referred object. Such a name is called *location dependent*. If a name of an object is independent of the object's address, it is called *location independent* [Tv03].

Location independent names stay valid if the named entities are relocated (e.g., in the case of reconfiguration), whereas location dependent names would become invalid. Therefore, location dependent names should only be used for objects that will not change their physical address in their entire lifetime.

5.2 Naming in the TTSoC Architecture

The TTSoC architecture differentiates between the *platform-independent structure* and the *physical structure* of an embedded distributed system. The platform-independent structure abstracts from the physical platform and describes the system from a logical point of view. It consists of the behavioral specifications of the logical entities within the system and describes the interaction patterns among these entities, including temporal constraints. The physical structure describes the system from the physical point of view, specifying the physical components and how these components are interconnected.

With respect to naming, these two aspects of a system pose different and contradicting requirements (e.g., logical entities are inherently location independent, while a physical core is strictly bound to a specific chip). Therefore, the TTSoC architecture provides two dedicated naming conventions, one for the platform-independent structure and one for the physical system structure.

The differentiation between these two structures is a key aspect for *model-based development* where a platform-independent system model is mapped on a specific hardware platform during the design phase, and for *dynamic resource management* where the allocation of physical resources (e.g., communication and computational resources) to logical system functions is adapted during runtime.

5.2.1 The Model-Based Development Process

This section gives a short overview of the model-based development process employed in the TTSoC architecture. The central concept in the development process is the *Fully-Specified Interface Model* (FIM) (see Figure 5.1). The FIM describes the *platform-independent structure* of the distributed system. It includes the full syntactic and semantic specification of the subsystem's interfaces and their temporal constraints, but abstracts from the mapping of subsystems to the physical resources of the platform. Thus, it abstracts also from the concrete hardware on which the subsystems will be executed.

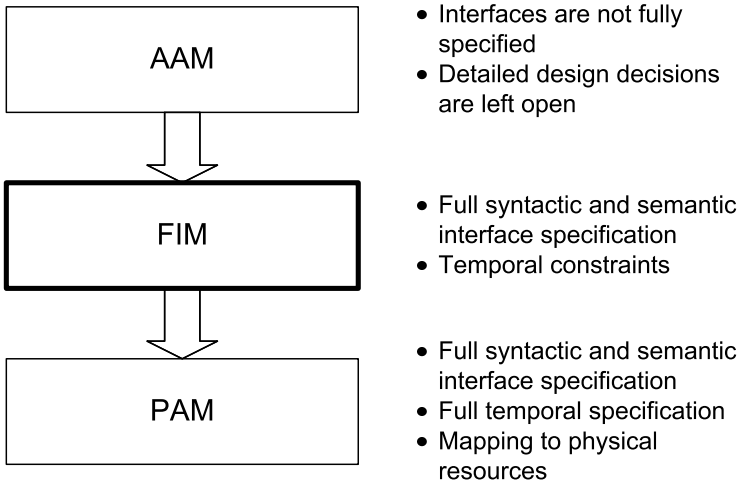

Figure 5.1: Model-Based Development Process

The *Abstract Application Model* (AAM) is a more abstract system representation than the FIM. In the AAM the interfaces of the individual subsystems are only specified on an abstract level, and thus, some design decisions are still left open (e.g., the selection of an adequate encryption method to achieve the desired security properties of a communication channel). The AAM is not specific to the TTSoC architecture and can exist in multiple variants for different application domains (e.g., a developer of a consumer electronic device will most likely use a different kind of AAM than a developer of a safety-critical real-time system). Since the AAM is not specific to the TTSoC architecture, the definition of an AAM is not in the focus of this thesis.

The *Physical Allocation Model* (PAM) is a more concrete system representation than the FIM. It describes the mapping of the FIM to the physical system structure. Thus, the definition of the individual subsystems in the PAM is specific to the concrete hardware on which they are executed. The semantic and syntactic interface specification of the subsystems in the PAM are exactly the same as in the FIM. The temporal properties of the subsystem's interfaces

in the PAM are fully specified and satisfy the temporal constraints defined in the FIM.

Table 5.1 summarizes the properties of the models that are involved in the model-based development approach.

Model	Semantic and Syntactic Properties	Physical Location	Temporal Properties
AAM	partly specified	unspecified	constraints
FIM	fully specified	unspecified	refined constraints
PAM	fully specified	fully specified	fully specified

Table 5.1: Overview of the System Representations

Since the FIM and the AAM abstract from the concrete hardware platform, we regard them as a *Platform Independent Models* (PIMs) as defined by the *Model Driven Architecture* (MDA) [OMG03]. The MDA defines *platform independence* as a *"quality, which a model may exhibit. This is the quality that the model is independent of the features of a platform of any particular type. Like most qualities, platform independence is a matter of degree. ... A platform independent view shows that parts of the complete specification that does not change from one platform to another. ... A platform independent model is a view of a system from the platform independent viewpoint." [OMG03]*

According to the definition of the MDA, the PAM is a *Platform Specific Model* (PSM) since it is linked to a specific technological platform.

The following sections describe the platform-independent structure (represented in the FIM) and the physical structure of a distributed system in the TTSoC architecture. Furthermore, it describes how the two structures can be mapped to each other in the PAM. At the end of this chapter, the properties of the proposed naming scheme are evaluated with respect to the concepts introduced in Section 5.1.

5.2.2 Platform-Independent System Structure

The platform-independent system structure in the TTSoC architecture is an extension of the structure in the DECOS integrated architecture [OPHES06]. In the TTSoC architecture, a real-time computer system is divided into a set of nearly independent application subsystems, with each application subsystem providing a part of the overall computer system's functionality. We call such an application subsystem a *Distributed Application Subsystem* (DAS) since it will be most likely distributed on multiple micro components, potentially residing

on different SoCs. Each DAS provides a meaningful service to the user (e.g., a multi media DAS or a steer-by-wire DAS in a car).

A DAS is further decomposed into smaller units called jobs which are atomic units with respect to the allocation to micro components (i.e., a single job cannot be distributed on multiple micro components). Also, a job is a unit of fault-containment for design faults.

In the TTSoC architecture the platform-independent structure of a distributed system is expressed by the *Fully-Specified Interface Model* (FIM). The FIM describes the functionality and the interaction patterns of the individual jobs of a system by a behavioral specification, including temporal constraints. It does not include any information about the micro components on which the jobs will be executed and abstracts from micro component specific implementation details of the jobs (e.g., a micro component can be realized as a special purpose microcontroller, as an FPGA or as a special purpose hardware IP block). The *FIM meta model* defines the rules and constructs according to which a FIM is created.

The TTSoC architecture defines two different types of FIMs to describe a system at two different levels of abstraction, the *Uniform FIM* (UFIM) and the *Macro FIM* (MFIM). The UFIM is a uniform representation of the FIM that is independent of any application domain. It describes the system at the level of the UNI. This means that, with respect to the interface specification of the jobs, the *UFIM meta model* defines exclusively constructs that refer to the communication services that are natively provided by the UNI (e.g., unidirectional communication channels). The specification of a job in the UFIM serves as a contract between the *system integrator* and the *job developer* and can be used for conformance testing.

The MFIM is a high-level representation of the FIM. It facilitates the modeling of DASs at a higher level of abstraction than the UFIM, by providing macros that translate high-level constructs into constructs supported in the UFIM. Thus, the interface specification of the jobs in the MFIM can rely on higher-level domain-specific services like voted channels for fault tolerance, encrypted channels for security, or bidirectional channels for request/reply transactions. The *MFIM meta model* can exist in multiple variations supporting different sets of domain-specific services. For each MFIM meta model, a set of transformation rules has to be specified that define the transformation of a MFIM to an equivalent UFIM as shown in Figure 5.2.

The Uniform FIM

In the *Uniform FIM* (UFIM), jobs interact with each other exclusively by the exchange of elementary messages via *UFIM-channels*. The term UFIM-channel

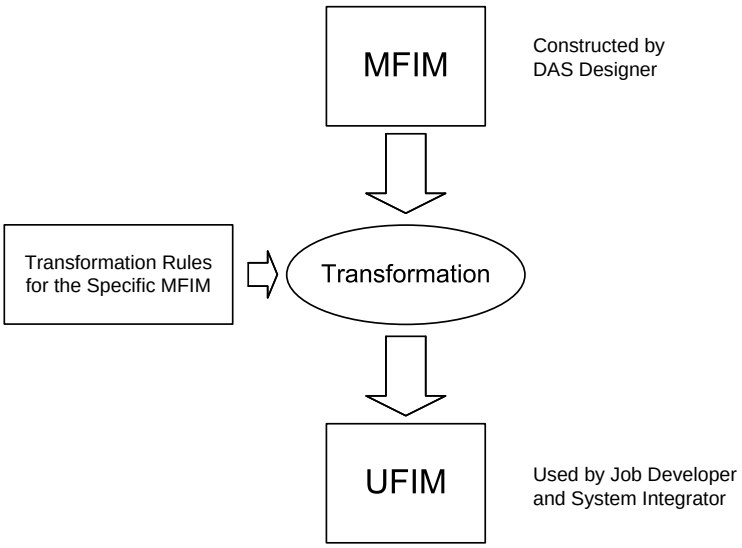

Figure 5.2: UFIM vs. MFIM

denotes an encapsulated unidirectional communication channel that transports messages under predefined temporal constraints (e.g., latency, period, absolute phase offset to the start of the period, or relative phase offset to another channel) from a single source job to one or more destination jobs. It was decided to restrict the communication in the UFIM to unidirectional channels in order to avoid any back-error propagation from a receiver to the sender at the level of UNI. If required, bidirectional channels can be provided at higher abstraction levels in the MFIM as described in the following section. UFIM-channels are not restricted to the scope of a single SoC. They can cross chip boundaries via gateways and can interconnect jobs that are located on different SoCs. The endpoints of an UFIM-channel are called *UFIM-ports*. A job can be attached to multiple UFIM-channels and can thus have multiple UFIM-ports.

UFIM-ports are identified by the fully-qualified *UFIM-port identifier*. The context of the namespace for UFIM-ports is the entire considered system. Thus, a fully-qualified UFIM-port identifier uniquely identifies an UFIM-port within the logical structure of the entire system. The fully-qualified UFIM-port identifier is formed by the concatenation of the following sub identifiers:

DAS Context. Uniquely identifies the context of the DAS (e.g., a car).

DAS Identifier. Uniquely identifies the DAS within the DAS context.

Local Job Identifier. Uniquely identifies the job in the scope of a DAS.

Local UFIM-port Identifier. Uniquely identifies the UFIM-port in the scope of a job.

The fully-qualified UFIM-port identifier is location independent, which means that it does not contain information about the physical location of the referred UFIM-port. It exactly identifies the position in the logical structure of the system with respect to DASs and jobs, but does not tell anything about the physical chip or micro component where the UFIM-port is located. Thus, the fully-qualified UFIM-port identifier stays stable if a job (and its UFIM-ports) is migrated from one micro component to another.

A UFIM-port is devoted to a specific part of a service provided or consumed by a job and is associated with a specific message (e.g. the periodic dissemination of the cruising speed of car). All the UFIM-ports of a job form the job's *Linking Interface* (LIF). The operational properties like syntax and temporal constraints and the meta-level properties (i.e., the semantics) of every UFIM-port of a job are captured in the job's *LIF specification* [KS03]. *In fact, the UFIM of a job is exactly the LIF specification of that job.* It describes the temporally constraint behavior of the job that can be observed at its LIF, but abstracts from its implementation. Thus, a model of a given job in the UFIM can serve as a reference for *conformance testing* of an implementation of that job.

The Macro FIM

A *Macro FIM* (MFIM) facilitates the description of DASs on a higher abstraction level than the UFIM. In addition to the modeling constructs provided by the UFIM meta-model (e.g., UFIM-channels), the MFIM meta-model provides macros that translate higher-level domain-specific constructs into constructs supported in the UFIM. Usually, a MFIM meta-model is supplied together with the appropriate middleware modules contained in the frontend (see Figure 4.3) that provide these services.

Jobs in the MFIM interact with each other by the exchange of messages via *MFIM-channels*. The endpoints of an MFIM-channel are called *MFIM-ports*. Contrary to UFIM-channels, MFIM-channels are not restricted to be unidirectional. An MFIM-channel realizes a *protocol abstraction* that is defined by the chosen MFIM meta model. An example for such a protocol abstraction would be a bidirectional *request/reply* channel in the MFIM. At the level of the UFIM, such a channel would be represented as two independent unidirectional channels, one for the request and one for the reply. At the level of the MFIM, the DAS designer can simply use the macro representing the bidirectional request/reply channel and can abstract from its concrete realization.

Figure 5.3 depicts an exemplary MFIM that provides protocol abstractions for fault-tolerant and secure communication channels and fault-tolerant jobs. Fault-tolerant channels and jobs are marked with `FT` and secure channels are

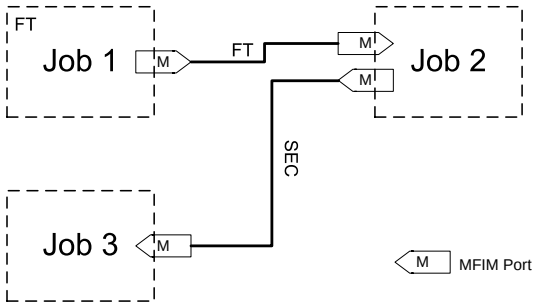

Figure 5.3: Exemplary DAS in a MFIM

marked with `SEC`. `Job 1` is a fault-tolerant job and sends messages via a unidirectional fault-tolerant channel to `Job 2`, while `Job 2` sends messages over a unidirectional encrypted channel to `Job 3`.

Transformation of the MFIM

As mentioned above, each MFIM meta model implies a set of transformation rules to transform a MFIM to the equivalent UFIM. Figure 5.4 shows the UFIM resulting from a transformation of the MFIM in Figure 5.3. The fault-tolerant job `Job 1` of the MFIM has been transformed to three replicas to form a *fault-tolerant unit* in a TMR configuration.

On the abstraction level of the UFIM, these replicas are considered as three independent jobs since there is no notion of fault-tolerance in the UFIM meta model. Also `Job 2` and `Job 3` have been transformed. The fault-tolerant MFIM-port of `Job 2` has been wrapped by a model of a voter module which performs incoming voting on the three UFIM-ports associated to the replicas of `Job 1`. The MFIM-port of the encrypted channel at `Job 2` has been wrapped by a model of an encryption module while the MFIM-port at `Job 3` has been wrapped by a model of a decryption module. The models for the voter, the encryption, and the decryption module are exclusively described with constructs defined in the UFIM meta model and can thus, be interpreted on UFIM level.

The transformation process results in internally structured models of the jobs in the UFIM (e.g., `Job 2` consists of the job described in the MFIM and the models of the wrapper modules that translate the high-level semantics of the MFIM-ports to the semantics of the UFIM-ports). *This internal structure only concerns the information contained in the UFIM model of the job. It does not commit the implementation of a job to anything.*

Since the UFIM model of a job is in fact the job's LIF specification, the LIF specification is also structured. A structured LIF specification reduces the cognitive complexity of a LIF which is a key aspect during system integration.

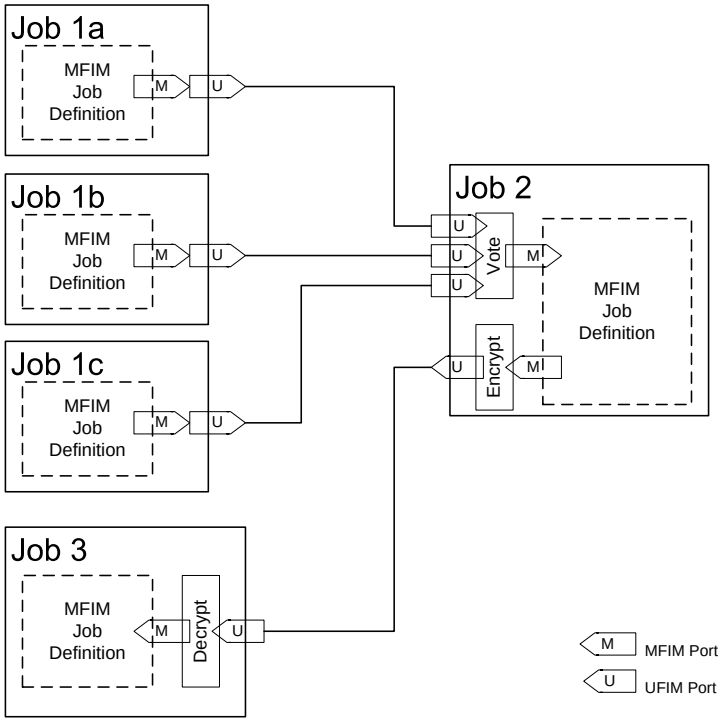

Figure 5.4: MFIM Transformed to the UFIM

Furthermore, a structured LIF specification is relevant for the diagnostic unit which performs diagnosis by observing the LIFs of the jobs of concern. If the LIF specification is not structured, it is impossible to interpret the exchanged messages on an abstraction level higher than the UNI (e.g., in a transaction based system the information that two UFIM-ports are closely related to each other because they form a transaction based request/reply port on a higher level of abstraction would be lost).

5.2.3 Physical System Structure

Regarding the physical structure of a distributed system designed according to the TTSoC architecture, a system consists of one or more clusters which consist of one or more SoCs which again consist of multiple micro components.

The micro components within a single SoC communicate with each other via *SoC-channels*. The term SoC-channel denotes an encapsulated unidirectional communication channel that transports messages at predefined points in time from a single source micro component to one or more destination micro components within the same SoC. SoC-channels cannot cross the boundaries of a single SoC. The endpoints of a SoC-channel are called *SoC-ports*. A micro com-

ponent can be attached to multiple SoC-channels and can thus have multiple SoC-ports.

SoC-ports are identified by the fully-qualified *SoC-port identifier*. The context of the namespace for SoC-ports is a single SoC. Thus, each fully-qualified SoC-port identifier uniquely identifies a SoC-port within a single chip. The fully-qualified SoC-port identifier is formed by the concatenation of the following sub identifiers.

Micro Component Identifier. Uniquely identifies the micro component within the SoC

Local SoC-port Identifier. Uniquely identifies the SoC-port within the micro component

Contrary to the fully-qualified UFIM-port identifier, the fully-qualified SoC-port identifier contains information about the physical location of the referred SoC-port. A good analogy for an fully-qualified SoC-port identifier is the address of a department of a company, where the micro component identifier stands for the address of the company building and the local SoC-port identifier for the post office box of the department. An important difference to the fully-qualified UFIM-port identifier is that the fully-qualified SoC-port identifier is not associated with any semantic information about the messages exchanged over the SoC-port.

The SoC-channels (i.e., the connections between the SoC-ports) are established by the TNA [OKESH07]. From an abstract point of view, the TNA acts like an operator in a traditional telephone exchange system, were the local terminations of telephone lines where connected by patch chords to establish a connection.

The interconnection between multiple SoCs is established via *Gateway Channels* (G-channels). A G-channel is a unidirectional communication channel that transports messages from a single source SoC to one or more destination SoCs. The endpoints of a G-channel are called *Gateway Ports* (G-ports). A SoC can be attached to multiple G-channels and can thus have multiple G-ports. A G-port is identified by the fully-qualified G-port identifier which is formed by the following elements.

Cluster Identifier. Uniquely identifies the cluster within the entire system.

SoC Component Identifier. Uniquely identifies the SoC within the cluster.

Local G-port Identifier. Uniquely identifies the G-port on an SoC.

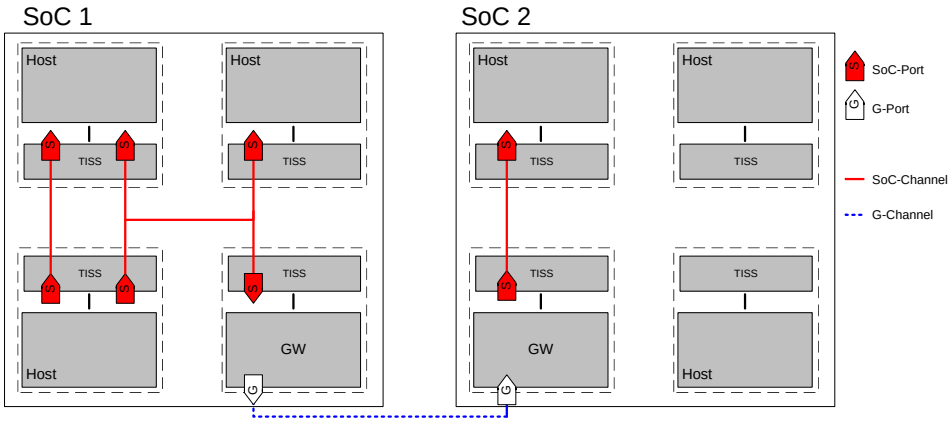

Figure 5.5: SoC-channels and G-channels

A G-channel can be implemented on top of different underlying networks. The underlying network determines the properties of a G-channel with respect to temporal determinism, latency, bandwidth, dependability, and security. The gateways in the micro components map the G-port identifiers to the corresponding names in the namespace of the underlying network (e.g., if TCP/IP is used as an underlying protocol to interconnect SoCs via the Internet, each SoC could be assigned an IP address, and each local G-port identifier could be assigned a TCP/IP port number).

Figure 5.5 depicts an example cluster comprising two SoCs. The SoC-channels interconnect the micro components within a single SoCs, while a G-channel interconnects the gateways of the two SoCs.

5.2.4 FIM-to-PAM Transformation

This section shows how the platform-independent structure of a distributed system is mapped to individual SoCs, micro components, SoC-channels and G-channels in the physical system structure.

The Physical Allocation Model

Before a UFIM of a job can be executed on a given micro component, it has to be transformed to a *Physical Allocation Model* (PAM). Contrary to the UFIM, the PAM is tailored to the specific characteristics of the micro component on which a job should be executed. Nevertheless, the semantic and syntactic properties of a job's LIF in the PAM are exactly the same as in the UFIM. The temporal properties of a job's LIF in the PAM are fully specified and satisfy the temporal constraints defined in the UFIM.

We illustrate the relation between the UFIM and the PAM by the example of the UFIM of `Job 3` in Figure 5.4. The purpose of this job is to do some kind of processing of incoming encrypted messages. The UFIM of the job is logically structured into a model of a decryption module and a functional job that does the actual data processing.

Figure 5.6 depicts three exemplary micro component-specific transformations of the UFIM of the job to an equivalent PAM:

- In the first case the microcontroller on which the job should be mapped is a simple general-purpose microcontroller. In this case, the PAM of the job can be described by a piece of C code representing the M-FIM definition of the job and a library that provides the decryption functionality.
- In the second case the micro component consists of a general-purpose microcontroller and an appropriate frontend that provides the decoder functionality in hardware. In this case, the PAM of the job can be defined by a piece of C code for the M-FIM definition of the job and the configuration parameters for the frontend.
- In the third case the micro component is an FPGA. Here the PAM of the job can be described by an FPGA configuration bit stream that includes both the M-FIM definition of the job and the decoder functionality.

Instantiating UFIM-channels

When a job is mapped to a micro component, an SoC-port that is adequate with respect to the message length and the temporal properties, has to be instantiated on the TISS of that micro component for each UFIM-port that is defined in the job's LIF. The host incorporates a data structure that holds the binding of the UFIM-ports to the SoC-ports (the binding of UFIM-ports to SoC-ports is established by the RMA [OKESH07]).

If all jobs that are attached to a UFIM-channel are mapped to micro components in the same SoC, the UFIM-channel is mapped to a single SoC-channel. If the jobs that are attached to an UFIM-channel are distributed over several SoCs, the UFIM-channel is mapped to multiple SoC-channels and one or more G-channels.

Figure 5.7 depicts an example with two UFIM-channels. One UFIM-channel interconnects only jobs within a single micro component and is thus mapped to a single SoC-channel. The other UFIM-channel crosses the chip boundaries and thus needs two SoC-channels, one for each SoC, and one G-channel. At the gateway, the G-ports have to be mapped to the corresponding SoC-ports.

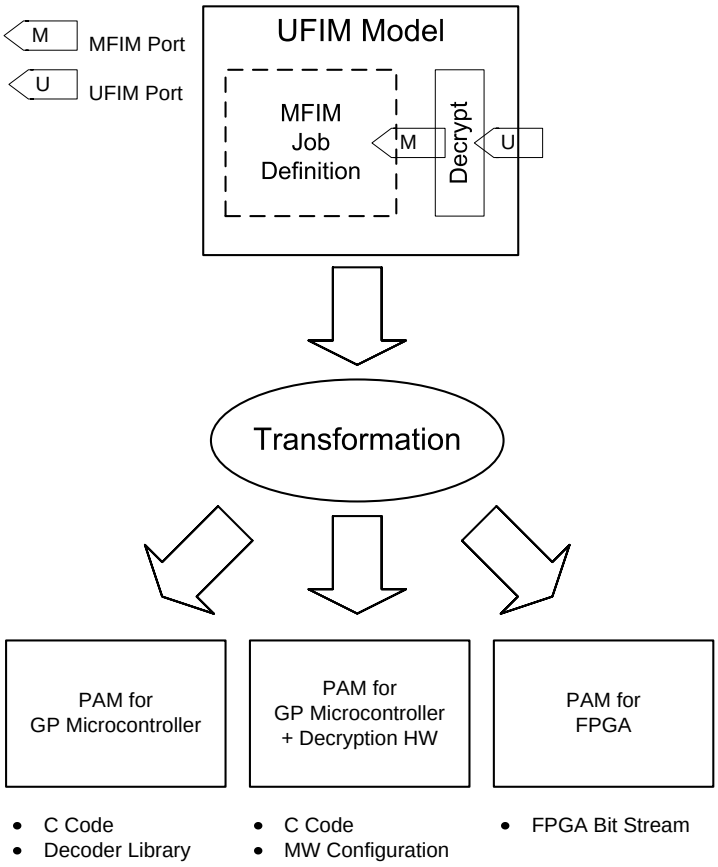

Figure 5.6: UFIM to PAM Transformation - Jobs

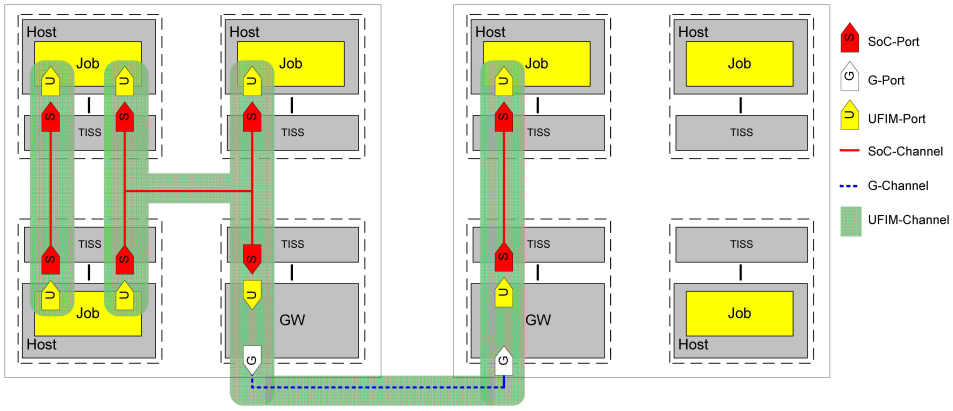

Figure 5.7: UFIM to PAM Transformation - Channels

5.3 Discussion

The proposed naming scheme is designed for complexity management, independent development, design reuse, and dynamic resource management.

One key aspect is the hierarchical system structure. Since each DAS within a distributed system has its own dedicated namespace, DASs can independently be developed by different suppliers without the need for a *central naming authority* coordinating the naming process for the entire system.

The naming scheme enables to look at the system from different abstraction levels:

System Level. The system integrator needs a system-wide view to be able to map the jobs to micro components and the UFIM-channels to SoC-channels and G-channels. Therefore, he will always use the fully-qualified identifiers for UFIM-ports, SoC-ports, and G-ports in order to be able to distinguish between UFIM-ports of different jobs, between jobs of different DASs, between SoC-ports of different micro components, and between micro components of different SoCs.

DAS Level. When a designer of a single DAS specifies an UFIM-channel, he will identify an UFIM-port only by the local job identifier and the local port identifier. By omitting the DAS identifier and the DAS context identifier in the design of the DAS (i.e., by using relative names), the DAS can be instantiated multiple times in the same system or it can be reused across different systems.

Job Level. The designer of job will refer in its design exclusively to the local port identifier of the UFIM-ports. This enables multiple instantiations of a job in the same DAS (e.g., for the construction of fault tolerant units in a TMR configuration) or the reuse of a job design across different DASs.

Another major aspect of the proposed naming scheme is that each identifier contains only the information about the identified entities that will be stable throughout the entire lifetime of the system. If an identifier would include information about the variable part of the state of the identified entity, the identifier would become invalid as soon as the respective part of the state changes. Therefore, the naming scheme is based on the following principles.

- The platform-independent system structure and the physical system structure have separated namespaces with dedicated naming conventions. The strict separation of the platform-independent structure and the physical structure ensures that the uniqueness and persistence principles of identifiers, as defined in Section 5.1.1, are not affected by dynamic resource management.
- The identifiers in the platform-independent system structure are *location independent*, which means that they do not contain information about

the physical location of the referred entities (i.e. DASs, jobs, or UFIM-ports). They identify just the *invariant* position of an entity in the logical structure of the system, but do not say anything about the physical chip or micro component where the given entity is located. Therefore, a logical identifier remains valid if the physical location of the identified entity changes.

- The identifiers in the physical system structure are *location dependent* (e.g., a micro component identifier specifies a physical location in an SoC component). Contrary to the identifiers in the logical system structure, the identifiers in the physical system structure do not contain any *semantic* information about the system (e.g., a SoC-port is agnostic to the semantics of the messages that are exchanged via that port).

Finally, the proposed approach supports multiple DASs with dedicated domain-specific naming schemes in parallel by supporting the transformation of multiple MFIMs, each with a dedicated namespace, onto the UFIM. The uniform namespace of the UFIM ensures that the heterogeneity of the MFIMs does not affect the integration of the system.

Chapter 6

The Trusted Interface Subsystem

The *Trusted Interface Subsystem* (TISS) controls the access to the time-triggered NoC and provides the UNI to the host. The design and the implementation of the TISS must facilitate feasible validation and certification up to the highest criticality levels, since the TISS is replicated in each micro component. Being part of the *Trusted Subsystem* (TSS), the TISS is considered to be free of design faults.

This chapter describes the core platform services, which are provided at the UNI and show how the TISS establishes encapsulation with respect to the communication infrastructure.[1] Furthermore, we focus on the TISS's capabilities with respect to dynamic resource management.

6.1 Communication Service

As already mentioned in Chapter 4, one major objective of the TTSoC architecture is to facilitate independent development of application subsystems by the use of encapsulation mechanisms that prevent any unintended interference between subsystems. On the micro component level, encapsulation is natively achieved by the physical separation of the individual micro components. On the next higher level, where application subsystems are formed by multiple micro components interacting which each by exchanging messages, encapsulation is required with respect to the communication infrastructure that interconnects these micro components. For this purpose, the TTSoC architecture provides *encapsulated communication channels*.

[1] The detailed memory layout of the UNI is described in Chapter 7.

6.1.1 Encapsulated Communication Channels

The term *encapsulated communication channel* denotes a unidirectional channel that transports messages at predefined points in time from a single source to one or more destinations. The endpoints of an encapsulated communication channel are called *ports*.[2] We distinguish between output ports which are located at the source—where messages are produced—and input ports which are located at the destinations where messages are consumed. A single micro component can be attached to multiple encapsulated communication channels, and thus, can have multiple input and output ports.

The topology of an encapsulated communication channel is defined by the number of destinations (i.e., the number of input ports) and by the assignment of the source and the destinations to specific micro components. Since the number of destinations of an encapsulated communication channel is variable, *singlecast*, *multicast*, and *broadcast* topologies are supported. Figure 6.1 depicts three exemplary encapsulated communication channels, each having a different type of topology.

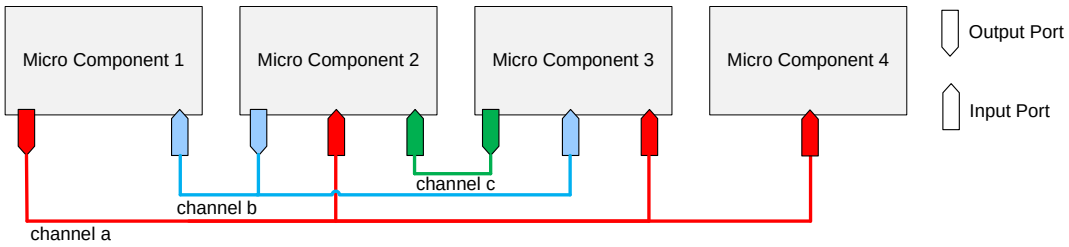

Figure 6.1: Broadcast (a), Multicast (b), and Singlecast Topology (c)

The TTSoC architecture ensures *temporal and spatial partitioning* with respect to encapsulated communication channels in order to prevent any unintended interference between application subsystems. 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 only micro component that can send messages over a given encapsulated communication channel is the micro component that 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).

Encapsulation is established by the TISS, which acts as a guardian for the shared time-triggered NoC by accessing it exclusively at a priori known points in

[2]The ports provided at the TISS are actually SoC-ports as defined in Section 5.2.3 (the TISS is agnostic to the ports in the FIM). For the sake of fluency, we will use the word *port* instead of SoC-port in this chapter.

time according to the TDMA scheme. The implementation of the TISS ensures that the host cannot alter the internal time-triggered schedule of the TISS in order to guarantee that the encapsulation properties of the communication service are not violated in the presence of a design fault or a hardware fault within the host. As already mentioned in Chapter 4 the TISS itself is part of the TSS and is considered to be free of design faults.

6.1.2 Interface to the Communication Service

The communication service of the TISS is accessed via the UNI. A key aspect of the UNI is to establish platform independence by abstracting from the actual implementation of the employed NoC (e.g., with respect to arbitration, encoding, routing, fragmentation, and reliability mechanisms). Looking at the *ISO-OSI reference model* [Ros90], the *transport layer* is the first layer where the offered services are independent of the implementation of the underlying network, while the upper layers become more and more application oriented. Therefore, we decided to position the UNI at the transport layer and to realize optional domain-specific higher-level services in the *frontend* within the host. With respect to the *waistline architecture*, this design results in a waist (i.e., the UNI) that abstracts from the implementation of the network without being "wider" than necessary.

A host accesses an encapsulated communication channel via the corresponding port. As mentioned above, we differentiate 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. Two fundamental types of ports are provided in order to satisfy the requirements of a wide range of application domains: *state ports* and *event ports*.

State Ports

We call a message a *state message*, if its value results from the *observation* of a state variable, i.e., a RT-entity [Kop97]. A state message contains always the complete state of the state variable, and is thus inherently idempotent. Following *state semantics*, a new arriving version of a state message overwrites an old version and exactly-once processing is not required. We call this behavior *update-in-place*.

State ports are used for the periodic transmission of messages with state semantics. Due to the update-in-place strategy, a state port holds only a single state message at a time.

In order to ensure that only consistent data is transmitted and received over the NoC, explicit synchronization mechanisms are provided for both, input state ports and output state ports that coordinate the update and read operations of the host and the TISS.

Input State Ports. For input state ports, we employ the *Non-Blocking Write Protocol* (NBW) [Kop97] to detect 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. Therefore, each input state port is associated with a sequencer which is exclusively written by the TISS and read by the host (see Figure 6.2, and pseudocode listing 6.1). At start-up the sequencer is initialized to zero. The TISS increments the sequencer each time before it starts updating the contents of the port. After the TISS has finished updating the contents of the port, it increments the sequencer again. The host starts each read access to an input state port by checking the port's sequencer. If the value of the sequencer is odd, the host retries the read access immediately because it knows that an update operation by the TISS is in progress. If the value is even, it continues the read operation. At the end of the read operation the host checks, whether the sequencer has been changed by the TISS during the read operation. If this is the case, the host knows that it has acquired inconsistent data and retries the read operation again, until it succeeds to read an uncorrupted version of the message within the port.

Listing 6.1: Synchronization for Input State Ports

```
Initialization:
  Seq := 0

TISS:
  Seq := Seq + 1
  <update port data>
  Seq := Seq + 1

Host:
  REPEAT
    REPEAT
      Seq_begin := Seq
    UNTIL (Seq_begin is even)
    <read port data>
  UNTIL (Seq_begin == Seq)
```

Hosts that are perfectly synchronized with the global time can access input state ports by *implicit synchronization* and ignore the NBW sequencer field which results in a higher performance for read accesses. 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 TISS in a way that avoids conflicts.

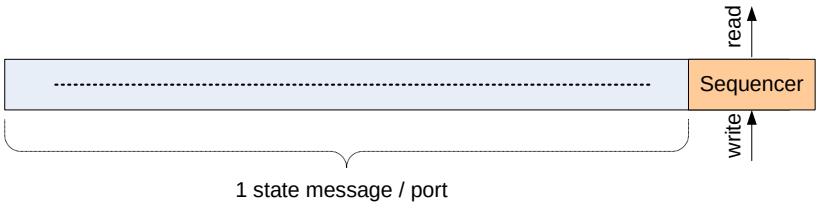

Figure 6.2: Input State Port

Output State Ports. The NBW protocol is inadequate for output state ports, since the TISS has to transmit messages over the time-triggered NoC with minimal jitter and thus, cannot afford a retry of a read access to an output state port, when the read access resulted in inconsistent data due to a concurrent update by the host.

For this reason, output state ports are synchronized by the use of a double buffer mechanism, as depicted in Figure 6.3, with the host alternately updating one of the buffers while the TISS accesses the other buffer which contains consistent data. Synchronization is established by the use of two synchronization fields, the *valid* field and the *transmit* field (see pseudocode listing 6.2). The valid field is exclusively written by host. It indicates which of the two buffers currently contains valid data. In contrast to the valid field, the transmit field is exclusively written by the TISS. It specifies which of the two buffers is currently used for message transmission. The host is only allowed to update the content of a buffer if the valid and the transmit field point to the other buffer.

At start-up, both fields are initialized to zero (the value zero denotes the first buffer). Since both fields are pointing to the first buffer, the host is allowed to update the second buffer. After it has finished updating the second buffer, it sets the valid field to one (the value one denotes the second buffer). At each periodic instant in time at which the message associated to the port has to be transmitted, the TISS checks the valid field and uses the corresponding buffer for the transmission. The purpose of the transmit flag is to prevent that a scenario like the following can occur: The valid field points to the first buffer and the TISS transmits the contents of the first buffer. While the TISS is transmitting the contents of the first buffer, the host updates the second buffer and sets the valid field to point to the second buffer. Consider the case that the TISS has not yet finished transmitting the contents of the first buffer. If the host would now only look a the valid field (which points to the second buffer), it could start updating the first buffer which would result in a transmission of inconsistent data.

In order to prevent such a scenario, the TISS sets the transmit flag to the value of the valid flag after the end of a message transmission, and the host verifies

Listing 6.2: Synchronization for Output State Ports

```
Initialization:
  Transmit := 0
  Valid := 0

Host:
  WHILE Transmit != Valid
    <do nothing>
  ENDWHILE
  IF Transmit == 0 THEN
    <update buffer 1>
    Valid := 1
  ELSE
    <update buffer 0>
    Valid := 0
  ENDIF

TISS:
  IF Valid == 0 THEN
    Transmit := 0
    <transmit buffer 0>
  ELSE
    Transmit := 1
    <transmit buffer 1>
  ENDIF
```

always both fields before it starts updating the content of a buffer (it is only allowed to update the content of a buffer if the valid and the transmit field are pointing to the other buffer).

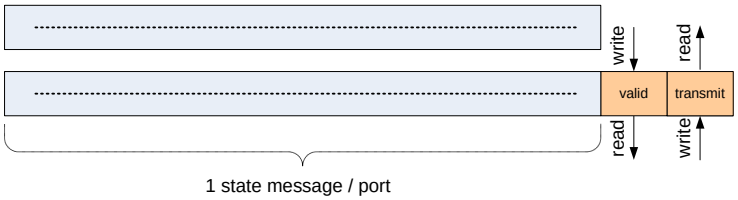

Figure 6.3: Output State Port

As for input state ports, output state ports can be accessed by *implicit synchronization* if the host is perfectly synchronized to the global time. Based on the a priori known points in time when the output state port is accessed by the TISS, the host can temporally interleave its write accesses with the read accesses of the TISS in way that conflicts are avoided. In this case, the double buffer is not required which avoids the associated memory overhead. Therefore, the explicit synchronization mechanism based on double buffering can be individually enabled and disabled for each output state port.

Event Ports

We call a message an *event message* if its value refers to the difference between an *old state* and a *new state* [Kop97]. This difference is also called *event information*. Event-messages have to be handled according to the *exactly-once semantics*. This means that every sent message should be processed exactly once by each correct receiver. Therefore, messages have to be consumed on reading and unread messages have to be queued instead of employing an update-in-place strategy as it is the case for state messages.

Event ports are used for the sporadic transmission of messages with event semantics. In order to support exactly-once semantics, event ports are realized as queues (ring buffers) that can hold multiple event messages at a time (see Figure 6.4). The length of a queue is variable.

Since the sender is blocked when the sender's queue is full, exactly-once semantics is supported under the condition that the receiver services its input queue fast enough to prevent overflows. Without queues, the receiver would have to guarantee a *maximal service time*. By using queues, weaker guarantees are possible.

One example for such a weaker guarantee is that the receiver guarantees a *mean service time* $t_{meanservice}$ within each time interval of length l. In this case, an overflow at the receiver can be prevented by configuring the associated encapsulated communication channel to a period equal to $t_{meanservice}$ and by configuring the queue length to $l/t_{meanservice}$.

Note that this condition is sufficient but not necessary to prevent overflows at the receiver and there exist other, more elaborate, queuing strategies that rely on weaker assumptions [BT04].

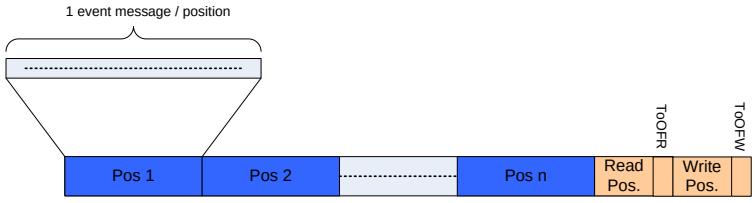

Figure 6.4: Event Port

Input Event Ports. Event ports are synchronized by means of variables for the write position and the read position of a given port (explicit synchronization). The TISS indicates the presence of a new *input 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 (see pseudocode listing 6.3). 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.

By looking only at the read and write position, it is not possible to differentiate between a full and an empty queue. Therefore, an event port incorporates two additional bits, the *Toggle on Overflow Read Position* (ToOFR) bit and the *Toggle on Overflow Write Position* (ToOFW) bit. At startup, both bits are initialized to zero. The ToOFR bit (resp. ToOFW bit) is always toggled when the increment of the read position (resp. the write position) has caused an overflow of the read position (resp. the write position). If the read and write position are equal and the ToOFR and the ToOFW bits are equal, the queue is empty. If the read and write position are equal and the ToOFR and the ToOFW bits are not equal, the queue is full.

Listing 6.3: Synchronization for Input Event Ports

```
Initialization:
  ReadPos := 0
  WritePos := 0
  ToOFR := 0
  ToOFW := 0

Host:
  IF (ReadPos == WritePos) AND (ToOFR == ToOFW) THEN
    RETURN <queue empty>
  ELSE
    <read message from position ReadPos>
    ReadPos := (ReadPos + 1) modulo QueueSize
    IF ReadPos == 0 THEN
      <toggle ToOFR>
    ENDIF
  ENDIF

TISS:
  IF (ReadPos == WritePos) AND (ToOFR != ToOFW) THEN
    <signal overflow>
  ELSE
    <write message to position WritePos>
    WritePos := (WritePos + 1) modulo QueueSize
    IF WritePos == 0 THEN
      <toggle ToOFW>
    ENDIF
  ENDIF
```

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

6.1.3 Port Interrupts

The TISS provides a dedicated *port interrupt* for each port to facilitate the temporal alignment of the encapsulated communication channels on the time-triggered NoC and the computational activities of the host. In the case of input ports, these interrupts are called *reception interrupts* and are triggered after a message was received at that port. In the case of output ports these interrupts are called *transmission interrupts* and are triggered after the associated message has been transmitted.

The host can enable or disable the port interrupts for each individual port.

6.1.4 Time Stamping Service

The TISS integrates a *time stamping service* that supports time stamping of received messages with respect to the global time. Time-stamps in event messages allows to temporally relate messages that have been sent by *different* micro components to each other. For state messages, time stamps can be used to determine whether the *real-time image* [Kop97] contained in a state port is still temporally accurate or not.

Since time stamps require additional memory space, the time stamping service can be individually enabled for each single port.

6.1.5 Message Ordering

The TTSoC architecture provides two types of guarantees with respect to message ordering. Within a single encapsulated communication channel, the architecture guarantees *total temporal ordering* of received messages with respect to their sent instances (i.e., messages are received by any receiver in the same order as they have been sent). Across multiple encapsulated communication channels, a *consistent delivery order* is guaranteed.

Ordering Within a Single Channel

Within a single encapsulated communication channel, the TTSoC architecture guarantees that messages are received by any receiver in the same order as they were sent. This property is essential for being able to interpret serial data.

For a single encapsulated communication channel, message ordering is offered natively by the design of the time-triggered NoC and the TISSs. The sending TISS injects the messages of a given encapsulated communication channel into

the time-triggered NoC in the same order as they have been produced by the host. The employed routing strategy within the time-triggered NoC guarantees that the messages in an encapsulated communication channel are kept in the same order as they have been injected. As mentioned in Section 4.2.2, the pulses of a pulsed data stream consist of a fixed number of fragments. If we consider a pulsed data stream with pulses that are partitioned into n fragments, the i^{th} fragment will be transmitted over the same path in each pulse of the pulsed data stream for all $i \in \{1, ..., n\}$. Thus, a pulse cannot "overtake" another pulse of the same pulsed data stream.

Nevertheless, the individual fragments, 1 to n, within a *single* pulse can be sent over different paths of different length and can thus be received inconsistently. An inconsistent reception order of fragments within a pulse is not visible for the host, since the host reads only completely received messages, which is assured either by using the explicit synchronization mechanisms of the ports or by implicit synchronization.

System-wide Ordering

In contrast to many other state-of-the-art NoCs, which provide ordering guarantees only within a single channel (e.g., Æthereal), the TTSoC architecture enables the establishment of a *consistent delivery order* across multiple encapsulated communication channels. Consistent delivery order means that any two micro components will see the same sequence of message receptions within a defined set of encapsulated communication channels. This type of ordering facilitates to establish *replica deterministic* behavior of micro components which is required for the transparent masking of hardware errors by TMR [Pol94].

Since the TTSoC architecture supports different topologies with respect to the time-triggered NoC, a message on a short path can be received before a message that has been sent earlier, over another encapsulated communication channel via a longer path. Thus, messages can potentially arrive in an inconsistent order at the TISSs of different micro components as depicted in Figure 4.10.

A consistent delivery order can be established by exploiting the global time base in conjunction with the time-triggered communication schedule and the reception interrupts (see Section 6.1.3) of the TISS. If consistent delivery order is required for a set of encapsulated communication channels, the reception interrupt for each message is delayed at each TISS until the message has been completely received by the last micro component. This instant is a priori known due to the predefined time-triggered message schedule. Thus, the TNA can program the delayed reception interrupts together with the MEDLs in the individual TISSs accordingly.

6.1.6 Security-Relevant Properties

For some applications, it might be required to prevent a sensitive part of an SoC from communicating with other parts of the system. For this purpose, the TTSoC architecture provides adequate security mechanisms by design.

Each input and output port is mapped to a dedicated memory region within the micro component. This mapping can be exclusively configured by the host. Thus, the host can decide which memory region should be accessible for a given port and the associated encapsulated communication channel. Memory regions that are associated with an input port can only be written by the source of the corresponding encapsulated communication channel, while memory regions that are associated with an output port can only be read by every destination of the corresponding encapsulated communication channel.

The TNA—as a trusted component—ensures that the encapsulated communication channels are routed only in such a way that security violations do not occur. The routing information of the encapsulated communication channels is stored within the TISSs of the micro components (see Section 7.1.1) and can be exclusively (re-)configured by the TNA. Thus, no other, potentially malicious, component in the SoC can interfere with the routes that have been set up by the TNA.

6.2 Additional Services

In addition to the communication service, the TISS provides the *global time service*, the *programmable timer interrupt service*, the *watchdog service*, the *power control service*, and the *diagnostic dissemination service*.

6.2.1 Global Time Service

The TISS provides access to the synchronized global time (see Section 4.2.2) via the *global time service*. The global time service allows to establish a temporal relationship between events that have been time-stamped by different micro components within the SoC.

Synchronizing the global time with an external time base (e.g., GPS) via a gateway facilitates the temporal coordination of activities spanning multiple SoCs (e.g., coordination of actuators attached to different SoCs).

If the consistent global time has not yet been established, because the whole chip has just been started-up or restarted, the global time in the TISS has the initial value 0.

6.2.2 Programmable Timer Interrupt Service

The TISS provides a *programmable timer interrupt service* that supports two kinds of interrupts. One the one hand, it supports periodic interrupts with definable periods and phase offsets and, on the other hand, interrupts that occur at a single definable point in time. The timer can be programmed by the host via two registers, the *interrupt_pattern register* and the *interrupt_mask register*, which both have the same width as the global time. An interrupt will be signaled whenever the following condition is true:

$$interrupt\_pattern == (global\_time \ \& \ interrupt\_mask)$$

6.2.3 Watchdog Service

The TISS provides a *watchdog service* to monitor the health state of the micro component's host. Therefore, the watchdog requires the host to update a dedicated memory location at the TISS with a definable maximum period. If the host fails to update this memory location within the defined period, it will be reseted by the TISS and a failure indication will be disseminated via the diagnostic dissemination service (see Section 6.2.5).

The watchdog comprises two registers, the *watchdog period register* and the *watchdog life sign register*. The watchdog life sign register is the register that has to be updated by the host with a predefined constant value.

The watchdog period register holds the maximum period with which the watchdog has to be updated. The period durations are restricted to negative powers of two of the second, i.e., a period can be 1 second, 1/2 second, 1/4 second, 1/8 second, and so forth. The duration of each period can, thus, be characterized by the corresponding bit of the binary time format; we call this bit the period bit. The watchdog period register encodes the position of the period bit within the binary time format. A reserved value in the watchdog period register indicates that the watchdog function is turned off. The watchdog period register can be exclusively set via the TISS CP interface by TNA.

6.2.4 Power Control Service

Since hosts are generally not considered to be free from design faults, it cannot be assumed that a host will always behave according to the desired configuration that has been set in the Host CP interface. Therefore, it has to be prevented that a misbehaving host can interfere with the correct operation of the other micro components.

In addition the NoC, power is also a shared resource between all the micro components on the SoC. It can be limited by the maximum allowed heat dissipation or by constraints implied by the battery (e.g., lifetime, capacity). A faulty micro component that consumes more power than allowed can influence the correct operation of the entire chip. Therefore, the TISS has physical control over the power lines or the clock lines of the host in order to be able to turn off the host in case it does not operate in conformance to its specification. The host can be powered down via the TISS CP interface which can be exclusively accessed by the TNA (see Host Mode in Section A.2.4, page 120).

6.2.5 Diagnostic Dissemination Service

The TISS records anomalies in the behavior of a host. Any error that is detected by the TISS is stored in the *Error Status Flags*, which is a read-only field for the host. The content of this field is periodically sent in a *status message* to the *Diagnostic Unit* (DU). Immediately after the content has been disseminated, the Error Status Flags are reset. The period of the status message, the the *status dissemination period*, is configured by the TNA.

The Error Status Flags contain information about the health state of the host, determined by the watchdog service, and about errors that are related to the encapsulated communication service (e.g., overflow for event ports or port configuration errors). For a detailed description of the Error Status Flags see Section A.2.4.

6.3 Dynamic Resource Management

The configuration of the TISSs can be dynamically adapted at runtime in order to lay the foundation for integrated resource management. These reconfiguration activities are partly performed by the TNA and partly by the hosts of the micro components.

6.3.1 Reconfiguration Time vs. Reconfiguration Period

With respect to reconfiguration we have identified two relevant performance indicators: The *maximal reconfiguration time* and the *minimal reconfiguration period*.

Maximal Reconfiguration Time. The maximal reconfiguration time bounds the time it takes the TISS to switch between any two configurations (i.e., the maximal duration of the reconfiguration phase). During

the reconfiguration phase, the TISS cannot receive or send any messages. Thus, the maximal reconfiguration time bounds the minimal period of any *continuous service*, i.e., a service that is not interrupted by the reconfiguration activities of the associated micro component.

The maximal reconfiguration time depends exclusively on the TISS, while the activities performed by the RMA and by the TNA to prepare and to check a new configuration are not included in the maximal reconfiguration time.

Minimal Reconfiguration Period. The minimal reconfiguration period bounds the minimal time interval between any two reconfiguration events. Thus, it determines the reactiveness of the entire SoC to reconfiguration requests.

In contrast to the maximal reconfiguration time, the minimal reconfiguration period depends on each activity that is required for reconfiguration. These activates include the preparation of a new configuration proposal by the RMA, the checking of the proposal by the TNA, the dissemination of the configuration data by the time-triggered NoC, and switching to the new configuration performed at the TISS.

6.3.2 Configuration Performed by the TNA

The TNA configures the parameters that are relevant for encapsulation and diagnosis. These parameters are:

- the MEDL and the routing information for the time-triggered NoC (see Section 7.1.1),
- the watchdog period,
- and the power mode of the host.

The time-triggered NoC provides an encapsulated and protected channel from the TNA to the TISS CP interface of all TISSs to guarantee that no other micro component than the TNA has the ability to change these parameters. This encapsulated channel can be realized by a separate physical channel or by dedicated time slots on the main network, protected by hardware mechanisms. Furthermore, the design of the TISS ensures that the host has no write access to the parameters configured by the TNA to prevent that a design error within the host can interfere with the temporal control of the shared time-triggered NoC.

The TNA periodically sends to each micro component a dedicated state message that contains the configuration data for the micro component's TISS. Sending the entire configuration data periodically simplifies the design and the implementation of the TISS, since it allows to use the same mechanisms for start-up, restart, and reconfiguration of a micro component.

In order to reduce the maximal reconfiguration time, the parameters configured by the TNA are stored in a double buffer, where one buffer holds the active configuration while the other buffer is updated by the TNA. Switching between the two buffers can be performed in a single clock cycle in our prototype implementation. Thus, the reconfiguration time has zero jitter—observed on a discrete timescale with a granularity of a single clock cycle.

The employed reconfiguration strategy ensures the MEDLs within the individual TISSs of the SoC are always kept consistent. For this purpose, the activation of the new configuration is performed at all TISSs at the same tick of the global time. This instant is set by the TNA and is called the *global reconfiguration instant*.

6.3.3 Configuration Performed the Host

The host is free to select the access paradigm according to which it wants to interact with the encapsulated communication channels that are attached to the micro component. Therefore, the host can configure each local port to act as a state port (with explicit or implicit synchronization) or as an event port, according to its specific needs. In the case of an event port, the host can also define the length of the message queue.

Furthermore, the host assigns a start address to each individual port, and enables or disables the port interrupt and the time stamping service for each individual port (the time stamping service can only be enabled for input ports).

Since these parameters are only locally relevant and not visible on the time-triggered NoC or at any other micro component, a faulty host cannot interfere with the encapsulation mechanisms of the SoC by manipulating these parameters.

Chapter 7

Prototype Implementation

This chapter describes the design and implementation of a prototype that demonstrates the feasibility of the proposed architecture. It covers the design of the architectural elements, the description of the prototype hardware, and the implementation results.

7.1 Design of the Architectural Elements

This section presents the principal operation of the NoC and the TISSs. Furthermore, the design of a generic frontend is described that provides a temporal firewall interface to the application computer and includes an interrupt controller to manage the interrupts generated by the TISS and by optional middleware modules. The design of the RMA and TNA are not part of this thesis.

7.1.1 NoC

The NoC comprises multiple *fragment switches*, which transport the fragments of a pulsed data stream from the sender to one or more receivers via one or more *hops*. The fragment switches operate exclusively on fragment level and are not aware that messages are split up into fragments in the sender's TISS and are reassembled in the TISSs of the receivers.

For the purpose of fragment transmission, fragment switches are interconnected via bidirectional links. The end points of such a link are called *interconnects*. The number of interconnects of a fragment switch is denoted as the *arity* of the switch. Since the interconnects are bidirectional, a fragment switch with arity n has n inputs and n outputs. The arity determines the possible topologies

that can be constructed with the fragment switches. For example, consider a two-dimensional *mesh*, a two-dimensional *torus*, and an arbitrary topology (see Figures 7.1, 7.2, and 7.3). The mesh requires switches with an arity of three, four, and five, while the torus requires switches with an homogeneous arity of five. The depicted example of an arbitrary topology requires switches with an arity of two, three, and four.

During the design of the NoC, the arity of a fragment switch can be chosen to support a wide variety of topologies. In the current prototype implementation, we use fragment switches with an arity of five and arrange the switches in a *torus* topology. Compared to a mesh, the torus provides a better performance due to its uniform connectivity which reduces the average logical distance between the individual switches [Wik05].

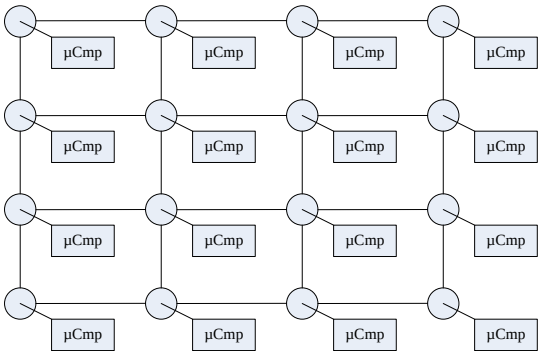

Figure 7.1: 2D Mesh Topology

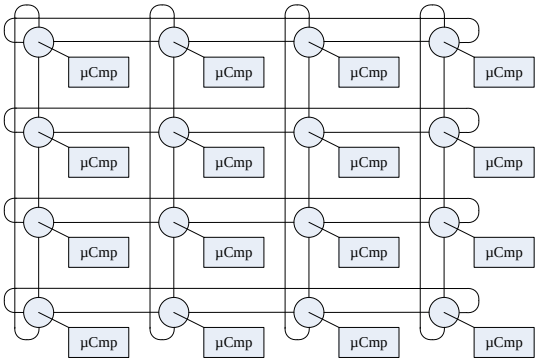

Figure 7.2: 2D Torus Topology

The NoC is based on *wormhole switching* and *source routing*. Wormhole switching means that a fragment of a pulsed data stream is sent as a sequence of fixed sized flow-control digits or *flits* through the network. The size of a flit is determined by the width of the NoC, 32 bit in our current implementation. The first flit of a fragment, called the *header flit*, contains the routing information,

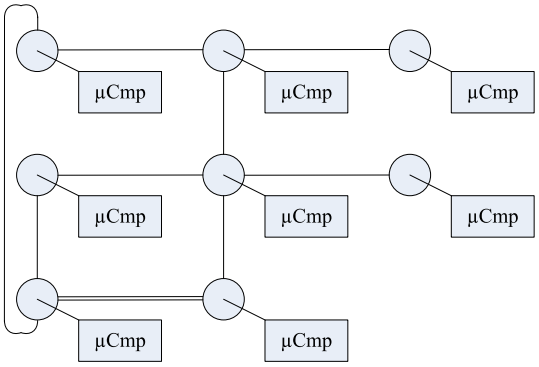

Figure 7.3: Example of an Arbitrary Topology

while the trailing flits of the fragment contain the data. In order to increase the bandwidth and to keep the latency low, the flits of a fragment traverse the path from the sender to the receiver in a pipelined manner, where the trailing flits are directly following the header flit without waiting until the header flit has reached its destination. Thus, a fragment "digs" through the network like a worm with the head (i.e., the header flit) controlling the direction (i.e., the route); hence the word *"wormhole switching"*.

Source routing means that the source (i.e., the TISS of the sender) supplies the entire routing information. In our case, the routing information for a particular fragment is contained in the fragment's header flit. A header flit consists of multiple *entries*, one for each switch in the route from the sender to the receivers.

The entry for a specific switch defines the interconnects to which the flits of the corresponding fragment have to be forwarded. For a switch with arity n an entry consists of n bits, where each bit corresponds to one of the switch's interconnects. A value of 1 in one of these bits indicates that, all flits that will be received via the same interconnect through which the header flit was received, should be forwarded to the interconnect that corresponds to that bit. Consider a fragment switch with five interconnects labeled a, b, c, and d (i.e., a fragment switch with an arity of 4). Assume that a header flit is received via interconnect a and the bit of interconnect c is set to 1 in the corresponding entry. This would mean that the switch should forward each flit that is received on interconnect a to interconnect c.

Whenever a fragment switch receives a header flit, it consumes the first entry and adjusts its internal settings (i.e., the mapping of inputs to outputs) accordingly. A flit switch with arity n requires $n(\log_2 n)$ bits to store the mapping of inputs to outputs (for each of the n outputs, $\log_2 n$ bits are required to specify one of the n possible inputs to which the given output should be connected). After having consumed the first entry, the fragment switch propa-

gates the remainder of the header flit (and all subsequent trailing flits) to the next fragment switch according to the internal settings. The internal settings remain persistent until the arrival of a new routing flit at any of the switch's interconnects.

The NoC transports a fragment always within a single uninterruptible *burst*, with one flit per clock cycle, with each flit following the same route. To allow pulse interleaving (i.e., multiple concurrent pulses sharing the same interconnect) a pulse can be divided into multiple fragments that can be individually scheduled and routed. The fragment switches operate only at fragment-level and are unaware that fragments are assembled to messages within the TISSs.

Furthermore, fragment switches are not aware of the time-triggered message schedule and contain no arbitration logic and message buffers. It is guaranteed by the time-triggered message schedule in the TISSs that the injection of messages into the NoC is done in a way such that no conflicts occur at any of the interconnects of the fragment switches.

7.1.2 TISS

As depicted in Figure 7.4, the TISS consists of two modules, the *pulse manager* and the *port manager*. The pulse manager sends and receives single fragments to and from the NoC, while the port manager assembles multiple fragments to messages and stores them in the memory of the frontend in the host. With respect to the *ISO-OSI reference model* [Ros90] the pulse manager implements the *data link layer* and the *network layer* (layers two and three), while the port manager implements the *transport layer* (layer four).

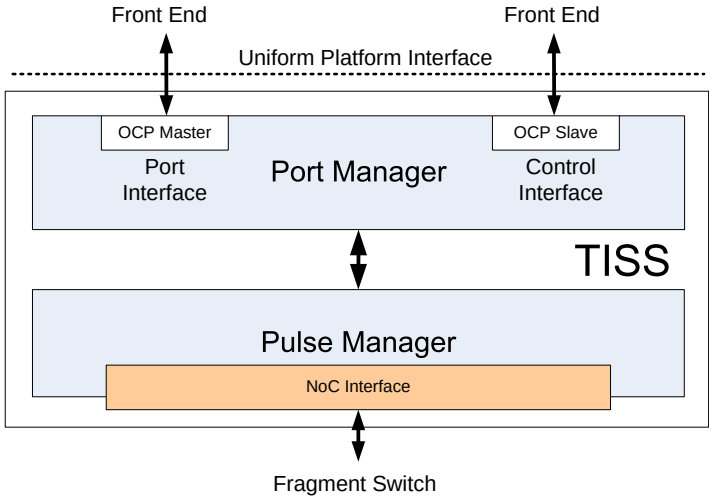

Figure 7.4: Structure of the TISS

Pulse Manager

The pulse manager is attached to exactly one fragment switch of the NoC via the NoC interface. Its purpose is to inject and to receive single fragments to and from its attached fragment switch, according to the time-triggered message schedule. For this purpose, it contains the locally relevant part of the time-triggered message schedule (i.e., all the fragments of a message that have to be send or received by the micro component) in the MEDL. The MEDL is organized in multiple linked lists of MEDL entires (one entry for each fragment), with each list being dedicated to a specific period, as shown in Figure 7.5. Within each list, the entries are organized in ascending order of their activation instants (e.g., activation of a send or receive operation) starting from the start-up or the reconfiguration instant. Conceptually, the TTSoC architecture supports 32 periods ranging from two seconds down to 2^{-30} seconds (approximately one nanosecond). In the current prototype implementation, the smallest achievable period is 2^{-27} seconds due to constraints imposed by the prototype implementation technology (FPGA).

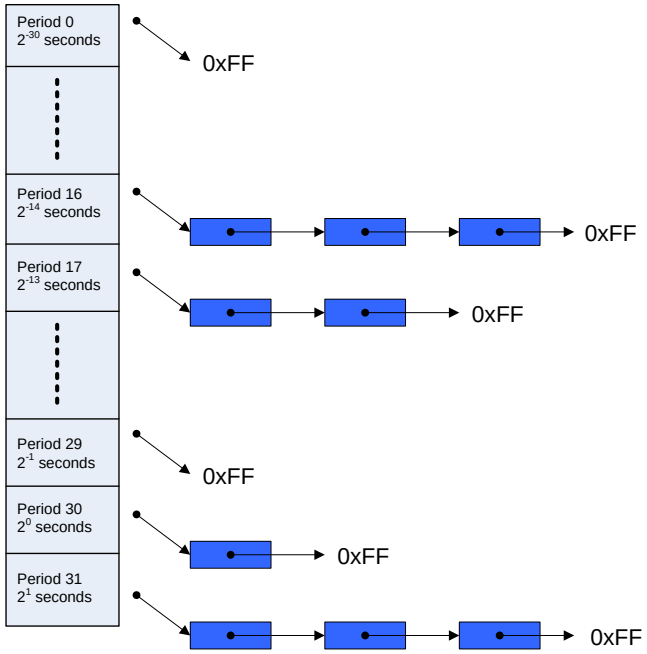

Figure 7.5: Organization of the MEDL

To support multiple periods, the pulse manager provides a dedicated compare logic for each list in the MEDL, i.e., for each period. That logic triggers the corresponding send or receive operation whenever the significant part of the global time (i.e., the bits right to the period bit, see Section 4.2.2) matches the phase of the operation in the next entry of the list.

The pulse manager does not contain any buffers for storing entire fragments. Whenever a fragment is received via the NoC, the flits of the fragment are passed on, one-by-one, to the port manager together with the associated port number, which identifies the corresponding message, and the fragment number, which identifies the fragment within the message. In reverse, when a fragment is scheduled for transmission, the pulse manager issues a request to the port manager which includes the port number and the fragment number of the requested fragment. The port manager then passes the requested fragment flit-by-flit to the pulse manager, which directly passes the flits on to the attached fragment switch via the NoC interface.

In addition to dealing with media access control, the pulse manager is responsible for routing. Therefore, it stores, for each fragment to be sent by the micro component, the corresponding header flit in the *header memory*. Whenever a fragment has to be sent, the pulse manager will first pass the corresponding header flit to the fragment switch and then forward the trailing flits from the port manager to the fragment switch.

The MEDL and the header memory are written by the TNA via the TISS CP interface. In the current prototype implementation, the TNA accesses the CP interface of the TISSs through the NoC via dedicated pulsed data streams. At each TISS, the pulsed data stream associated with the TISS-CP interface is received on port 127. Whenever a fragment is received via that port, the pulse manager does not pass the fragment to the port manager, but it uses it for updating the MEDL and the header memory. The MEDL and/or the header memory are realized as double buffers, with one buffer holding the active configuration while the other buffer is updated by the TNA. The activation of the new configuration (i.e., the switch from one buffer to the other buffer) is performed at all TISSs at the same tick of the global time (at the periodic reconfiguration instant, see Section 6.3) in order to ensure consistency of the configuration in all TISSs of the SoC.

Port Manager

The port manager provides the UNI to the host (see Section A for a detailed specification of the UNI). Its main purpose is to manage the ports in the memory of the host's frontend. Based on the *port configuration memory* (see Section A.2.2), the port manager maps each fragment that is received or sent by the pulse manager to the corresponding address in the memory of the frontend. Message fragments are properly aligned and event messages are placed or fetched at or from the correct position of the associated queue.

Furthermore, the port manager manages the port synchronization flags (e.g., write and read position of event ports and synchronization flags for state

ports) that ensure that data transfers between the TISS and the host are consistent.

In addition to port management, the port manager implements the *programmable timer-interrupt service*, the *watchdog service*, the *power control service*, and the *diagnostic dissemination service* (see Section 6.2).

7.1.3 Frontend

This section describes the design of a generic frontend that can be easily extended to include domain-specific middleware modules that refine and extend the services provided at the UNI. As depicted in Figure 7.6, the frontend consists out of the *port memory*, optional *middleware modules*, the *interrupt controller*, the *application computer address decoder*, and the *port interface address decoder*.

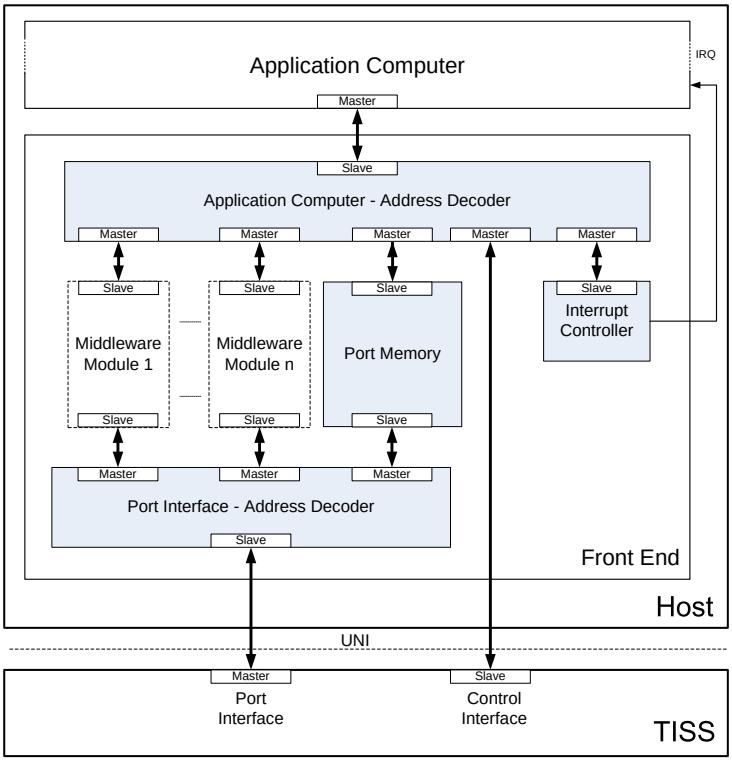

Figure 7.6: Structure of the Frontend

Optional Middleware Modules

The integration of middleware modules into the frontend enables the refinement and extension of the core platform services provided at the UNI. The design

of the frontend enables the integration of multiple middleware modules that provide different domain-specific services and operate in parallel. In order to increase the resource efficiency, only those ports will be mapped to a specific middleware module that actually require the module's service (e.g., the port of an encrypted communication channel can be mapped to an encryption module). All other ports that do not use the middleware module will not experience any additional delay or jitter.

A horizontal structure of middleware modules exploits the parallel nature of multiple hardware blocks and makes the end-to-end latency independent of the total number of middleware modules in the frontend. Vertically structured middleware modules are only required when the functionality of different modules has to be applied sequentially to the same port (e.g., decryption of a voted port).

Port Memory

The port memory stores the data part of all ports of the micro component, except for the ports of communication channels that are fed to the middleware modules. While the data part of all ports is stored in the frontend (i.e., in the port memory or in middleware modules), the synchronization flags of the ports are exclusively stored within the TISS.

The port memory acts as a *temporal firewall interface* [KO02] for the time-triggered NoC by supporting the *information pull* paradigm for input ports and the *information push* paradigm for output ports. Information pull is ideal for sending messages, because the sender can determine the instant for passing messages to the communication system, while information pull is ideal for the reception of messages, since tasks of the receiver will not be interrupted by incoming messages.

The port memory is realized as a dual-ported memory in order to avoid concurrency problems arising from possible simultaneous accesses of the application computer and the port manager of TISS. In addition to handling concurrency, the dual-ported memory supports a different word and address width at its two ports which allows the width of the application computer's data bus to differ from the width of the NoC.

Interrupt Controller

The interrupt controller captures interrupt events generated by the TISS (the `SFlags` and the `SError` signals of the UNI-control interface, see Section A) and by any optional middleware module in the frontend.

Since interrupt events will be set to logic 1 only for a single clock cycle[1] after the corresponding trigger condition was satisfied (e.g., a queue overflow has occurred), any interrupt event has to be captured by the interrupt controller and has to be transformed to an *Interrupt Request* (IRQ), which is pending until it is acknowledged by the host.

The interrupt controller can be accessed by the application computer via the following registers.

Interrupt Status Register. This register captures any interrupt event generated by the TISS or by a middleware module in the frontend. The detailed layout of the interrupt status register is depicted in Figure 7.7. A pending interrupt is indicated by a value of 1 of the corresponding bit. Writing a 1 to one of the bits acknowledges and clears the corresponding interrupt, whether it was pending or not. The `port index` bits have a special meaning described below.

Port Status Registers. The `port status registers` are four 32 bit registers which are always set together with the `port operation complete interrupt`. They indicate which of the ports triggered the interrupt. Each port is represented by dedicated bit in one of the four registers (port 0 by bit 0 of register 0; port 1 by bit 1 of register 0; ...; port 32 by bit 0 of register 1; ...; port 127 by bit 31 of register 3).

The purpose of the `port index` bits in the `interrupt status register` is to free the host from searching through all four `port status registers` to find the ports that have triggered a `port completion interrupt`. Therefore each of the `port index` bits represents one `port status register`. A value of 1 in a `port index` bit indicates that at least one port represented by the bits in the corresponding register has raised a `port operation complete interrupt`.

The bits in the `port status registers` and the `port index` bits can be cleared by writing a 1 to them. In order to ensure that the application computer does not miss any interrupt, it should always acknowledge the `port operation complete interrupt` before it inspects the `port index` bits and the `port status registers`. If the `port operation complete interrupt` is still cleared (i.e., has a value of 0) after the application computer has serviced and cleared all interrupts in the `port status registers`, it can be sure that no new interrupts arrived while it was servicing the old interrupts.

Interrupt Mask Register. This register selects which interrupts should be enabled. The layout of this register is the same as the layout of the

[1] with respect to the OCP clock of the UNI

`interrupt status register` shown in Figure 7.7. An interrupt is enabled by writing a 1 to the corresponding bit in the interrupt mask. An enabled interrupt will be recorded in the `interrupt status register` and will be signaled to the host via an IRQ. Disabled interrupts do not generate IRQs, but are still be recorded in the `interrupt status register`.

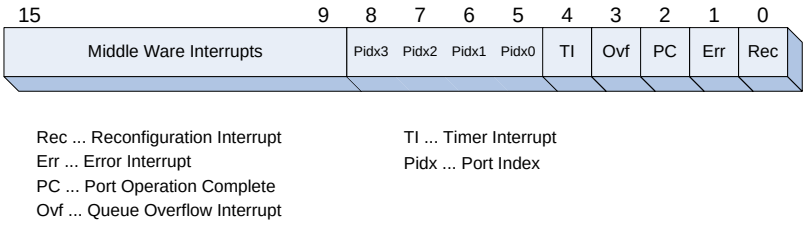

Figure 7.7: Interrupt Status Register

Application Computer Address Decoder

The purpose of the application computer address decoder is to map the port memory, the control interface of the TISS, the registers of the interrupt controller, and the middleware modules into structured address spaces that can be accessed by the application computer. The exact layout of the address space is specific to the frontend and its middleware modules and is not part of the TTSoC architecture definition.

Port Interface Address Decoder

The TISS assumes a homogeneous address space at the *UNI-port interface* and is agnostic to the fact that the addresses of some ports identify a memory location in a middleware module. The purpose of the port interface decoder is to map the port memory and the memories of all middleware modules into a homogeneous address space for the TISS.

7.2 Prototyping Hardware

The prototype implementation is based on a custom-made development kit designed and manufactured by TTTech[2]. The heart of the development kit is motherboard with an FPGA device and with nine extension slots—conforming

[2]http://www.tttech.com

to TTTech's Powerlink specification—for attaching different extension boards. The kit emulates an SoC by housing the NoC, the TNA, and all the TISSs in the FPGA on the motherboard, while the hosts are realized by the extension boards. The available extension boards include several CPU boards equipped with different CPUs and volatile and non-volatile memory, FPGA boards providing various memory resources, multimedia boards, and I/O boards to interface the environment.

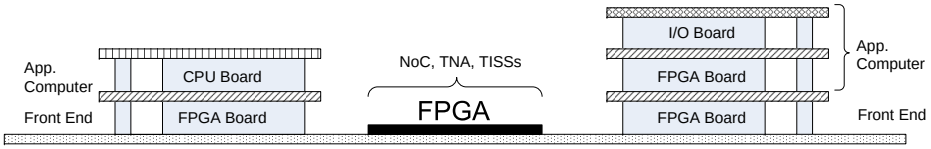

Figure 7.8: Prototype Hardware

The CPU and the FPGA extension boards provide slot connectors on the top and on the bottom which enables stacking multiple extension boards on a single slot of the motherboard (see Figure 7.8). The bottom board is always an FPGA board housing the frontend of the host, which contains optional middleware modules. On top of the frontend there is either a CPU board or an FPGA board taking the role of the application computer. If the micro component has to communicate with the environment, the application computer will also contain an I/O board or a multimedia board on the top of the stack.

7.2.1 The Motherboard

The central element of the motherboard is an Altera EP2C70 FPGA, which is the biggest device of the Altera Cyclone II device family. The features of the FPGA are summarized in the following table:

Logic Elements	68,416
M4K RAM Blocks (4 kbit + 512 Parity Bits)	250
Total RAM Bits	1,152,000
Embedded 18x18 Multipliers	150
PLLs	4
Maximum User I/O Pins	622

The motherboard can be connected to a PC via an Altera USB-Blaster JTAG adapter in order to ease the development process. The Altera development environment enables on-the-fly configuration and flash programming, as well as the communication with an embedded logic analyzer for debugging purposes.

The motherboard contains a 8MByte Flash memory and 2MByte SRAM to enhance the storage capacity of the FPGA. This extra storage may be required

for the implementation of the TNA, which will be realized as a soft core in the central FPGA.

Furthermore, the motherboard contains a slot to connect a Navman Jupiter Pico-T GPS receiver in order to provide a high-precision time base.

7.2.2 FPGA Boards

The FPGA boards will be used to implement the frontend of every host and in some hosts they will also serve as the application computer. The FPGA extension board is based on an Altera Cyclone II EP2C35 FPGA, which has the following features.

Logic Elements	33,216
M4K RAM Blocks (4 kbit + 512 Parity Bits)	105
Total RAM Bits	483,840
Embedded 18x18 Multipliers	35
PLLs	4
Maximum User I/O Pins	475

The board provides various memory devices which are 128Mbit SDRAM, 18 Mbit SSRAM, 64 Mbit Flash memory, and 1 kbit EEPROM to save non-volatile configuration data of the application. Furthermore, it is equipped with an SD-card reader for accessing SD-Flash cards.

7.2.3 CPU Boards

The CPU boards are exclusively used to implement the application computers of some of the hosts (the application computers of the other hosts will be implemented on the FPGA extension boards). All CPU boards comply to TTTech's Powerlink specification. We use different CPU boards to demonstrate the ability to integrate heterogeneous micro components. The following boards are available:

- **Freescale MPC555** 4 MByte Flash, 512kByte SRAM
- **Freescale MC9S12XDP512** 2MByte Flash, 512kByte SRAM
- **Infineon TriCore TC1796** 4MByte external Flash, 4MByte internal Flash, 1MByte SRAM
- **Infineon C167** 1MByte Flash, 512kByte SRAM

7.2.4 I/O Boards

The I/O extension boards interface the environment, by providing serial interfaces for RS232, CAN, LIN, TTP/A, and Ethernet, as well as, digital I/Os.

7.2.5 Multimedia Boards

The multimedia board is equipped with a 240 x 320 pixel LCD color touchscreen and an AC97 compatible audio device. Furthermore, it provides an USB controller that can function as an USB host as well as an USB device.

7.3 Results

In order to evaluate the feasibility of the proposed approach, a VHDL model of the described architectural components was designed and synthesized for the FPGA on the target platform[3].

The time-triggered paradigm enabled a very efficient implementation of the NoC's fragment switches with respect to power, area consumption, and performance. Since the time-triggered schedule is defined in such a way that contention never occurs, the NoC does not have to incorporate any arbitration logic or message buffers. The TISSs inject messages always at the proper instants of the global time, while the fragment switches themselves are agnostic to the time-triggered schedule. Thus, the task of dispatching, which is quite complex for messages with different periods, is contained in the TISSs which reduces the resource overhead of the fragment switches.

Furthermore, the proposed routing strategy has a positive effect on the area and power consumption of the fragment switches. Due to the source routing concept, a fragment switch has to store only the routing information of the fragments that are currently passing through the switch. As described in Section 7.1.1, a fragment switch with arity n requires only $n(\log_2 n)$ bits of storage for routing, independently of the total size of the NoC or the number of different fragments that are passing sequentially through the switch.

The proposed NoC outperforms a traditional bus in two aspects. First of all, a bus scales very poorly as the number of attached cores increases, since a large number of cores results in a high capacitive bus loading due to fan-out [Wik05]. In order to further optimize the power efficiency of the NoC, the individual fragment switches are automatically set into a standby mode when

[3]The VHDL model was designed, implemented, and tested by Christian Paukovits in the course of his PhD thesis.

no fragments are to be processed by the switch. This is accomplished by means of clock gating. The power consumption induced by the transmission of a single flit is only determined by the power consumption of a single fragment switch since a single flit causes exactly one fragment switch to be active at a time. The overall peak power consumption is determined by the maximum number of concurrently transfered flits and is independent from the length of the routes.

FPGA — Altera EP2C70		
Component	Logical Elements	FMax
Switch	444	342,58 Mhz
Port M.	1027	142,98 Mhz
Pulse M. 16	1453	172,53 Mhz
Pulse M. 32	2918	139,98 Mhz

Table 7.1: Implementation Results

Table 7.1 summarizes the cost and performance results of the synthesized VHDL-model. An interesting observation is that the port manager of the TISS which includes all the synchronization mechanisms for state and event ports consumes only a relatively small portion of the total number of logical elements required by the TISS. The main part of the logical elements of the TISS is consumed by the pulse manager due to its support for dispatching messages with multiple periods (each period requires a dedicated comparator logic). The table shows that doubling the number of supported periods results in double the amount of logical elements for the pulse manager.

Table 7.2 depicts the memory requirements of the TISS. The total size of the MEDL depends on the number of fragments that should be send or received by the micro component (each periodic send or receive operation consumes an entry in the MEDL). Each fragment has a to be preceded by a header which specifies the route of the fragment, but multiple fragments can share the same header (in case they use the same route). Thus the number of required headers is less-than-or-equal the number of fragments that have to be sent by the TISS. The size of the memories for the port flags and the port configuration depends on the number of ports that should be supported by the micro component, since each port requires an entry in both of the memories.

For an exemplary configuration we consider a micro component that supports 32 different ports. We assume that the messages of 24 of the ports can be sent within a single fragment, while the messages of the 8 remaining ports have to be fragmented into two fragments due to concurrent transfers over the same link of any of the fragment switches. This results in a total number of 40 fragments that have to be processed by the TISS. We assume that 8 of the fragments share the same route, which means that 32 headers are required for

Entity	Width
MEDL Entry	71 bit
Fragment Header	32 bit
Port Configuration Entry	32 bit
Port Flags Entry	32 bit

Table 7.2: TISS - Memory Requirements

the fragments. This exemplary configuration would require 32 port flags entries, 32 port configuration entries, 40 MEDL entries and 32 fragment headers, which results in a total memory requirement of 739 bytes.

Summarizing the implementation results, we can conclude that the time-triggered paradigm enabled a very efficient implementation with respect to power, area, and performance. Furthermore, the design is flexible enough to be customized according to domain-specific requirements.

Chapter 8

Conclusion

If we want to manage the complexity of an evolving system at a higher level of abstraction, we have to conceptualize components that constitute stable intermediate forms and exhibit aggregate properties. If these properties can be described by an appropriate interface model, it is not required to understand the internal structure of the components in order to reason about the interactions among components and the emerging system properties. Furthermore, an appropriate interface model contributes to the *"evolvability"* of a system by allowing to change and to enhance a component's implementation, in order to response to technological developments without revising the overall system architecture.

Inspired by this insight, we have introduced in the proposed time-triggered SoC architecture the notion of a *micro component*, which can be considered as a basic unit of abstraction that provides its functionality via a message-based interface, which is defined both in the value domain and in the temporal domain. The micro components within an SoC are interconnected through a predictable and deterministic time-triggered on-chip network with inherent fault isolation and a global time base.

In contrast to existing synchronous on-chip interconnects (e.g., Æthereal, Sonics SiliconBackplane), which employ a TDMA scheme solely to establish resource guarantees with respect to *bandwidth* and *latency*, the proposed SoC architecture is based on a new communication primitive, called *pulsed data stream* [Kop06], which uses a TDMA scheme to schedule periodic send instances of entire application-level messages. The supported periods are in the range from a few nanoseconds, to milliseconds, up to seconds. This enables the perfect alignment of the activities within the network to the activation of periodic application task (e.g., periodic dissemination of a sensor value in a process control application). In addition, a global time base is provided at the

application level in order to facilitate the temporal coordination of subsystems distributed across multiple micro components.

The proposed SoC architecture prevents by design any unintended and unwanted interference between micro components. This reduces the complexity of the resulting system and enables the integration *mixed-criticality* subsystems. Interactions between micro components are restricted to occur exclusively via explicitly exchanged messages on the time-triggered NoC.

As a key element for encapsulation we have introduced the TISS, which provides a set of core platform services via the UNI. Among these services are the so-called *encapsulated communication channels*, which are unidirectional channels that transport messages at predefined points in time from a single source to one or more destinations. The 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. Temporal and spatial partitioning with respect to encapsulated communication channels is guaranteed by the TISS, which acts as a guardian for the shared time-triggered NoC. This ensures that a design fault (e.g., a software fault) within a given micro component cannot lead to a violation of the micro component's temporal interface specification in such a way that the communication between other micro components would be disrupted.

The inherent fault isolation and the determinism of the proposed architecture can be ideally exploited to perform error masking by TMR. We have shown (*i*) how the architecture's determinism ensures that correct replicas always reach the same computational result within a bounded time interval, which is required for exact voting, and (*ii*) how the encapsulation mechanisms preserve the independence of replicas by preventing common mode failures.

Complementing the architectural framework, we have introduced a naming scheme that is ideally suited for large embedded systems based on multiprocessor SoCs. One key aspect is the hierarchical system structure, which provides a dedicated, independent, and domain-specific namespace for each application subsystem within a distributed system. Thus, subsystems can be independently developed by different suppliers without the need for a central *naming authority* that coordinates the naming process for the entire system. A further benefit of the proposed solution is the *location transparency* for logical system entities, established by decoupling the logical and the physical system structure, which facilitates *model driven design* and *dynamic resource management*.

By building a prototype implementation we have shown that the time-triggered paradigm enables a very efficient implementation of the proposed architecture with respect to power, area, and performance. A major advantage of our

approach is that the NoC does not have to incorporate any arbitration logic or message buffers since the time-triggered message schedule is constructed in such a way that contention never occurs.

Appendix A

Specification of the UNI

The chapter contains the detailed specification of the *Uniform Network Interface* (UNI). The UNI consists of two parts: the *port interface*, through which the TISS accesses the ports in the memory of the host, and the *control interface*, which is used by the host for configuration and synchronization purposes. Both interfaces were designed to be fully compliant with the Open Core Protocol in order to insure a wide acceptance by the industry.

A.1 The Port Interface

The purpose of the port interface is to access the ports in the memory of the host. Transactions over this interface are exclusively triggered by the TISS according to the time-triggered message schedule. Therefore, the port interface is realized as an OCP Master on the side of the TISS and as an OCP Slave on the side of the host.

A.1.1 Signal Specification

This section describes the OCP signal specification of the UNI port interface. The individual signals are listed in Table A.1.

Clk. At the UNI, the TISS drives the OCP clock while the frontend in the host is responsible for controlling the clock-domain crossing between the clock domains of the TISS and the host. Freeing the TISS from dealing with clock-domain crossing simplifies its design significantly.

MAddr. This field specifies the address of the current read or write transfer. To be fully OCP-compliant, `MAddr` is a byte address. Since OCP requires

Name	Width	Driver	Function
Clk	1 bit	TISS	OCP clock
MAddr	16 bit	TISS	transfer address
MCmd	3 bit	TISS	transfer command
MData	32 bit	TISS	write data
SData	32 bit	host	read data
SError	1 bit	host	error status

Table A.1: OCP Signals of the Port Interface

all addresses to be word aligned, and since the port interface has a data word width of 32 bit, the two least significant bits of the address are hardwired to 0.

MCmd. This field indicates the type of transfer the TISS is requesting; the following three OCP command are supported.

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

MData. `MData` carries the write data from the TISS to the host; the width of this field is equal to the width of the NoC (32 bit in our prototype implementation).

A.1.2 State and Event Ports

This section describes the memory layout of state and event ports as they are stored in the frontend of the host. The TISS accesses the port memory of the frontend by data words that have the same width as the width of the NoC, which allows the TISS to store or to fetch an entire flit in each cycle to or from the port memory. In our current implementation of the NoC, this width is 32 bit.

State Ports. The memory layout of a state port depends on the message size (in number of flits), the synchronization method (explicit or implicit), and the optional activation of the time stamp service for that port. Figure A.1 shows the layout of an output state port of size N (flits) with explicit synchronization. For an output state port with explicit synchronization, $2N$ data words are allocated in the memory within the frontend due to the shadow

memory required for explicit synchronization. An output state port with implicit synchronization consumes only N data words. For input state ports the memory consumption depends on the activation of the time stamping service. Figure A.2 shows the memory layout of an input state port with enabled time stamps. An input state port with enabled time stamps consumes $N + 2$ data words due to the 64 additional bits required for storing the global time, while an input state port with disabled time stamps consumes only N data words.

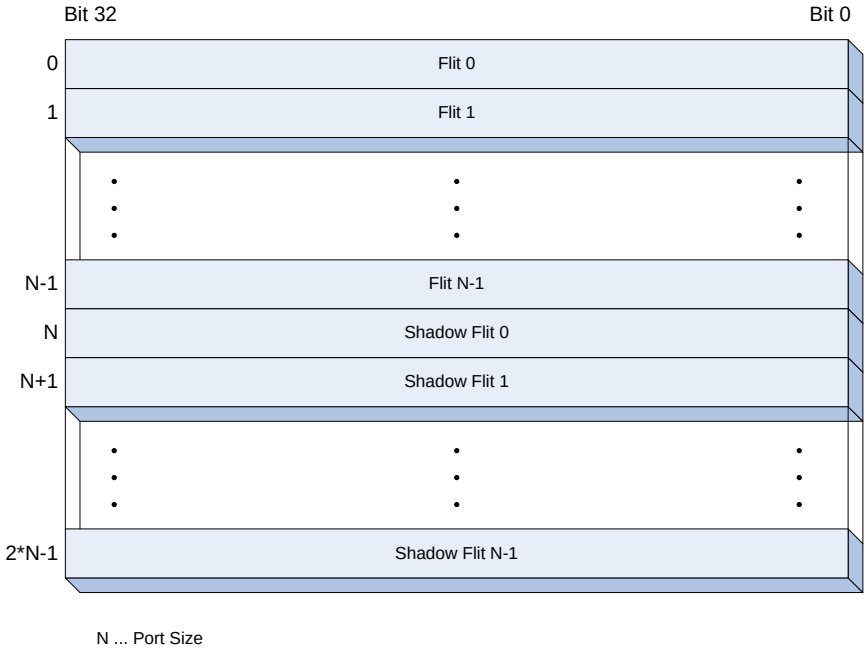

Figure A.1: Output State Port with Explicit Synchronization

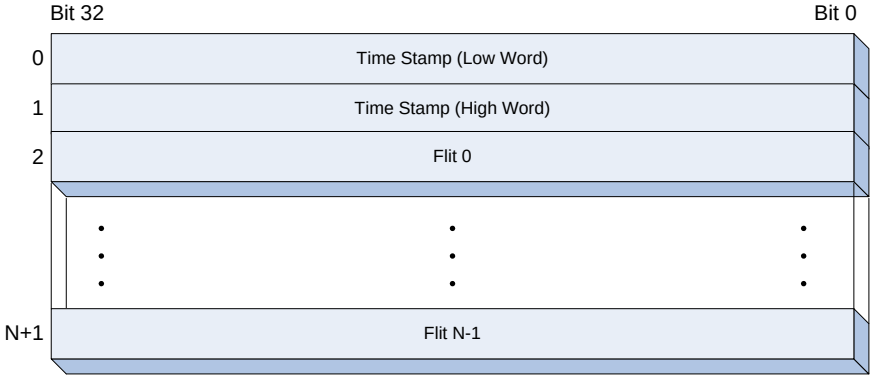

Figure A.2: Input State Port with Enabled Time Stamps

Event Ports. The memory layout of event ports depends on the message size (in number of flits), queue length (in number of messages), and the optional activation of the time stamp service. Figure A.3 depicts an event port with message size N, queue length L, and time stamping enabled. With enabled time stamps an event port consumes $L(N + 2)$ data words in the frontend memory, since a time stamp is stored for each message in the queue. Event ports with time stamping disabled consume only LN data words.

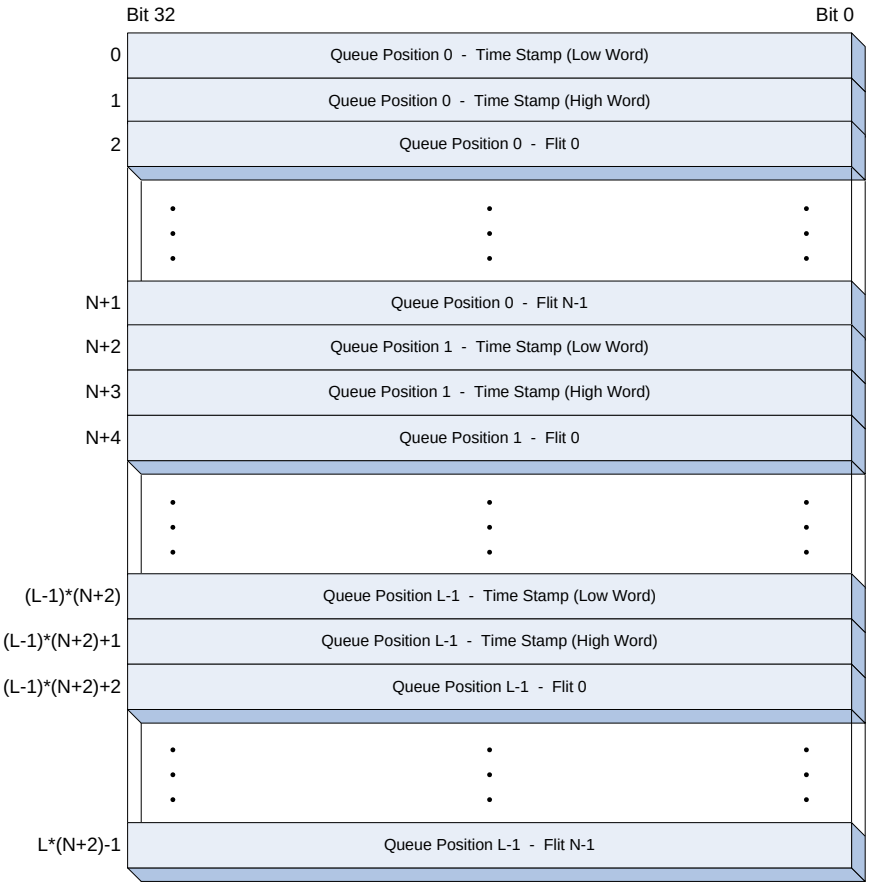

Figure A.3: Event Port with Enabled Time Stamps

A.2 The Control Interface

The *control interface* is used for configuration and synchronization purposes. It is implemented as an OCP slave and has—independently of the width of the NoC—a fixed data word width of 32 bit.

A.2.1 Signal Specification

This section describes the OCP signals of the control interface. Besides the OCP data flow signals, which are used to access TISS-internal registers and memories for configuration and synchronization purposes, the interface provides sideband signals to reset the host and to indicate error conditions and interrupts (see Table A.2).

Name	Width	Driver	Function
Clk	1 bit	TISS	OCP clock
MAddr	9 bit	host	transfer address
MAddrSpace	2 bit	host	address space
MCmd	3 bit	host	transfer command
MData	32 bit	host	write data
MByteEn	4 bit	host	write byte enable
MRespAccept	1 bit	host	host accepts response
SCmdAccept	1 bit	TISS	TISS accepts transfer
SData	32 bit	TISS	read data
SResp	1 bit	TISS	transfer response
SError	1 bit	TISS	error status
SFlag	10 bit	TISS	IRQ status
SReset_n	1 bit	TISS	host reset

Table A.2: OCP Signals of the Control Interface

Clk. As for the port interface, the TISS drives the OCP clock lines while the frontend in the host is responsible for handling the clock-domain crossing between the clock domains of the TISS and the host.

MAddr. This field specifies the address of the current read or write transfer. `MAddr` is a byte address, with the two least significant bits are hardwired to 0.

MAddrSpace. This field specifies the address space and is an extension of `MAddr`. The control interface is partitioned into three different address spaces:

MAddrSpace[1:0]	Address Space
00	reserved
01	port configuration
10	port synchronization flags
11	register file

The function of the address spaces will be described below.

MCmd. This field indicates the type of transfer the host is requesting. The following three OCP commands are supported: idle, write, and read. The encoding is the same as in the `MCmd` field of the UNI port interface (see Section A.1).

MData. This field carries the write data from the host to the TISS. The width of this field is independent of the width of the NoC.

MByteEn. This field enables a partial access of a data word, by indicating which individual bytes of the word are part of the current transfer. Therefore, `MByteEn` consists of one bit for each byte in the data word. Setting `MByteEn[i]` to value 0 indicates that the byte associated with `MData[(8i+7):8i]` or `SData[(8i+7):8i]` is masked out in the current transfer.

MRespAccept. The host indicates that it accepts the current read data from the TISS by setting this signal to 1. As long as the signal is set to zero, the TISS will hold the value of the last response steady. This signal may be required for slower hosts that cannot read the response data in a single cycle.

SCmdAccept. A value of 1 on this line indicates that the TISS accepts the transfer request of the host.

SData. This field carries the requested read data from the TISS to the host. The width of this field is independent of the width of the NoC.

SResp. The TISS uses this field to respond to a request from the host. The `SRep` is encoded as follows:

SResp[1:0]	Response
00	no response
01	data valid
10	request failed
11	response error

If `SResp` has the value 01, the `SData` field holds valid read data. The value 10 indicates that a read or write transfer was requested at an invalid address. The value 11 indicates that a write transfer was requested for a read-only address (e.g., the global time register).

SError. Whenever the TISS detects one or more errors (see the description of the register file) it sets the `SError` signal to 1 for exactly one cycle of the OCP clock.

SFlag. Via this field, interrupt events generated by the TISS are communicated to the host (to be precise, to the interrupt controller in the frontend). If an interrupt conditions occurs, the corresponding signals will be set to 1 for exactly one cycle of the OCP clock. (The interrupt events are synchronous signals!) Thus, in most cases the frontend of the host will incorporate an interrupt logic that records the interrupts until they have been serviced by the host. The `SFlag` field consists of the following signals:

Signal	Interrupt
SFlag[6:0]	Port Number
SFlag[7]	Port Operation Complete
SFlag[8]	Global Reconfiguration Instant
SFlag[9]	Programmable Timer Interrupt

Global Reconfiguration Instant. This interrupt is triggered each time the TISS switches to a new communication schedule. This will happen at all TISSs at the same tick of the global time.

Programmable Timer Interrupt The programmable timer can be configured by the host to trigger an interrupt at a defined instant with respect to the global time.

Port Operation Complete. This interrupt is triggered whenever a message has been completely received or transmitted. This interrupt is triggered by an entry in the MEDL of the TISS, which is written by the TNA. For multicast or broadcast messages, the TNA assures that the interrupt is triggered at the same tick of the global time at each of the receivers.

Port Number. The port number indicates the corresponding port when the `Port Operation Complete` interrupt is triggered.

SReset_n. The **SReset_n** signal is a synchronous (sampled with the OCP clock), low-active reset signal through which the TISS can reset the host.

A.2.2 Port Configuration Memory

The port configuration memory is exclusively written by the host. It is used for configuring the local port parameters: the type of the port, the synchronization method, the port interrupts, the time stamp service, the queue length, and the port address. Globally relevant port parameters that are, like the send and receive instants or the route are written by the TNA and are, thus, not part of this memory.

The port configuration memory consists of consecutive entries (one for each port) which have a constant width of one data word (32 bit). The word address of an entry equals the number of the corresponding port. Thus the entry of a given port can be directly accessed without the need to search in the port configuration memory. In the current implementation, 128 ports are supported, with port number 125 being reserved for the watchdog, port 126 for the diagnostic dissemination service, and port 127 for the TISS CP interface which is written by the TNA. An entry in the port configuration memory consists of the following fields (see Figure A.4).

Port Enable. Enables or disables the port (0 ... port is disabled; 1 ... port is enabled)

Port Type. Sets the type of the port (0 ... state port; 1 ... event port)

Port Sync. Sets the synchronization method for state ports (0 ... explicit synchronization; 1 ... implicit synchronization). Explicit synchronization consumes double the amount of memory due to the required double buffer. In case of an event port this field is a "don't care" value.

Interrupt Enable. Enables or disables `Port Operation Complete` interrupt for that port (1 ... interrupts are enabled; 0 ... interrupts are disabled).

Time Stamp Enable. Enables or disables the time stamping services for that port (1 ... time stamps are enabled; 0 ... time stamps are disabled).

Port Base Address. Specifies the start address (in words) of the port in the port memory of the frontend.

QLength. Specifies the total size of an event port in data words (see Section A.1.2 for details on how the size is calculated)

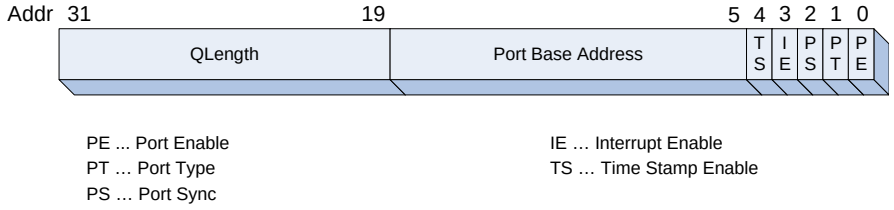

Figure A.4: Port Configuration Entry

A.2.3 Port Synchronization Flags

The port synchronization flags are used to synchronize the port accesses between the host and the TISS. Like the port configuration memory, the port synchronization flags are organized into consecutive entries (one for each port) which have a constant width of one data word (32 bit). The word address of an entry equals the number of the corresponding port. The interpretation of an entry of the port synchronization flags depends on the port type configured in the `Port Configuration Memory` (see Figure A.5).

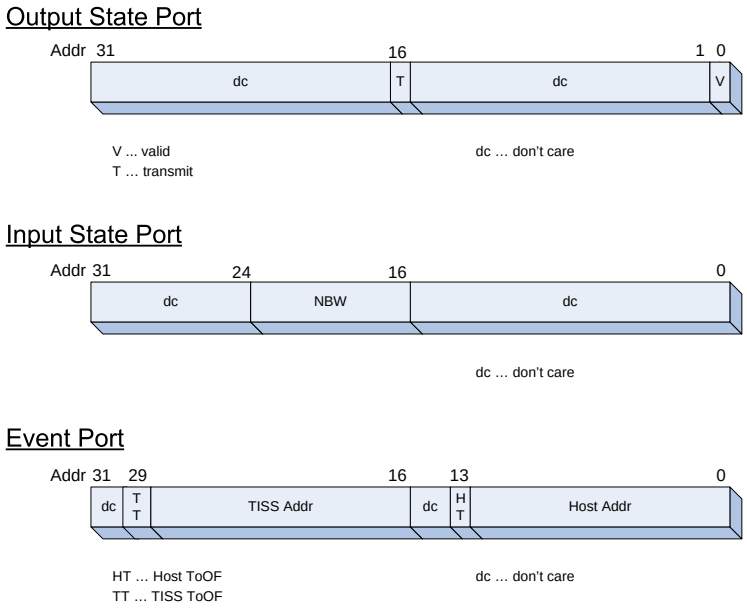

Figure A.5: Port Synchronization Flags

TISS Addr. This field is only significant, if the port is configured as an event port. In case of an *input* event port it specifies the write position; in case of an *output* event port it specifies the read position. The read or write position is interpreted as a word offset from the base address of the given port. The `TISS Addr` is incremented according to the following formulas where N denotes the message size and $QLength$ the total port size in 32 bit wide data words:

Time Stamps	Increment
disabled	$TISSAddr = (TISSAddr + N) \mod QLength$
enabled	$TISSAddr = (TISSAddr + N + 2) \mod QLength$

TISS ToOF. *(Toggle on Overflow of `TISS Addr`)* This field is only significant, if the port is configured as an event port. This bit is toggled by the TISS

each time the `TISS Addr` becomes 0 after it was incremented (i.e., after an overflow). This bit is used together with the `Host ToOF` bit to distinguish between an empty and a full queue (see Section 6.1.2).

NBW. This field is only significant if the port is configured as an input state port. It represents the sequencer for the NBW protocol (see Section 6.1.2).

Transmit. This field is only significant if the port is configured as an output state port with explicit synchronization. The semantics of the `transmit` bit is described in Section 6.1.2.

Host Addr. This field is only significant if the port is configured as an event port. For *input* event ports, it specifies the read position, for *output* event ports it specifies the write position. The read or write position is interpreted as a word offset from the base address of the given port. The `Host Addr` is incremented according to the following formula where N denotes the message size and $QLength$ the total port size in 32 bit wide data words.

Time Stamps	Increment
disabled	$HostAddr = (HostAddr + N) \mod QLength$
enabled	$HostAddr = (HostAddr + N + 2) \mod QLength$

Host ToOF. *(Toggle on Overflow of `Host Addr`)* This field is only significant if the port is configured as an event port. This bit is toggled by the host each time the `Host Addr` becomes 0 after it was incremented (after an overflow). This bit is used together with the `TISS ToOF` bit to distinguish between an empty and a full queue (see Section 6.1.2).

Valid. This field is only significant if the port is configured as an output state port with explicit synchronization. The `valid` bit is described in Section 6.1.2.

Under normal operation, the `TISS Addr`, `ToOF TISS`, `NBW`, and `transmit` fields can be exclusively written by the TISS and are read-only for the host. The host can only perform write accesses to these fields if the communication is disabled via the `Communication Disable` bit in the register file. This is required to reset the port flags after the host has changed the port configuration.

A.2.4 Register File

The register file, shown in Figure A.6, contains read-only and read/write registers, which are all 32 bit wide. The register file consists of the following

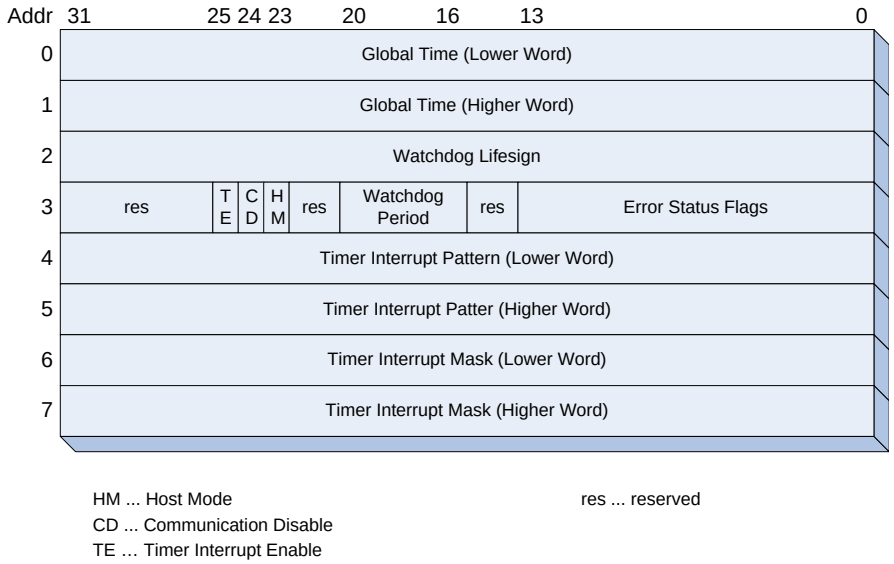

Figure A.6: Register File

fields.

Global Time. The 64 bit wide global time can be accessed via the first two (read-only) registers of the register file. In order to guarantee that the 64 bits are consistently read with two 32 bit accesses, the following synchronization method is employed: Whenever the lower 32 bits of the global time are accessed by the host, the upper 32 bits of the global time are frozen by copying them into shadow register. An access to the upper 32 bits of the global time will return the contents of this shadow register. Thus, the host will always read a consistent value as long as it accesses the lower word before the higher word, independently from the time that elapses between the two accesses.

Watchdog Life Sign. If the watchdog is enabled, the host has to update this register with the value "0x55555555" with a maximal period as defined in the `Watchdog Period` field. If the host fails to update this field within the defined period, it will be reseted by the TISS (via the **SReset_n** signal) and the failure will be recorded in the `Error Status Flags`.

Communication Disable. Via this bit, the host can disable all communication activities despite the reception of the TNA configuration message and the dissemination of the error status field (1 ... communication is disabled; 0 ... communication is enabled).

Timer Interrupt Enable. This bit enables or disables the programmable timer interrupt service (0 ... timer interrupt is disabled; 1 ... timer

interrupt is enabled). The programmable timer interrupt service should always be disabled while the `Timer Interrupt Pattern` and the `Timer Interrupt Mask` fields are configured to prevent invalid interrupts caused by inconsistent data in these fields during the configuration.

Host Mode. The Host mode is set by the TNA and specifies the on/off state of the host (1 ... host is turned on; 0 ... host is turned off). This bit controls either the power lines or the clock line of the host.

Watchdog Period. This field specifies the watchdog period. A value of "11111" indicates that the watchdog is disabled. This field is configured by the TNA and is read-only for the host.

Error Status Flags. Any error that is detected by the TISS will be recorded in the `Error Status Flags`. This field is read-only for the host and its content is periodically sent to the diagnostic unit by the diagnostic dissemination service of the TISS (see Section 6.2.5). After the content has been disseminated, the `Error Status Flags` are immediately reset. The dissemination period is configured by the TNA. The Error Status Flags consist of the following fields (see Figure A.7).

Queue Overflow Error. At least one overflow occurred.

Overflow-Port Number. The port number where the last overflow occurred. This field is only valid if `Queue Overflow Error` = 1.

Port Memory Error. A write or read operation of the TISS to the port memory in the frontend of the host has failed (e.g., due to a mis-configured address in the port configuration).

Port Configuration Error. A message was scheduled for a port that is disabled in the port configuration memory.

Communication Error. A message was scheduled while communication was disabled by the host by the `Communication Disable` bit.

Watchdog Miss. The host missed updating the watchdog for at least one period.

Repeated Error. At least one of the indicated errors has occurred more than once within the current error status dissemination period (see Section 6.2.5).

Timer Interrupt Pattern & Timer Interrupt Mask. These two fields configure the programmable timer interrupt service. The functionality of the programmable timer interrupt service is described in Section 6.2.2.

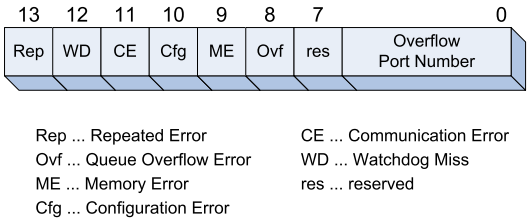

Figure A.7: Error Status Flags

Appendix B

List of Acronyms

AAM	Abstract Application Model
DAS	Distributed Application Subsystem
DMA	Direct Memory Access
DU	Diagnostic Unit
ECU	Electronic Control Unit
FCR	Fault Containment Region
FIM	Fully-Specified Interface Model
FPGA	Field Programmable Gate Array
FTU	Fault-Tolerant Unit
G-channel	Gateway Channel
G-port	Gateway Port
GPS	Global Positioning System
IP	Intellectual Property
IRQ	Interrupt Request
ITRS	International Technology Roadmap for Semiconductors
LIF	Linking Interface
MDA	Model Driven Architecture

MEDL	Message Descriptor List
MFIM	Macro FIM
NBW	Non-Blocking Write Protocol
NoC	Network-on-a-Chip
OCP	Open Core Protocol
ONA	Out-of-Norm Assertion
PAM	Physical Allocation Model
PIM	Platform Independent Model
PSM	Platform Specific Model
RCU	Replica Coordination Unit
RMA	Resource Management Authority
SoC	System-on-Chip
TDMA	Time Division Multiple Access
TISS	Trusted Interface Subsystem
TMR	Triple Modular Redundancy
TNA	Trusted Network Authority
TSS	Trusted Subsystem
TTSoC	Time-Triggered System-on-a-Chip
UNI	Uniform Network Interface
UFIM	Uniform FIM

Appendix C

Glossary

Abstract Application Model (AAM) The *Abstract Application Model* is an abstract representation of the system or a subsystem, where the interfaces of the individual subsystems are not fully specified, and thus, some design decisions are still left open (e.g., the selection of an adequate encryption method to achieve the desired security properties of a communication channel). In a subsequent step in the design process, the AAMs of all subsystems have to be transformed into a *FIM*, which includes the full specification of the *LIF* of each *job* of the system.

Application Computer The *application computer* is part of the *micro component*'s *host*. It provides the computational resources of the micro component and controls the micro component's local I/O interfaces (e.g., for sensors or actuators). It can be realized as a general-purpose microcontroller or FPGA or as a specialized hardware IP block (e.g., an MPEG encoder).

Application Service The *application service* is the intended sequence of messages that is produced by a *job* via output ports at the *LIF* and the *controlled object* interface in response to the progression of time, inputs (via input ports at the LIF and the controlled object interface), and state.

Architecture A technical *system architecture* (or architecture for short) is a framework for the construction of a system for a chosen application domain. It provides *core platform services* and imposes an *architectural style* for constraining an implementation in such a way that the ensuing system is understandable, maintainable, and extensible and can be built cost-effectively. (see also → *federated architecture*, → *integrated architecture*)

Architectural Style The architectural style consists of rules and guidelines for the partitioning of a system into *subsystems* and for the design of the interactions among subsystems. The subsystems must comply with the architectural style to avoid a property mismatch at the interfaces between subsystems.

Behavior The *behavior* of a subsystem is the sequence of messages (i.e., intended and unintended) that is produced by the subsystem at its *LIF*.

Cluster A *cluster* is a physically distributed computer system that consists of a set of *nodes* interconnected by a physical network. If the cluster supports a single DAS only, we speak of a federated cluster. In this case, the DAS is physically separated from the clusters of other DASs. Since the jobs belong to the same DAS, they possess a common level of criticality.

An *integrated* cluster, on the other hand, supports more than one DASs. Each of these DASs receives a share of the communication and component resources of the integrated cluster.

Controlled Object The *controlled object* is the industrial plant, the process, or the device that is to be controlled by the computer system.

Core Platform Services The *core platform services* (e.g., predictable transport of messages, global time service, watchdog service) are provided via the *UNI* and are independent of any particular *DAS*. They facilitate the development of distributed real-time applications and separate the application functionality from the underlying platform technology to reduce design complexity and to enable design reuse.

The core platform services can be adapted, refined, and extended by a *frontend*, which is a hardware element that translates the UNI to the interface of the attached *application computer*.

Declared State The *declared state* is the state of a subsystem, which is considered as relevant by the system designer for future behavior of the subsystem (forward view).

Diagnostic Unit (DU) The *Diagnostic Unit* is a dedicated *micro component* for the purpose of diagnosis. It performs failure detection at the *LIF* of the jobs by executing assertions on the syntactic, temporal, and semantic correctness of messages according to the DSoS message classification [GIJ+02]. Furthermore, the DU receives failure indication messages generated by other architectural elements of the SoC (e.g., *hosts*, *TISSs*, the *TNA*, or the *RMA*). A failure indication message includes information

concerning the type of the occurred failure (e.g., crash failure of a host), the time of detection w.r.t. to the global time base, and the location within the SoC (i.e., the micro component). Based on the gathered failure information, the DU establishes a holistic system view and executes *ONAs* to correlate the different failure indication messages in space and time.

Distributed Application Subsystem (DAS) A *Distributed Application Subsystem* is a nearly independent distributed subsystem of a large distributed real-time system that provides a well-specified application service. Examples of DASs in a present day automotive application are body electronics, the power-train system, and the multimedia system. Examples of DASs in a present day avionic application are the cabin pressurization system, the fly-by-wire system, and the in-flight entertainment system. DASs are often developed by different organizational entities (e.g., by different vendors) and maintained by different specialists. Since DASs may be of different criticality (e.g., vehicle dynamics control vs. multimedia system), the probability of error propagation across DAS boundaries must be sufficiently low to meet the dependability requirements.

A DAS is further decomposed into smaller units called *jobs*.

Event Message An *event message* is a message that contains event observations. An event observation contains the difference between the "old *state*" (the last observed state) and the "new state". The time of the event observation denotes the point in time of the state change. In order to maintain state synchronization, the handling of event messages requires exactly-once semantics. The arrival of an event message usually gives rise to a control signal, which triggers subsequent computational and communication activities.

Error Containment Although a *fault containment region* can demarcate the immediate impact of a fault, fault effects manifested as erroneous data can propagate across the boundaries of fault containment regions. For this reason the system must also provide *error containment* for avoiding error propagation by the flow of erroneous messages (→ *Error Containment Region*).

Error Containment Region The set of *fault containment regions* that performs error containment is denoted as an *error containment region*. An error containment region must consist of at least two independent fault containment regions. The error-detection mechanisms must be part of a different fault containment region than the message sender, otherwise the

error detection service can be affected by the same fault that caused the message failure.

Fault Containment Region (FCR) A *Fault Containment Region* is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region [LH94].

Fault Hypothesis The fault hypothesis is the specification of the *faults* that must be tolerated without any impact on the essential system services. The fault hypothesis states the assumptions about units of failure (→ *fault containment region*), failure modes, failure frequencies, failure detection, and *state recovery*.

Fault-Tolerant Unit (FTU) A unit consisting of a number of *replica determinate* micro components that provides the specified service even if some of its micro components fail.

Federated Architecture In a *federated architecture*, each DAS is implemented on a dedicated distributed computer system, consisting of nodes dedicated to jobs (in the automotive industry called Electronic Control Units - ECUs) and a physical network (e.g., a CAN network) among the nodes. In a federated architecture, each *DAS* is physically separated from other DASs, which leads to clear boundaries for responsibility and error propagation.

Frontend The *frontend* is part of the *micro component*'s *host*. It adapts, refines, or extends the *core platform services*, provided by the *UNI*, according to the requirements of the attached *application computer*. In its simplest version, the frontend is realized as a dual-ported memory providing a *temporal firewall interface* [Kop97] to the application computer. If required, the frontend can provide *higher-level services*, which are tailored to the needs of specific application domains. Examples are a fault tolerance service which performs majority voting of replicated inputs for failure masking by TMR, or an encryption and decryption service to facilitate secure communication with chip external entities.

Fully-Specified Interface Model (FIM) The *Fully-Specified Interface Model* describes the functionality and the interaction patterns of the individual jobs of a system by a behavioral specification, including temporal constraints. It does not include any information about the micro components on which the jobs will be executed and abstracts from micro component specific implementation details of the jobs (e.g., a micro component can be realized as a special purpose microcontroller, as an FPGA or as a special purpose hardware IP block).

The TTSoC architecture defines two different types of FIMs to describe a system at two different levels of abstraction, the UFIM and the MFIM.

Gateway Channel (G-channel) A *Gateway Channel* is a unidirectional communication channel that transports messages from a single source SoC to one or more destination SoCs (i.e., a channel between gateways of different SoCs).

Gateway Port (G-port) A *Gateway Port* is an endpoint of a *G-channel*.

Host The *host* performs the computations that are required to deliver the intended service of a *micro component*. It is structured into two architectural elements, the *application computer* and the *frontend*.

Integrated Architecture An *integrated architecture* is characterized by the integration of multiple *DASs* within a single distributed computer system. An integrated architecture possesses a single physical network that is exploited for the construction of multiple virtual networks. In the TTSoC architecture, architectural services are employed to encapsulate DASs and restore the complexity management advantages and natural error containment between DASs of a federated architecture.

Interface State The *interface state* contains the history of the component that is relevant for the future behavior of the component as seen from this interface. Interface state is defined between the intervals of activity on the *sparse time base*. Interface state is a subset of the state of the component and should be accessible from the interface.

Job A job is a subsystem of a *DAS* and the basic unit of distribution (i.e., a single job cannot be distributed on multiple micro components). It is the object of temporal and spatial partitioning and interacts with other jobs solely by the exchange of messages through its *LIF*.

An example for a job in a safety-critical brake-by-wire DAS of a car would be the software, which fits into a single micro component, for computing the brake force based on the actual wheel slip. For fault-tolerance reasons, multiple instances of the job will be executed redundantly at different components, e.g., three instances in a triple-modular redundancy configuration.

Linking Interface (LIF) A *job* provides its real-time services, and accesses the real-time services of other jobs by the exchange of messages across its *Linking Interface*. These messages have to be fully specified in a *LIF specification* which consists of an *operational* specification and a *meta-level* specification [KS03]. While the operational specification deals with

the syntactic and temporal aspects of the messages exchanged across the LIF, the meta-level specification describes the meaning of the information contained in these messages.

Macro FIM The *Macro FIM* is a high-level representation of the *FIM*. It facilitates the modeling of *DASs* at a higher level of abstraction than the *UFIM*, by providing macros that translate high-level constructs into constructs supported in the UFIM. Thus, the interface specification of the jobs in the MFIM can rely on higher-level domain-specific services like voted channels for fault tolerance, encrypted channels for security, or bidirectional channels for request/reply transactions. The *MFIM meta model* (→ *meta model*) can exist in multiple variations supporting different sets of domain-specific services. For each MFIM meta model, a set of transformation rules has to be specified that define the transformation of a MFIM to an equivalent UFIM.

Message Descriptor List (MEDL) The *Message Descriptor List* is a data structure within each *TISS* that determines when a message must be sent on, or received from, the NoC.

Micro Component A *micro component* is a self-contained computational unit that provides its functionality over a well defined *message-based* interface. It is composed out of two structural elements, the TISS and the *host*. While the host performs the computations, which are required to deliver the intended service of the micro component, the TISS provides a stable set of *core platform services* to the host. Furthermore, the TISS acts as a guardian for the NoC by ensuring that a fault within the host of a micro component cannot lead to a violation of the micro component's temporal interface specification in a way that the communication between other micro components would be disrupted.

MFIM-channel *MFIM channels* are used to describe the communication between *jobs* in the *MFIM*. Contrary to the *UFIM-channels* in the *UFIM*, MFIM channels are not restricted to be unidirectional. Their characteristics are determined by the chosen MFIM meta model (→ *meta model*).

MFIM-port An *MFIM port* is an endpoint of an *MFIM-channel*. A *job* in the *MFIM* can have multiple MFIM-ports since it can be attached to multiple MFIM-channels.

Non-Blocking Write Protocol (NBW) The *Non-Blocking Write Protocol* is a synchronization protocol between a single writer and many readers. It achieves data consistency without blocking the writer [Kop97].

Out-of-Norm Assertion (ONA) An *Out-of-Norm Assertion* detects anomalous component behavior that cannot be judged as correct or faulty at the time of occurrence. Out-of-norm assertions operate on the output messages and the interface state and encode fault patterns on the consistent distributed state induced by a sparse time base and are specified in the dimensions of value, time and space.

Meta Model A *meta model* defines the rules and constructs according to which a model is created.

Physical Allocation Model (PAM) The *Physical Allocation Model* is a more concrete system representation than the FIM. It describes the mapping of the FIM (to be more precise, the UFIM) to the physical system structure. Contrary to the UFIM, the PAM is tailored to the specific characteristics of the micro component on which a job should be executed. Nevertheless, the semantic and syntactic properties of a job's LIF in the PAM are exactly the same as in the UFIM. The temporal properties of a job's LIF in the PAM are fully specified and satisfy the temporal constraints defined in the UFIM.

Platform-Independent Model (PIM) A *Platform Independent Model* is a model of a system that is independent of the specific technological platform used to implement it.

Platform-Specific Model (PSM) A *Platform Specific Model* is a model of system that is linked to a specific technological platform.

Replica Determinism *Replica determinism* is a desired property between replicated subsystems. A set of replicated subsystems is replica determinate if all subsystems in this set produce exactly the same output messages that are at most an interval of d time units apart, as seen by an omniscient outside observer. In a time-triggered system, the subsystems are considered to be replica-deterministic if they produce the same output messages at the same global ticks of their local clock [Kop97].

Resource Management Authority (RMA) The *Resource Management Authority* is, besides the *TNA*, one of the two dedicated architectural elements for resource management. It accepts *resource request messages* from the jobs and generates, according to internal rules, a resource allocation mapping for the entire SoC.

SoC-Channel The term *SoC-Channel* denotes an encapsulated unidirectional communication channel in the physical system structure that transports messages at predefined points in time from a single source micro

component to one or more destination micro components within the same SoC (SoC-Channels cannot cross the boundaries of a single SoC).

SoC-Port An *SoC-port* is an endpoint of an *SoC-channel*. A *micro component* can have multiple SoC-Ports since it can be attached to multiple SoC-channels.

Sparse Time Base If the time base of the global time in a distributed system is *dense* (i.e., the events are allowed to occur at any instant of the timeline), then it is in general not possible to generate a consistent temporal order of events on the basis of the time-stamps. Due to the impossibility of synchronizing clocks perfectly and the denseness property of real time, there is always the possibility that a single event is time-stamped by two clocks with a difference of one tick. By introducing the concept of a *sparse time base* [Kop92] this problem can be solved. In the sparse time model the continuum of time is partitioned into an infinite sequence of alternating durations of activity (π) and silence (Δ). Thereby, the occurrence of significant events is restricted to the activity intervals of a globally synchronized action lattice. In this time model, the costly execution of agreement protocols can be avoided, since every action is delayed until the next lattice point of the action lattice.

State The *state* enables the determination of a future output solely on the basis of the future input and the state the system is in. In other word, the state enables a "decoupling" of the past from the present and future. The state embodies all past history of the given system. Apparently, for this role to be meaningful, the notion of the past and future must be relevant for the system considered (taken from [MT89, p. 45]) ($\rightarrow$ *declared state*, $\rightarrow$ *interface state*).

State Message A *state message* is a periodic message that contains state observations. An observation is a state observation, if the value of the observation contains the state of a real-time entity. The time of a state observation denotes the point in time when the real-time entity was sampled. The handling of state messages occurs through an update in place and non-consuming read.

State Recovery *State recovery* is the action of (re-)establishing a valid *state* in a *subsystem* after a *failure* of that subsystem.

Subsystem A subsystem is a part of a system that represents a closure with respect to a given property.

Trusted Interface Subsystem (TISS) The *Trusted Interface Subsystem* is part of the *micro component* and provides a stable set of *core platform*

services via the *UNI*. Furthermore, it acts as a guardian for the NoC by ensuring that a fault within the *host* of a micro component (e.g., a software fault) cannot lead to a violation of the micro component's temporal interface specification in a way that the communication between other micro components would be disrupted.

Trusted Network Authority (TNA) The *Trusted Network Authority* is, besides the *RMA*, one of the two dedicated architectural elements for resource management and is responsible for the (re-)configuration of the *TISSs* and the NoC. It checks the resource allocation proposal, provided by the RMA, against a set of predefined constraints (e.g., conflict-freeness of the message schedule or availability of statically assigned resources for safety-critical application subsystems). If the mapping is valid, the TNA (re-)configures the NoC and the TISSs accordingly.

Trusted Subsystem (TSS) The *Trusted Subsystem* consists of the *TNA*, the time-triggered NoC, and the *TISSs*. The TSS is assumed to be free of design faults and has to be certified according the criticality level of the most critical micro component in the SoC.

Uniform Network Interface (UNI) The *Uniform Network Interface* is the basic architectural interface of the TTSoC architecture. It is located between the *TISS* and the *host*. The UNI provides a set of *core platform services* which facilitate the development of distributed real-time applications and separate the application functionality from the underlying platform technology to reduce design complexity and to enable design reuse.

Uniform FIM (UFIM) The *Uniform FIM* is a uniform representation of the *FIM*. It describes the system at the level of the *UNI*. This means that, with respect to the interface specification of the jobs, the *UFIM meta model* (→ *meta model*) defines exclusively constructs that refer to the communication services that are natively provided by the UNI (e.g., unidirectional communication channels). The specification of a job in the UFIM serves as a contract between the *system integrator* and the *job developer* and can be used for conformance testing.

UFIM-channel *UFIM-channels* are used to describe the communication between *jobs* in the *UFIM*. The term UFIM-channel denotes an encapsulated unidirectional communication channel that transports messages under predefined temporal constraints (e.g., latency, period, absolute phase offset to the start of the period, or relative phase offset to another channel) from a single source job to one or more destination jobs.

UFIM-port A *UFIM-port* is an endpoint of a *UFIM-channel*. A *job* can have multiple UFIM-ports since it can be attached to multiple UFIM-channels.

Bibliography

[ARM99] ARM. AMBA Specification Rev. 2.0, 1999.

[ARM01] ARM. Multi-layer AHB Overview, 2001.

[ARM04] ARM. AMBA AXI Protocol Specification V 1.0, 2004.

[BCDGG00] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *Transactions on Computers*, 49:230–245, March 2000.

[Bir03] S. Birch. Pre-safe headlines S-Class revisions. *Automotive Engineering*, 111(1):15–18, 2003.

[BM06] L. Benini and G. Micheli. *Network on Chips: Technology and Tools*. Morgan Kaufmann Publishers, 2006.

[Bor07] S. Borkar. Thousend core chips—a technology perspective. In *Proceedings of the 44th Design Automation Conference - DAC*. ACM, June 2007.

[Bos91] Robert Bosch Gmbh, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.

[Bro95] F.P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley, 1995.

[BT04] J. Boudec and P. Thiran. *Network Calculus, A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2004.

[Dei02] A. Deicke. The electrical/electronic diagnostic concept of the new 7 series. In *SAE Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, 2002.

[Fle05] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.

[GDR05] K. Goossens, J. Dielissen, and A. Radulescu. The æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005.

[Ger02] C. Gershenson. Complex philosophy. In *Proceedings of the 1st Biennial Seminar on Philosophical, Methodological & Epistemological Implications of Complexity Theory*, La Habana, Cuba, 2002.

[GIJ+02] M. C. Gaudel, V. Issarny, C. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and F. Taiani. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585) Deliverable CSDA1*, December 2002. Available as Research Report 54/2002 at http://www.vmars.tuwien.ac.at.

[Ham03] R. Hammett. Flight-critical distributed systems: Design considerations [avionics]. *IEEE Aerosdpace and Electronic Systems Magazine*, 18(6):30–36, 2003.

[Hof05] H. Peter Hofstee. Introduction to the cell broadband engine. Technical report, IBM Corporation, 2005.

[ITR05] ITRS. The international roadmap for semiconductors—2005 edition. Technical report, 2005.

[KAGS05] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The Time-Triggered Ethernet (TTE) design. *8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, May 2005.

[KG94] H. Kopetz and G. Grünsteidl. TTP – a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994. Vienna University of Technology, Real-Time Systems Group.

[KO87] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933–940, 1987.

[KO02] H. Kopetz and R. Obermaisser. Temporal composability. *IEEE's Computing & Control Engineering Journal*, 2002.

[Kop92] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of 12th International Conference on Distributed Computing Systems*, Japan, June 1992.

[Kop97] H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, 1997.

[Kop05] H. Kopetz. A time-triggered SoC-platform for distributed embedded application. Research Report 34/2005, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2005.

[Kop06] H. Kopetz. Pulsed data streams. In *Working Conference on Distributed and Parallel Embedded Systems (DIPES)*, pages 105–124, Braga, Portugal, October 2006. Springer.

[KS03] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60, May 2003.

[LH94] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. In *Proceedings of the IEEE*, volume 82, pages 25–40, January 1994.

[LV62] R.E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200, April 1962.

[MC02] Germany MOST Cooperation, Karlsruhe. MOST Specification Version 2.2, 2002.

[MGRG+04] K.D. Muller-Glaser, C. Reichmann, P. Graf, M. Kuhl, and K Ritter. Heterogeneous modeling for automotive electronic control units using a case-tool integration platform. In *IEEE International Symposium on Computer Aided Control Systems Design*, pages 83–88, September 2004.

[Moo89] J.F. Moore. Arinc 629, the civil aircraft databus for the 1990s. In *IEE Colloquium on Time Critical Communications for Instrumentation and Control*, 1989.

[MT89] M. D. Mesarovic and Y. Takahara. *Abstract Systems Theory*. Springer-Verlag, 1989.

[Nee02] Needham, R. M. . Naming. In Sape Mullender, editor, *Distributed Systems*, pages 315–327. ACM Press, second edition, 2002.

[OCP05] OCP-IP Association. Open Core Protocol Specification 2.1, 2005.

[OD83] D. C. Oppen and Y. K. Dalal. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Information Systems*, 1(3):230–253, 1983.

[OKESH07] R. Obermaisser, H. Kopetz, C. El Salloum, and B. Huber. Error containment in the time-triggered system-on-a-chip architecture. In *Proceedings of the International Embedded Systems Symposium (IESS'07)*, Irvine, CA, USA, May 2007.

[OMG03] OMG. MDA guide version 1.0.1, 2003.

[OPHES06] R. Obermaisser, P. Peti, B. Huber, and C. El Salloum. DECOS: An integrated time-triggered architecture. *e&i journal (journal of the Austrian professional institution for electrical and information engineering)*, 3, March 2006.

[PM05] M. Posner and D. Mossor. Designing using the AMBA 3 AXI protocol. Synopsys, 2005.

[PMH98] B. Pauli, A. Meyna, and P. Heitmann. Reliability of electronic components and control units in motor vehicle applications. In *VDI Berichte 1415, Electronic Systems for Vehicles*, pages 1009–1024. Verein Deutscher Ingenieure, 1998.

[Pol94] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6:289–316, 1994.

[Pol96] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.

[Pol99] F. J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[RDG+] A. Radulescu, J. Dielissen, K. Goossens, E. Rijpkema, and P. Wielage. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network configuration.

[Rob91] Robert Bosch Gmbh, Stuttgart, Germany. CAN Specification, Version 2.0, 1991.

[Ros90] M. T. Rose. *The Open Book: A Practical Perspective on OSI.* Prentice Hall International, 1990.

[SM98] J. Swingler and J.W. McBride. The synergistic relationship of stresses in the automotive connector. In *Proceedings of the 19th International Conference on Electric Contact Phenomena*, pages 141–145, 1998.

[Son02] Sonics. Sonics μNetwork technical overview (http://www.sonicsinc.com), 2002.

[SSA98] A. Silva, P. Sousa, and M. Antunes. Naming: Design pattern and framework, 1998.

[SV02] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[SWH95] N. Suri, C.J. Walter, and M.M. Hugue. *Advances In Ultra-Dependable Distributed Systems*, chapter 1. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264, 1995.

[Tv03] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall International, 2003.

[VHR+07] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. "an 80-tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS". In *IEEE International Solid-State Circuits Conference, ISSCC*, February 2007.

[vN56] J. von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[WdJ92] Roel Wieringa and Wiebren de Jonge. The Identification of Objects and Roles – object identifiers revisited, 1992.

[WdJ95] Roel Wieringa and Wiebren de Jonge. Object identifiers, keys, and surrogates: Object identifiers revisited. *Theory and Practice of Object Systems*, 1(2):101–114, 1995.

[Wik05] D. Wiklund. *Development and Performance Evaluation of Networks on Chip*. PhD thesis, Department of Electrical Engineering Linköping University, SE-581 83 Linköping, Sweden, 2005.

[Wik07] Wikipedia, the free Encyclopedia. Roman naming conventions. Wikimedia Foundation, June 29 2007.

Curriculum Vitae

Christian El Salloum

October 31^{th}, 1975	Born in Vienna (Austria)
September 1982 – June 1986	Elementary School in St. Aegyd (Austria)
September 1986 – October 1995	Comprehensive Secondary School in Lilienfeld (Austria)
October 1995 – May 1996	Military Service in Spratzern (Austria)
October 1996 – January 2003	Studies of Computer Science at the Vienna University of Technology with distinction
March 2002 – July 2002	Exchange Student in Computer Science at the École nationale supérieure de l'aéronautique et de l'espace Toulouse (France)
June 2003	Master's Degree in Computer Science
since April 2003	PhD Studies and Research/Teaching Assistant at the Vienna University of Technology