MASTER THESIS

An Extensible Interface Subsystem For A Novel Time-Triggered System-on-a-Chip Architecture

fulfilled at the institute of

Computer Engineering, Real-Time Systems Group

of the

Vienna University of Technology

under the guidiance of

O. Univ.-Prof. Dr. phil Hermann Kopetz

and

Univ.Ass. Dipl.-Ing. Christian El Salloum

by

Bakk. techn. Roman Seiger

Overbeckgasse 37

A-1130 Wien

Vienna, August 2007 .....

An Extensible Interface Subsystem For A Novel Time-Triggered System-on-a-Chip Architecture

The Time-Triggered System-on-a-Chip (TTSoC) architecture provides a predictable integrated execution environment for the component-based design of many types of embedded applications. It was inspired by the experience gained in previous related research efforts, like the DECOS project, which focused on the integration of mixed-criticality application subsystems. The TTSoC architecture provides a predictable time-triggered Network-on-a-Chip (NoC) to maintain strict encapsulation of the different cores and to support safety-critical real-time applications. The encapsulation mechanisms facilitate to manage the rising complexity of today's and future embedded systems by allowing a larger system to be broken down into smaller subsystems that can be designed independently. The NoC establishes a message-based, a priori scheduled communication between different cores. This thesis focuses on the design and implementation of the NoC interface. In addition, possible extensions used to improve the functional range of the TTSoC, Middleware Plug-Ins, are introduced. To show the advantages of this concept, a Direct Memory Access (DMA) Plug-In and, to support application systems requiring Triple Modular Redundancy, a Voter Plug-In are described.

Contents

List of Figures

List of Tables

1 Introduction

Over the last years, an increasing amount of embedded computer systems were designed and integrated into consumer products, especially in the automotive industry. In a competitive economic environment where time-to-market issues and product originality are the driving factors for development, embedded computer systems, which, among other advantages, offer the possibility to improve a car's performance envelope while optimizing its fuel efficiency and to introduce additional comfort functions, could achieve a sustained, extraordinary success. This trend, supported by the semiconductor industry which made a vast progress in developing smaller, more powerful, and more energy efficient systems, is continuing or even increasing, although not all of its effects are beneficial. Among the problems attached to the inflationary replacement of mechanical or hydraulic systems by embedded computer systems are rising development and electronic hardware costs, dependability requirements, and Intellectual Property (IP) protection [OPHS06, p. 84], to point out only some of them.

With these challenges in mind, the Dependable Embedded Components and Systems (DECOS) project [Dep] focuses on the development of an integrated, distributed architecture capable of managing arising complexity issues as well as economical difficulties. One aspect of the DECOS architecture is to brake down the whole application system into so called Distributed Application Subsystems (DASs). Each DAS consists of a number of jobs and is responsible for providing a meaningful part of the overall service of the application system. This and other concepts behind the DECOS architecture and a number of experiences gained during the DECOS design process were reviewed carefully, leading to the decision to further improve the DECOS architecture. The intended goal was to develop a System-on-a-Chip (SoC) capable of housing jobs of different Distributed Application Subsystems with different criticality levels on the same microchip while preserving strong encapsulation of the single DASs.

By adapting the well established Time-Triggered Architecture (TTA) [KB03] the design led to the Time-Triggered System-on-a-Chip (TTSoC) architecture, an SoC housing multiple potentially heterogenous cores. Predictable communication among the single cores of the TTSoC is provided by a deterministic time-triggered Network-on-a-Chip (NoC). A chip-wide global time service which can be synchronized to external time references (e. g., the Global Posi-

tioning System (GPS) time) allows to temporally coordinate the action of multiple cores. Among other additional services, strict encapsulation of the cores to prevent any unintended interference is one of the major goals of the Time-Triggered System-on-a-Chip architecture. As a guard for the time-triggered NoC, the Trusted Interface Subsystem (TISS) is introduced. The TISSs protect the NoC from cores violating their temporal specification and furthermore provide a small and stable set of generic services which are required by the different cores. These generic services can be customized by adding Middleware Plug-Ins taylored to the required application domain (e. g., fault tolerance, security, etc.).

Dynamic resource management (i. e., allocation of computational and communication resources at runtime) in the TTSoC architecture is carried out by a Resource Management Authority (RMA) and a Trusted Network Authority (TNA) as a two-stage process. While the RMA dynamically builds and adapts the resource allocation, the certified TNA checks this allocation for conflicts and ensures that safety-critical functions are always provided with sufficient resources.

Since every core in the TTSoC architecture hosts only a single job of a DAS and unintended interference between DASs is prevented by the TTSoC architecture, each DAS can be designed independently from the other DASs, limiting the cognitive complexity of the design process and simplifying the final integration of the Distributed Application Subsystems.

By introducing gateways, multiple TTSoCs can be combined to form a cluster in a higher-level communication system (e. g., Time-Triggered Ethernet [Ste06]), therefore a Distributed Application Subsystem can be spread among different (physically separated) TTSoCs.

To make use of the communication service of the Network-on-a-Chip and the various other services the TTSoC architecture provides, a standardized host processor interface is required. To adapt to the changing demands of different applications, the interface subsystem of the TTSoC architecture has to provide possibilities to extend it with additional, customized hardware components providing domain specific higher-level services. This thesis describes the concept and the implementation of such an extendible interface subsystem for the Time-Triggered System-on-a-Chip architecture.

The thesis is structured as follows: Chapter 2 points out some of the background issues that led to and/or influenced the development of the Time-Triggered System-on-a-Chip architecture, which is described in chapter 3. Being the main subject of this thesis, the interfaces (sections 4.2 and 4.2.2.2) as well as the implementation (section 4.3) of the LLC and CNI Layers are

discussed in chapter 4. To enlarge the functional range of the TTSoC architecture, the CNI Layer was designed to be extendable. Chapter 5 deals with the different kinds of CNI Layer extensions. Chapter 6 offers a conclusion of the thesis, and gives some suggestions for future developments concerning the Time-Triggered System-on-a-Chip architecture. Appendices A, B, and C conclude this thesis with detailed information on the Open Core Protocol (OCP), an Avalon Adapter, and a Powerlink Bus Adapter, respectively.

2 Background

2.1 Time-Triggered Communication

An embedded system has to be aware of the progression of time if its services should be of any use in the real world. Any results presented by an embedded system have to be valid not only in the value domain, but also have to be "on time". Otherwise, no use can be made of the results, even if correct, since the time in the real world advanced and the achieved results became irrelevant.

To avoid such situations, embedded systems can be designed with sufficient processing power to guarantee completion of their tasks before reaching the deadline at which the produced results would become outdated. Very detailed analysis of the system's timely behavior (reaction time, worst case execution time (WCET), etc.) and the time constraints arising from the application environment (i. e., the deadlines that have to be met) become necessary to ensure dependable real-time behavior.

The efforts to guarantee timeliness rise significantly when dealing with larger distributed embedded systems. Communications between components of a distributed system and the possibility of unpredictable behavior due to indeterministic arbitration or scheduling algorithms, along with possible inherent indeterminsim of the application itself make matters worse.

A more promising approach to cope with the increasing effort in designing and evaluating real-time application sytems is not only to design embedded systems that are aware of real-time, but to make the notion of time an integral part of a dependable distributed embedded system architecture.

The Time-Triggered Architecture (TTA) [KB03] provides a solution to the increasing complexity of large real-time embedded systems by decomposing such a system into nearly autonomous nodes and clusters and introducing a fault-tolerant global time base used for interface specification, error detection, communication protocols, and to guarantee the timeliness of real-time applications. Research on the TTA started at the Technical University of Berlin in 1979 and continued at the Vienna University of Technology since 1982. The TTA has been further developed ever since, including implementation of several

prototype and even commercially used systems. To ensure further development and marketing of the time-triggered technology, a spinoff company of the Vienna University of Technology, the TTTech Computertechnik AG [TTTb] was founded in 1998.

Two time-triggered communication services are described by the TTA, the TTP/C [TTTa] and the TTP/A [EEE+01] protocol. The TTP/C protocol is a fault-tolerant time-triggered communication protocol connecting several distributed nodes of a system to form a cluster and providing fault-tolerant clock synchronization between the nodes, a membership service to monitor the systems "health-state", and message transport with known delay and bounded jitter. The TTP/A protocol is a time-triggered fieldbus protocol which is used to connect low-cost smart transducers forming a transducer cluster to a so called master node. This master node can act as a gateway to a communication system on a higher level, like a TTP/C cluster (figure 2.1).

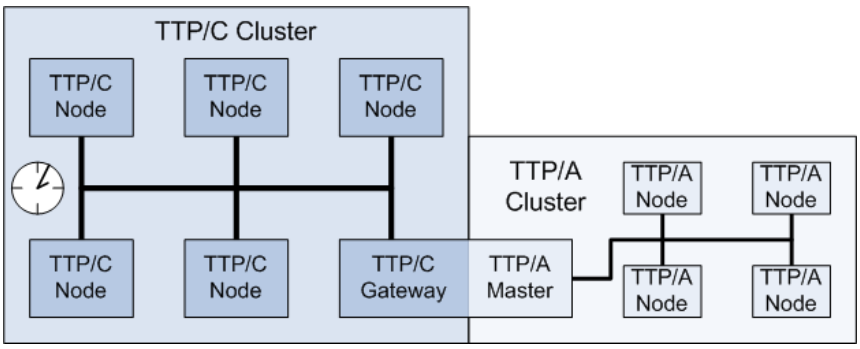

Figure 2.1: TTP/C Cluster with TTP/A Transducer Cluster

Some concepts of the Time-Triggered Architecture model which largely influenced the development of the Time-Triggered System-on-a-Chip architecture are described in more detail in the following paragraphs.

Time

The key concept of the TTA is the use of (physical) real-time as an integral part of the system.

Any happening occurring at a specific time instant, either in the real world (e. g., turning on a switch) or inside the system (e. g., reception of a message), is called an event. A distribiuted real-time system has to be capable of temporally ordering these events, even if they happen at different nodes which can be distributed over multiple SoCs. To accomplish this task, a system-wide (global) time base, synchronized among all nodes, is needed. By using the same global

time in each node, every event can be timestamped and a temporal order of events can be maintained throughout the system.

The difficulties in using a global time base in distributed nodes are the impossibility to perfectly synchronize distributed clocks and the denseness of real-time, i. e., no matter how small the clock granularity is, an event can still occur in between two clock ticks and, as a consequence, can be timestamped differently by two different nodes. Since clocks with an infinetely small granularity are physically impossible, the TTA introduces a sparse time base [Kop04b, p. 55].

Time is divided into an infinite sequence of activity and silence intervals in the sparse-time model, in which the duration of the activity interval has to be larger than the precision of the clock synchronization. Figure 2.2 depicts the concept of a sparse time base.

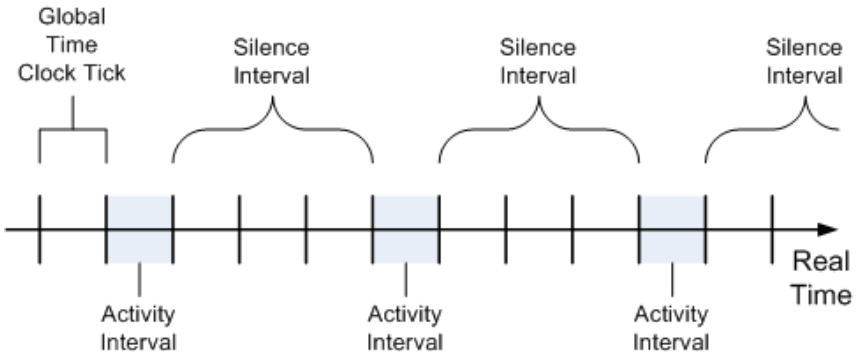

Figure 2.2: Sparse Time Base

Any events that happen in the same activity interval (at different nodes) are considered simultaneous by the system, events occurring in different activity intervals can definitely be temporally ordered. All events triggered by the system itself (e. g., sending of a message) are restricted to the activity intervals, while events happening during a silence interval outside of the sphere of control of the nodes have to be aligned to the sparse time base by using an agreement protocol.

State Information vs. Event Information

Embedded systems interact with their environment by observing its properties and reacting to them. These observations are done by a variety of different kinds of sensors (e. g., switches, temperature sensors, etc.) which examine the current state of the environment. The examination results can be stored or

transmitted inside the system by means of event or state information, enclosed in state or event messages, respectively.

An event is a change of the state of an entity at a specific time instant, therefore event information consists of the name of the observed entity, the time of the event, and the difference between the state before and after the event.

Since the current state of an entity is determined by the initial state and all state changes that occurred since then, all events have to be recorded by an application to ensure a consistent view of its environment. This implies the need for messages containing event information to be sent and received exactly once. Any lost or repeated event message leads to a wrong internal image of the actual state of the observed entity, hence a communication protocol supporting and ensuring exactly-once semantics is required.

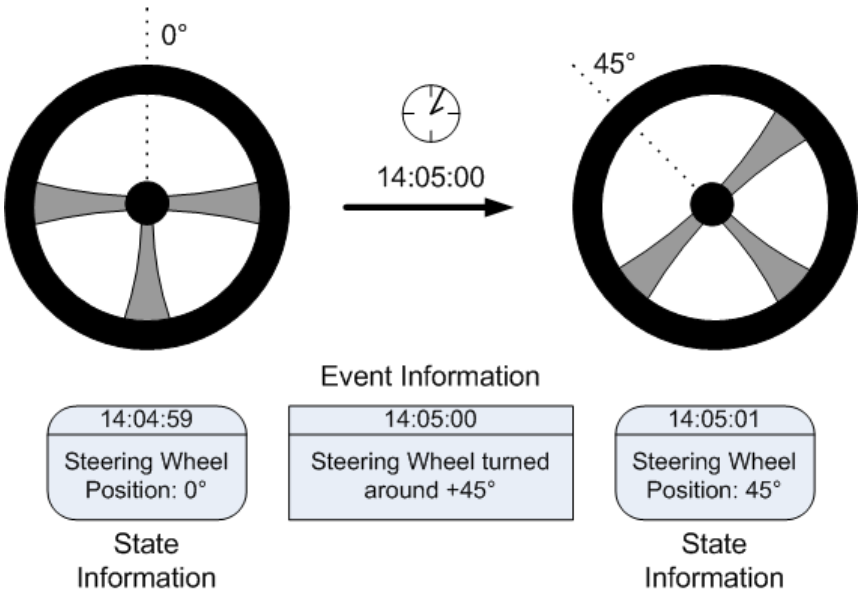

Figure 2.3: State Information vs. Event Information

State information consists of the name of the observed entity, the time of the observation, and the state of the entity at observation time. The initial state of the observed entity and/or previous state changes leading to the current state are inherently enclosed in this information. As a consequence, state messages are idempotent and require at-least-once semantics. Repeated state messages do not corrupt the internal image of the observed entity, but lost state messages may lead to an internal image which is not up-to-date due to missed state

changes. The difficulty arising with state messages is to determine the optimal update interval of the state information. An update interval which is too long may lead to missed state changes of the observed entity (e. g., a temperature spike at a sensor in between two state messages may not be recognized by a controlling component), whereas a too short update interval may produce unnecessary traffic on the communication system.

Time-Triggered vs. Event-Triggered Communication

Regardless of the type of information contained in a message, the transmission of the message can be triggered by either reaching a specific time instant or by the occurrence of an (external) event.

Time-triggered communication is characterized by a schedule which determines the transmission times of certain messages, usually organized in time periods and phase offsets. These periods and phases can be calculated a priori and remain constant during the whole system operation time, or the schedule can be changed regularly to adapt to different communication requirements throughout the system operation time. By nature, a time-triggered communication system transports state messages in regular intervals, no matter if a state change at the observed entity has occurred.

The advantage of the time-triggered approach is that any message sent over the communication system can be identified solely by its send time instant. Neither a sender nor a receiver identification is required to be enclosed in the message since the time of the send instant inherently identifies the message. The active schedule, which is common knowledge at the sending and the designated receiving core(s), is examined with respect to the current time and subsequently triggers a send or receiving operation. Furthermore, a missing message due to an erroneous core which missed a scheduled send slot can easily be detected by a diagnostic entity or by the other cores, simply by not receiving any message. Corrective actions can be commenced immediately to protect the whole system from failing (e. g., restart or disabling of the erroneous core, activation of backup cores, entering a system-wide fail-safe state, etc.).

The difficulty of a time-triggered communication system is to establish a synchronized distributed global time base among all cores with sufficient precission. With respect to dependable distributed embedded systems, the required clock synchronization has to be fault-tolerant to ensure communication, and thus (possibly reduced) system operation, despite the presence of arbitrary (i. e., Byzantine) faults. Another necessity for dependable operation is to restrict core accesses to the communication medium to their designated sending

intervals. A core trying to send a message outside of its sending interval will most likely corrupt the data sent by another core, or even mask out core failures by preventing detection of a missing message. The TTA introduces guardians (bus guardians attached to each core in case of a bus topology or guardians located in the center of a star topology network) to protect the system from arbitrary core failures (e. g., "babbling idiot" failures). These guardians ensure fail-silent behavior of all cores by restricting the communication medium access to the respective sending intervals and blocking any other sending attempt.

Event-triggered communication does not follow a designated schedule. The sending of an event message is triggered by the occurrence of the associated event. Such events can have an external source (e. g., a user pressing a button, a temperature value exceeding a threshold) or can be triggered by the embedded system itself (e. g., completion of a calculation).

Since an event-triggered communication system only starts the transmission of a message if an event (a state change) occurred, traffic on the network can be significantly reduced as long as the frequency of occurring events is sufficiently low, in contrast to a time-triggered communication system, which transmits messages regardless of any state change. This allows not only for a more efficient use of the available network bandwidth, but also helps to save power, which is a crucial requirement of embedded systems.

On the downside of event-triggered communication systems is the need to enclose sender and/or receiver identification information in every message, increasing the bandwidth usage. In addition, the gain in available bandwidth can be limited by the peak-load of the network (worst-case scenario, usually reached if all cores try to send simultaneously). If the communication system is designed with enough spare bandwidth to handle the rare-event peak-load scenario without violating any system deadlines, the spare bandwidth is useless and wasted most of the operational time of the system (assuming the peak-load scenario is rare under normal operating conditions). By designing the communication system to make a more efficient use of the available bandwidth under normal conditions while abandoning the ability to fulfill all deadlines under higher load, communication integrity can not be guaranteed under peak-load scenarios. This is a design decision which has to be taken according to the intended use of the embedded system (e. g., safety-critical, i. e., (partially) operational under all circumstances, or accepting possible system breakdowns due to high load).

The lack of a priori knowledge of send instants prevents the communication system from detecting a failed core by monitoring missing messages. A fail-silent core shows the same behavior when failed as a working core when no events occur at its observed entity. Even worse, cores with an arbitrary

failure mode may block the communication system by uninterrupted sending of messages ("babbling idiot" behavior). As a consequence, the application has to perform failure detection inside the cores by means of restricting network access and/or some kind of fail-silent failure detection (e. g., regular lifesign ("heartbeat") messages), if dependability is to be achieved.

2.2 Integrated vs. Federated Architecture

A distributed application can be classified with respect to the overall system architecture. Although most real systems are hybrid systems, the two extremes, integrated and federated architectures, are briefly discussed in this section.

A federated *architecture is characterized in that every major function of an embedded system is allocated to a dedicated hardware unit.* [Kop04a, p. 160] This results in a vast number of independent hardware subsystems, each performing a certain, specialised function, with minimal or even no interaction between the different subsystems. Every single subsystem is treated as a "stand-alone" system, therefore any maintenance or upgrade actions, although less complex, have to be performed separately for every system. In spite of these and other deficiencies, several advantages of federated architecures can be identified, the major ones are described in [OPHS06, p. 84-85] and briefly listed below.

Fault containment A hardware fault in a core or a disturbance of the communication medium of a federated architecture only affects a single application subsystem, the other application subsystems are not concerned, whereas a hardware fault occurring in a core or communication medium of an integrated architecture may influence multiple different application subsystems.

Error containment Error distribution among different subsystems is prevented (or at least reduced) by the nature of federated architectures, since interaction between different subsystems is limited.

Independent development The independence of the different subsystems allows for their independent development, therefore the subsystems can be manufactured by different vendors or design teams and easily combined to form the whole application system.

Complexity control With no need to adapt a subsystem to operate using the same resources as another subsystem, as inevitable in an integrated architecture, a federated architecture helps to keep the overall system complexity low. Any side effects or dependencies between the different sub-

systems, which significantly increase the cognitive complexity of a system, are prevented.

We call an architecture integrated *if a single core can support a number of partitions that can host different functions and a physical wire can host many different virtual encapsulated communication channels of known temporal characteristics.* [Kop04a, p. 160] In an integrated architecture, different functions, located in their own, separated subsystems in a federated architecture, are merged into a single system, sharing common resources. Some advantages of integrated architectures are concisely listed below, taken from [OPHS06, p. 85].

Hardware cost reduction Sharing resources between different application subsystem can significantly reduce the number of required hardware units (e. g., processors, sensors, communication media, etc.), resulting in reduced overall system hardware cost.

Dependability improvements due to reductions of wiring and connectors Since a large amount of electrical failures in embedded systems is due to connector or wiring problems, dependability can be improved by an integrated system, which shares communication media and therefore uses less physical wires in contrast to a federated system.

Fault-tolerance In integrated as well as in federated architectures, fault tolerance is achieved by using replicated hardware. The benefit of an integrated architecture is that these replicated resources are available among the different subsystems, while every subsystem in a federated architecture has to provide its own replicated hardware. Failure of a subsystem in a federated architecture may occur in spite of the presence of a sufficient number of free overall system resources able to tolerate the occurred fault, since a subsystem cannot access the spare resources of other subsystems.

Improved coordination of application subsystems Tight coupling of different subsystems, which is needed by a growing number of complex embedded application systems, is supported by integrated architectures, the coordination of inter-subsystem communication is simplified in contrast to federated architectures.

Both, integrated and federated architectures, offer advantages for future embedded application systems. The ideal future embedded application system *would combine the complexity management advantages of the federated approach, but would also realize the functional integration and hardware efficiency benefits of an integrated system.* [Ham03, p. 32] To achieve this goal, logical separation and strong encapsulation of different Distributed Application Subsystems (DASs) in an integrated system has to be provided by the underlying system architecture. Although parts of different Distributed Application

Subsystems may share the same resources, e. g., the same microchip, the same physical communication medium, etc., the system architecture has to prevent any side effects or unwanted dependencies between the different DASs and has to support the possibility of an independent development of the different Distributed Application Subsystems.

2.3 DECOS

The Dependable Embedded Components and Systems (DECOS) project [Dep] is part of the Sixth Framework Programme (FP6) for Research, Technological Development and Demonstration (RTD) of the European Community, which started in 2002. *DECOS methodically targets, investigates, and develops approaches to significantly alleviate ... the identified five key obstacles ... to the deployment of advanced electronic functions in embedded systems.* [CS04, p. 1] Those key obstacles are (taken from [CS04, p. 1]):

- Electronic Hardware Cost
- Diagnosis and Maintenance
- Dependability
- Development Cost
- Intellectual Property (IP) Protection

The DECOS project intends to provide an integrated distributed architecture, consisting of pre-validated hardware components, which allows to execute certified or custom software modules while providing strict encapsulation of different subsystems to ensure their independent development, seamless integration, and operation without any unwanted side effects.

The DECOS architecture offers a framework for the development of distributed embedded real-time systems integrating multiple Distributed Application Subsystems (DASs) with different levels of criticality and different requirements concerning the underlying platform. It *is based on a time-triggered core architecture and a set of high-level services that support the execution of newly developed and legacy applications across standardized technology-invariant interfaces.* [POT+05, p. 3]

In an ebedded system using the DECOS architecture, the overall application system functionality is divided into Distributed Application Subsystems (DASs), each of which should provide a meaningful part of the service of the overall system (e. g., a brake-by-wire DAS could be responsible for all matters concerning the brake service of a car). Each DAS can be further devided into

jobs. A job *is the basic unit of work that employs the communication system for exchanging information with other jobs, thus working towards a collective goal.* [POT+05, p. 4] Figure 2.4 displays a schematic of the DECOS architecture.

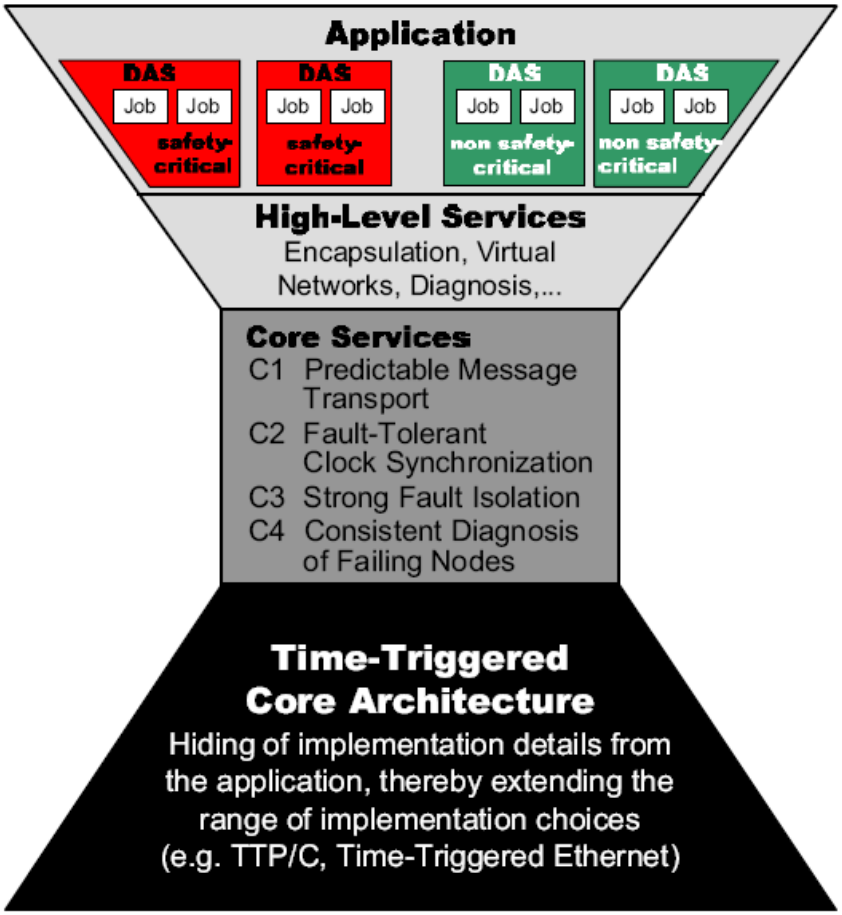

Figure 2.4: The DECOS Integrated System Architecture [POT+05, p. 4]

A prototype implementation of a DECOS cluster was built at the Technische Universität Wien, Institut für Technische Informatik, and described in [OPHS06, sect. 5].

During the course of the DECOS project, the demand for a System-on-a-Chip (SoC) fulfilling the requirements of the DECOS architecture, namely strict encapsulation of DASs and a dependable, deterministic communication system, arose. As a consequence, the development of the Time-Triggered System-on-a-Chip (TTSoC) architecture (chapter 3) was initiated.

3 Time-Triggered System-on-a-Chip

3.1 Requirements

In contrast to other System-on-a-Chip (SoC) development efforts aiming at improvement of overall system performance by designing more sophisticated, close-coupled processor cores with strong mutual dependencies, like the Cell Processor [PAB+06], the development of the TTSoC architecture described herein was focused on the overall architecture of the SoC and the used on-chip communication system, namely the Network-on-a-Chip (NoC). The overall goal of the TTSoC project was to design an SoC architecture which provides strict encapsulation of all host processors and dependable and predictable communication among them to support safety-critical Distributed Application Subsystems (DASs).

As a result, the TTSoC architecture allows the use of arbitrary application specific host processors, as long as a proper interface to the NoC can be established.

The main requirements of the Network-on-a-Chip, identified during the development of the Time-Triggered System-on-a-Chip architecture, are briefly described in the following list.

Dependability In order for a safety-critical DAS to function within the given parameters, the underlying communication network has to fulfill the same or even higher dependability demands. The core components of the TTSoC architecture are designed to be certified to the highest criticality level required by a given application.

Encapsulation A single TTSoC may not run safety-critical applications on all of its cores, but a mixture of safety-critical and standard applications, e. g., parts of a brake-by-wire DAS and a multimedia DAS. To prevent disturbance of the critical DASs caused by faulty or mischievous cores, the NoC has to ensure strict encapsulation of all DASs. Any communication between different DASs has to be regulated, blocking of the communication medium, corruption of communication data, message spoofing,

or similar illegitimate actions by any core have to be suppressed. Even an arbitrary (i. e., Byzantine) failure of a non safety-critical host is not allowed to hamper the service of the other cores of the TTSoC.

Controlled Complexity Large distributed embedded systems can become fairly complex, which makes them hard to design and maintain. By encapsulating the application subsystems, the TTSoC architecture allows a system designer to focus on a specific subsystem without the need to know all aspects of the overall system architecture. Every host processor connects to the communication system through a standardized interface, the exact location inside the network (i. e., on which core the software is running) is of no concern during the design process of the host software. Even a migration to another core after finishing the application design or during the system's operational life due to maintenance issues is possible in a well designed application system.

This location-independent behavior is not restricted to a single TTSoC, a cluster of multiple TTSoCs, connected through gateways, can be built to allow a physical separation of different jobs in a DAS. Being able to migrate a job to another core without the need to adapt the host software allows for every DAS to be designed for its own, knowledge of other DASs sharing the same TTSoC or TTSoC cluster is not necessary at design time.

Real-Time Many distributed embedded systems require real-time behavior. To support such systems, the TTSoC architecture has to provide a distributed global real-time service.

Replica Determinism In highly dependable systems, such as drive-by-wire systems, the use of Triple Modular Redundancy (TMR) is very common. For an efficient use of TMR, an application has to show replica deterministic behavior [Pol94]. As a consequence, all components of the implemented NoC have to be replica deterministic too.

Resource Efficiency The use of on-chip resources has to be limited for an embedded system to be competitive on the market. The two crucial resources of an embedded SoC are **chip area** and **power**. While the chip area used by the SoC is determined by the design of all of the implemented SoC components and remains constant during the system's life cycle, power consumption can be regulated "on-line" during operation.

The TTSoC architecture design aims at reducing the used chip area by building all components as compact as possible. Power is considered a dynamic resource, the TTSoC architecture is intended to manage the available on-chip power by disabling the power supply of cores not needed at the moment, reducing a host processor's internal clock frequency, and/or

setting low power consumption host modes in low-duty situations. Besides saving power on mobile, battery supplied embedded systems (e. g., in a car), the intention is to secure an uninterrupted power supply for cores running safety-critical applications.

Reusability Any distributed communication system should provide a uniform interface to the attached host processors to simplify access to all communication functions. The NoC fulfills these requirement by introducing a standardized Host Interface providing independence of application software from the communication network. Additional supporting functions (e. g., interrupt, global real-time, and watchdog services) are provided to enhance the functional range of the NoC.

Extendability To bridge the gap between resource efficiency and usability, the TTSoC architecture is designed to be extended by adding additional, special purpose functions. Such functions widen the functional range of specific cores according to application demands, with the disadvantage of increased resource usage.

Diagnosis & Maintainability Embedded systems are, by nature, built into larger technical systems (e. g., cars), which impose various difficulties concerning maintenance. Hence, the possibility to easily diagnose failures and to "repair" a system by simply exchanging a failed component like a whole TTSoC with a new one, is a crucial requirement.

Bandwidth The NoC has to provide enough communication bandwidth to allow unhindered operation of all attached cores. Since the required bandwidth is highly application dependent, no minimum bandwidth is specified. Although intentions are to maximize the available bandwidth, the speed of the NoC (e. g., on-chip clock frequency) is not a driving design factor, the other design requirements are prioritized.

3.2 Overview

The Time-Triggered System-on-a-Chip (TTSoC) architecture is developed to house multiple different cores, each consisting of a host processors with peripheral devices (i. e., local I/O controllers like Controller Area Network (CAN) controllers), a Communication Network Interface (CNI) Layer, and a Trusted Interface Subsystem (TISS) on the same microchip and to connect them through a dependable, time-triggered communication system, the Network-on-a-Chip (NoC). The hosts have no other possibility to influence one another but the NoC, guaranteeing encapsulation of different DASs. The TTSoC architecture is designed for safety critical applications, thus some components of the NoC

are part of a "trusted region" within the TTSoC architecture. These components have to be certified to the highest criticality level required by any core of a TTSoC, because failures of those core components would cause failures of the entire TTSoC.

The design of the TTSoC architecture and of its components was motivated by [OPHS06] and some early specifications and concepts were introduced in [Obe06] and [KHO+06], some of which are used in the implementation described in this thesis.

Figure 3.1 shows an overview of the Time-Triggered System-on-a-Chip architecture.

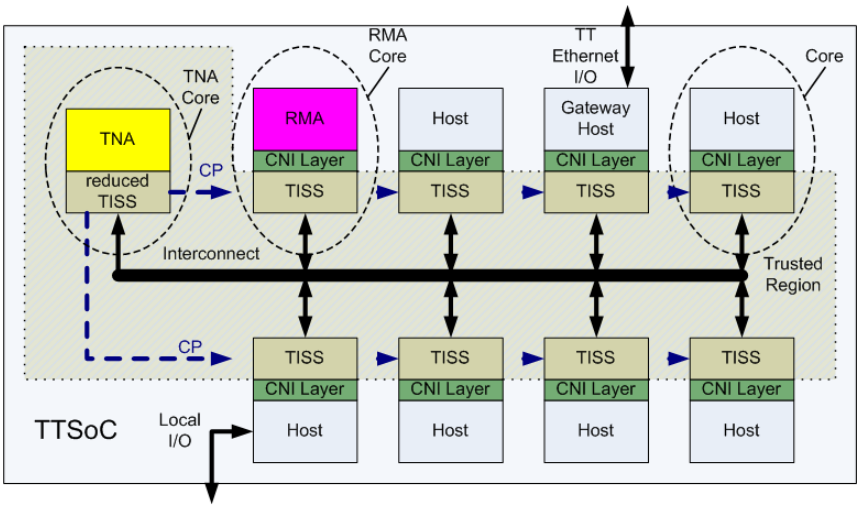

Figure 3.1: Overview of the TTSoC architecture

A host processor reads and writes all communication data as well as additional status and configuration data, by means of a memory mapped interface provided by the attached Communication Network Interface (CNI) Layer. Since the NoC is a shared resource, a Trusted Interface Subsystem (TISS) is used to connect the CNI Layer to the underlying NoC Interconnect to protect the communication medium from a host violating its temporal specification. A single TISS consists of the Medium Access Control (MAC) Layer, responsible for protecting the communication medium and initiating data transfers according to the communication schedule, and the Logical Link Control (LLC) Layer, which handles all host related communication issues (e. g., presenting the communication data in a convenient way, synchronization, etc.). A host and its attached CNI Layer and TISS is called a core of the TTSoC.

Two special cores which are part of the NoC can be identified: The Trusted Network Authority (TNA) and the Resource Management Authority (RMA).

The RMA, a standard core, manages the available resources of the TTSoC, especially by building a conflict-free communication schedule, while the TNA, as part of the trusted region of the TTSoC, acts as kind of a guard for the NoC. The TNA core uses a special host processor to verify the communication schedule and the resource allocation plan submitted by the RMA and implements an additional configuration planning (CP) interface to forward the valid configuration to all other TISSs.

The host processors can be standard commercial microprocessors, dedicated Intellectual Property (IP) modules, gateways to other communication networks like Time-Triggered Ethernet [SK06] or TTP/C [TTTa], or any custom built hardware module.

The following sections describe some of the crucial concepts associated with the TTSoC architecture development (section 3.3), the general operation of the TTSoC (section 3.4), and the components that form the NoC (section 3.5).

3.3 Concepts

3.3.1 Time

The TTSoC architecture provides a synchronized global real-time service for all cores connected to the NoC. The global real-time is not only used to establish the time-triggered communication but can also be accessed by the cores to support real-time operation of installed application software (e. g., job scheduling, etc.).

The time format used by the TTSoC architecture is derived from the Time-Triggered Ethernet time format [Ste06, p. 50] and closely related to the Global Positioning System (GPS) time format. The binary TTSoC architecture time format is 8 bytes (64 bit) wide and is based on the physical second. It inherits the epoch of the GPS (January 6, 1980 at 00:00), unless no external synchronization is possible. In that case, the epoch starts with the system startup instant.

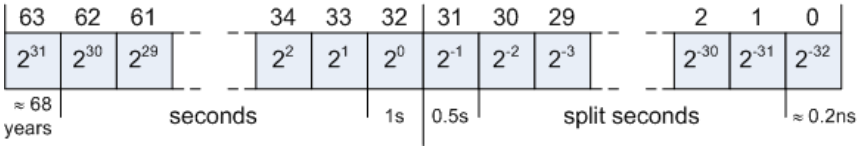

Figure 3.2: TTSoC Architecture Time Format

The 8 bytes of the TTSoC architecture time format (figure 3.2) can be splitted into two four byte segments: the higher 32 bit represent the elapsed number of seconds since the start of the epoch while the lower 32 bit denote the fractions of a second in negative powers of two (i. e., 0.5s, 0.25s, 0.125s, etc.). Thus a time horizon of 2^{32} seconds (i. e., approximately 136 years) and a granularity of 2^{-32} seconds (i. e., approximately 0.2 ns) is established, providing the possibility to uniquely represent every time instant since January 6, 1980 with a precision of under a nanosecond. The small granularity became necessary because a granularity of 2^{-24} seconds (i. e., approximately 60 ns), as introduced by the Time-Triggered Ethernet time format, would restrict the time-triggered Network-on-a-Chip to an on-chip clock frequency of about 16 MHz. Instead of widening the time format to more than 8 bytes, it was decided to reduce the time horizon to support the needed granularity. Table 3.1 compares the Time-Triggered Ethernet and the TTSoC architecture time format with respect to their main characteristics.

	TT Ethernet	TTSoC
Overall Size	8 bytes	
$\geq/<$ 1 second	5/3 bytes	4/4 bytes
Epoch	January 6, 1980, 00:00 / system startup	
Horizon	approx. 30 000 years	approx. 136 years
Granularity	approx. 60 ns	approx. 0.2 ns

Table 3.1: Time Format Comparison: TT Ethernet vs. TTSoC

Due to implementation hardware restrictions[1], the lowest six bits of the TTSoC architecture time format are set to zero throughout the TTSoC implementation described in this thesis, reducing the resource effort for the internal time representation from 64 to 58 bits. External time representation (i. e., the global real-time accessible by the hosts) is not affected. A detailed description of the implemented time format can be found in [Eng07].

3.3.2 Messages

Communication among the cores in the time-triggered NoC is organized in periodic messages. The messages consist of a constant number of fragments which transport 128 bit data each (figure 3.3).[2] The message size ranges from

[1] The granularity of the TTSoC architecture time format of 0.2 ns would require an on-chip clock frequency of at least 5 GHz to be accurate.

[2] Unless mentioned otherwise, a "fragment", as used in this document, refers to a 128 bit wide part of a message.

0 (1 fragment) to 255 (256 fragments) and is set by the TNA taking the host requests (see section 3.5.5) into account. Message transport is performed by means of periodic Pulsed Data Streams (3.3.3) scheduled by the TNA. Sources and sinks of messages are represented by ports (section 3.3.4) at the sending core and the receiving core(s), the transmission is initiated by the MAC Layer according to the schedule.

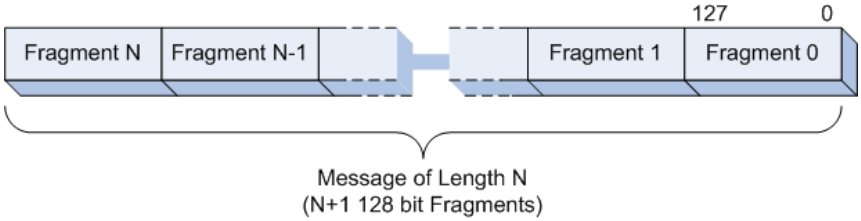

Figure 3.3: NoC Message

Three types of messages are supported by the NoC:

State Messages are the intended main communication form of the TTSoC architecture. State Messages contain state information, therefore they are suited best for transportation over a time-triggered network. Because of the at-least-once semantics required by state information, the state information at the receiving core can be updated in place without forcing the receiving core to consume the previous message in advance. A state message is transmitted periodically regardless of any change of the contained state information. An update of this information by the sending core's host is automatically transmitted to the receiving core at the next scheduled message period and can be used by the receiving core's host with a maximum (worst-case) delay of one message period duration[3].

Event Messages are sent after the occurrence of an event. Since the NoC incorporates a time-triggered communication service, these messages have to be scheduled like any other message to reserve the needed amount of bandwidth. Actual sending of event messages can be suppressed until an event occurred, but queues are needed at the sending and receiving event message ports to ensure the required exactly-once sematics. This mechanism helps to save power since no sending and receiving operations have to be performed until there is actual data to be transmitted, and guarantees delivery of event information even under peak-load scenarios, as long as the periods of the event messages in the predefined message

[3]Disregarding the constant sending and receiving operation latencies for every fragment.

schedule are shorter than the minimum time interval between the associated events. A higher event occurrence frequency can be tolerated for a short amount of time, as long as the sender and receiver queues are large enough to store all generated event messages until they can be transmitted over the NoC or consumed by the receiving host. In addition, a longer event occurrence pause is required from time to time to allow the NoC to empty the send queue again.

Streaming Messages are used to transport streaming data (e. g., an audio stream). In contrast to state or event messages, streaming messages are not used to describe the state or the change of state of an entity, but to transport big amounts of information (i. e., a serialized data stream) in a convenient way. Because of the nature of streaming data which requires a constant feed of input by the sender and immediate processing at the receiver, the communicating hosts are forced to update (sender) or read (receiver) the associated Streaming Port before / after the transmission of every single fragment, respectively.

Due to the bus topology of the core interconnection used by the TTSoC implementation described in this document, all messages are broadcasted to all cores. Point-to-point or multicast communication is incorporated transparently by the different medium access schedules of the cores, which allow only the designated receiving cores to read data from the bus while a certain message is presented by a sending core. Future implementations may support true point-to-point communication by using a switched network or other forms of core interconnection. See sections 3.5.3 and 6 for details on the current TTSoC interconnection and possible future improvements.

3.3.3 Pulsed Data Streams

The concept of pulsed data streams was proposed in [Kop06]. The basic requirements for pulsed data streams are an a priori known set of communication participants, a synchronized global notion of time, and a cyclic communication schedule. *A pulsed data stream is a cyclic data stream that transports data uni-directionally in pulses from one sender to* n *a priori identified receivers at a specified phase of the cycle for a specified duration.* [Kop06, p. 4].

In the TTSoC architecture, the cyclic communication schedule is established by the TNA among the cores of the NoC. All messages distributed over the NoC are periodic and the cores are aware of a common global real-time, hence the pulsed data stream concept can easily be applied.

Furthermore, since all TTSoC architecture messages consist of fragments, the NoC implements a special pulsed data stream version. In addition to its sender

and receiver identification, each message can be described by its period (i. e., the cycle duration), the message pulse start instant (i. e., the phase offset), and the message pulse duration. To prevent blocking of the communication medium caused by long messages, each message pulse is subdivided into single fragment pulses. The fragment pulses are characterized by the fragment period and the fragment pulse start instant inside a message pulse, the fragment pulse duration is constant due to the constant transmission delay of the fixed length fragments. Figure 3.4 shows the pulsed data streams used for message transmission in the NoC.

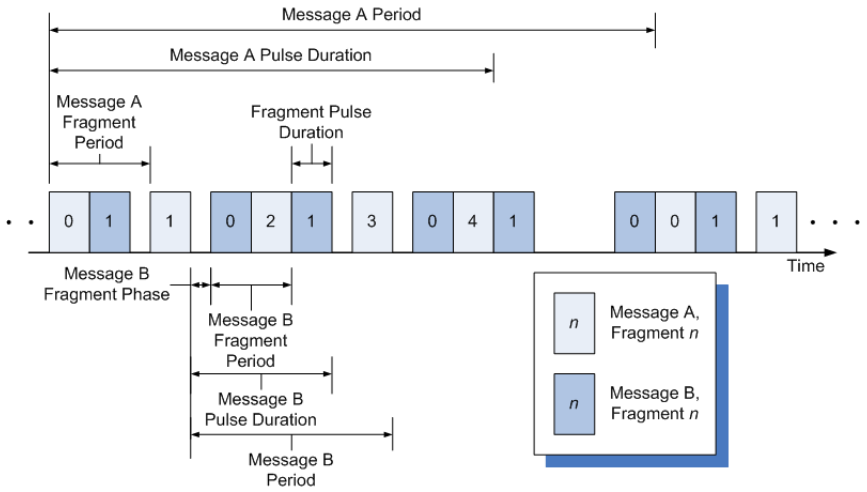

Figure 3.4: NoC Pulsed Data Streams

The advantage of this implementation is that the fragments of different messages (with possibly different periods) can be interleaved on the communication medium. Otherwise two messages, one message with a long pulse duration and another message with a message period shorter than the first message's pulse duration could not be scheduled, either the long message would never gain possession of the communication medium or the short-period message would miss its reception deadlines.

The message schedule provided by the TNA aims to fulfill all message requests without violating any message periods or deadlines. If no valid schedule can be found, non safety-critical messages are declined to ensure proper transmission of the safety-critical data among a priori specified high-reliability DASs.

3.3.4 Ports

The NoC ports form the end points (one source and one or more sinks) of the communication line between different cores. They were designed to store variable size messages (fragment granularity) ready to be sent at the next message period (output ports), or previously received messages waiting to be processed by a host processor (input ports).

Every port can be identified locally (inside a core) by its unique PortID which is set and associated with a specific message by the TNA. Every message needs a single output port in its sending core (the source of the message) and one input port at every receiving core (the sink(s)).

Three input and output port types matching the three message types described in section 3.3.2 are featured by the TTSoC architecture.[4] All port types provide a different kind of synchronization mechanism to prevent data corruption due to simultaneous read and/or write accesses. Further details of the implemented port types can be found in section 4.2.2.2.1.

State Ports store a whole state message. While input State Ports use an adapted form of the non-blocking write (NBW) protocol [KR93] to ensure proper reception of all state messages, output State Ports are designed to store a second instance of the state message to be sent in a so called "shadow register". Using this mechanism, the host software is allowed to write the next state message to the "shadow register" while transmission of the current state message, stored in the "standard register", is still in progress, or vice versa.[5]

Event Ports provide variable length queues to store a sequence of complete event messages. Input and output Event Ports use the same synchronization mechanism, namely two counters which hold the current read and write positions in the event message queue. The host acts as the *writer* (output) or the *reader* (input) of a certain Event Port, the TISS performs the respective other role. Receiving or sending an event message and simultaneously writing or reading another one to or from the same Event Port is possible at all times, on condition that the queue is not full or empty, respectively.

[4] The NoC has no possibility to determine if the message type matches the configured port types of the associated input and output ports. Unless an application is intentionally designed to handle a certain type of messages by using a port of a different type (e. g., a diagnostic entity logging all kinds of message transfers by means of input Streaming Ports), it is strongly recommended to align the port types in the port configuration with the message schedule to prevent system errors due to misinterpreted data.

[5] The "standard" and "shadow registers" are no actual registers but consecutive memory areas in the CNI memory.

Streaming Ports do not store whole messages but only a single fragment of a streaming message. To maintain a continuous data stream between two cores, the sending host has to update the fragment data before every fragment send instant and the receiving host has to fetch this data before the next fragment arrives over the NoC, otherwise a fragment is sent more frequently than once (sender) or a fragment is overwritten by the following one (receiver). Streaming Port access synchronization is performed with the same mechanisms that State Ports use, synchronization of the data stream (e. g., start, stop, pause, etc.) has to be maintained by the application itself (e. g., by using additional event messages).

3.4 Operation

Sending and receiving of messages in the TTSoC architecture is triggered by the progression of time. Every message is associated with a time period and a phase offset at which transmission of this message has to start. Since a message consists of fragments, every message is additionally characterized by a fragment period and a fragment phase offset. These periods and phase offsets are stored in the message schedule of every TISS, allowing the MAC Layer to determine the exact send or receive instant for every fragment.

Figure 3.5 shows the transmission of a message from one core to another, the following paragraphs describe this operation in detail.

The MAC Layer requests the data to be sent from the LLC Layer prior to the calculated send instant by driving the PortID and the fragment number of the fragment to be sent on the appropriate lines (a).

The LLC Layer calculates the address of the requested fragment in the CNI memory using the port configuration data written by the host and the current state of the port synchronization information (b). After that, it fetches the fragment from the port stored in the CNI Layer and forwards the data to the MAC Layer (c).

The MAC Layer sends the fragment over the network precisely at the send instant determined by the message schedule (d).

At the receiving core, the message schedule noted the same time instant as receive instant, so the receiving core's MAC Layer stores the fragment data presented on the network and forwards it to the local LLC Layer together with the local PortID and the fragment number (e).

The LLC Layer calculates the local CNI memory address based on the information obtained from the port configuration memory and the synchronization

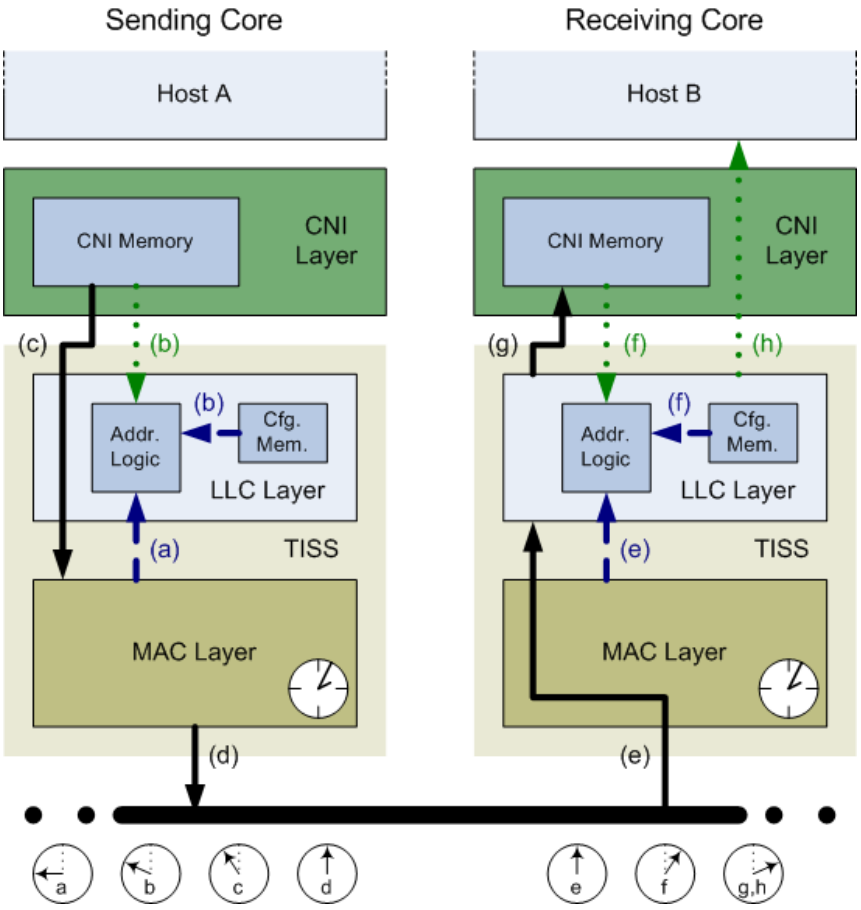

Figure 3.5: TTSoC Operation

flags (f) and finally stores the fragment in the port in the CNI Layer (g).

After completing the reception or sending of a whole message, a "New Message" or "Ready" interrupt can be generated to inform the host that the receiving port contains valid data or the sending port is ready for the next message to be written by the host, respectively (h).

3.4.1 Configuration

The message schedule is generated by the RMA according to message requests sent by the hosts, and verified and distributed to the TISSs by the TNA. A reduced message set can be scheduled due to short resources, a failure, or after system startup.

The layout of the message request mechanism is application specific, but the

intended standard configuration operation between the RMA, TNA, and the TISSs is shown in figure 3.6 and described below.

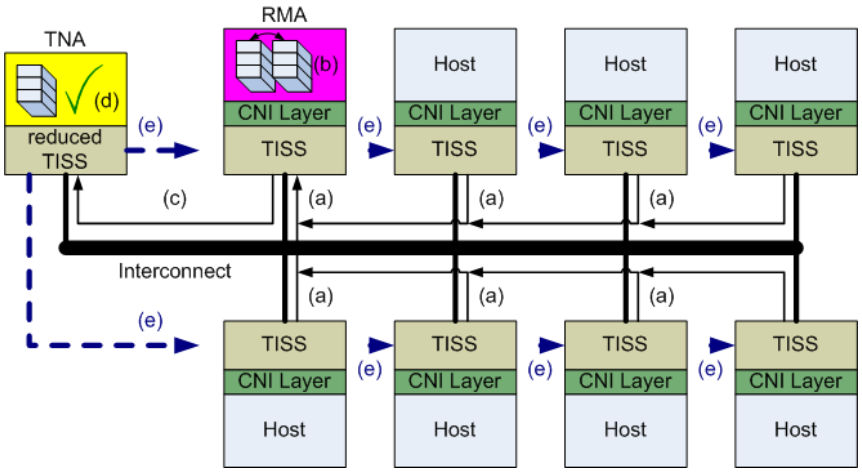

Figure 3.6: TTSoC Message Configuration Procedure

After system startup, an a priori defined, fixed message schedule is active, allowing the cores to request their needed messages by sending message request messages to the RMA (a). Depending on the application requirements, additional diagnostic messages and/or messages used by services that have to start operation immediately after startup can be scheduled too.

The RMA tries to build a feasible message schedule including all requested messages (b) and transfers its solutions to the TNA (c).

The TNA verifies the validity of the received message schedule (d) and, if it is confirmed, sends the according core-local message schedules in combination with a common reconfiguration time period and phase offset to the TISSs (e). When the next reconfiguration time instant is reached, all cores switch to the new configuration simultaneously and normal network operation commences.

Reconfiguration of the TTSoC is a periodic process. If no new messages (and no termination of active messages) are requested by the cores, the previous configuration is kept. A change of the current message schedule can become necessary due to changed application service requirements (e. g., a user requested an additional service by turning a switch) or a core failure that has to be compensated. A change of the message schedule is possible at every periodic reconfiguration instant.

For safety-critical applications, it can be useful to define a fixed backup message schedule during the system design process and store it in the TNA to

be activated in case of a possible RMA failure. Although all flexibility gained by the dynamic resource allocation process is lost in such a case, essential safety critical services can be kept active until a fail-safe state is reached or the occurred failure can be corrected by maintenance, by starting a backup system, or by other means.

The described configuration mechanism can be used to allocate system resources other than communication medium access in future TTSoC implementations, the current TNA implementation supports only message scheduling.

3.5 Components

3.5.1 Trusted Interface Subsystem

The Trusted Interface Subsystem (TISS) (figure 3.7) consists of two parts: the Medium Access Control (MAC) Layer (section 3.5.1.1) and the Logical Link Control (LLC) Layer (section 3.5.1.2).

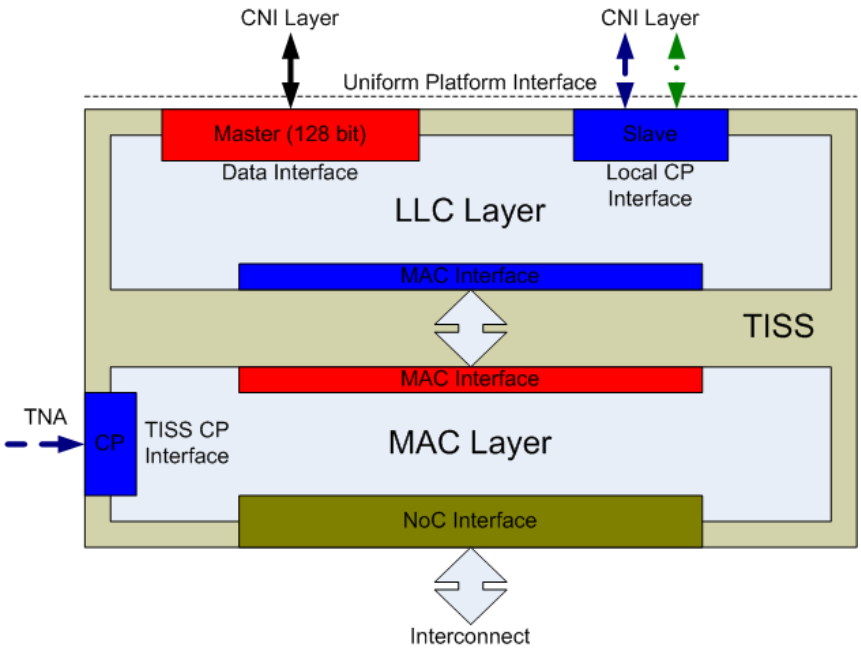

Figure 3.7: Trusted Interface Subsystem (TISS) Overview

The purpose of the TISS is to send and receive messages to or from other cores according to the schedule determined by the TNA and fetch or store these

messages from/to the CNI Layer. Since the TISS is responsible for communication medium access and provides the different types of ports and their synchronization mechanisms, the TISS is part of the trusted region of the TTSoC architecture. Therefore it is intended to be certified to the highest criticality level of any host within the TTSoC.

3.5.1.1 MAC Layer

The MAC Layer, described in [Eng07], is responsible to protect the communication medium from undesired disturbance. It stores the message schedule determined by the TNA and uses it not only to restrict medium access, but to initiate sending and receiving operations performed by the LLC Layer.

The communication schedule is transfered directly from the TNA to the MAC Layer through the TISS configuration planning (CP) interface, the configuration data needed by the LLC Layer is forwarded upon reconfiguration. In addition, the MAC Layer keeps the global real-time accurate, which is continuously transferred to the LLC Layer through the MAC interface.

Although the host is able to read the MAC Layer configuration, there is no possibility of a host write access to the MAC Layer, hence any disturbance of the time-triggered communication by an erroneous host is avoided. Even the LLC Layer cannot influence the operation of the MAC Layer, thus the whole core is in the sphere of control of the MAC Layer, concerning communication medium access.

3.5.1.2 LLC Layer

The LLC Layer allows a host processor to access the communication service presented by the NoC in a convenient way. While the MAC Layer regulates reception of single fragments at specific time instants and signals these events to the LLC Layer, the LLC Layer is responsible to assemble these fragments to form messages of configurable length, to store these messages in the designated ports of certain types, and to provide the necessary synchronization mechanisms to allow a consistent data transfer to the host. Transmission of fragments is handled similar: Before the occurrence of a send time instant, the MAC Layer requests the fragment to be sent from the LLC Layer which fetches it from the memory location determined by the port identification, fragment position, port configuration (base address, type), and current port synchronization status (e. g., queue positions).

To maintain this highly flexible (concerning port size, port type, queue length, etc.) behavior, the provided ports can be configured by the host according to the application requirements.

Apart from the communication service, the LLC Layer houses two additional trusted services: a watchdog service and an error status register to detect and monitor possible core failures.

The LLC Layer and the CNI Layer (section 3.5.2) were both implemented in the course of this thesis, chapter 4 offers a closer view on their operation and implementation.

3.5.2 CNI Layer

Memory space to store all port data and the related synchronization data is provided by the Communication Network Interface (CNI) Layer. It therefore acts as a temporal firewall [KO02] for the NoC. The CNI Layer works closely with the LLC Layer, concerning especially the timeliness of read and write accesses to the CNI memory.

The standard CNI Layer implements a memory-mapped host interface to allow the host software to access the port data memory, the port configuration memory, the MAC Layer configuration memory, possible CNI Layer extensions, and the LLC Layer and CNI Layer register files. Furthermore, an interrupt service is located in the CNI Layer, keeping track of any interrupt triggered by the TISS, the CNI Layer, or any Middleware Plug-In.

The CNI Layer was designed to allow easy modification to suit application needs, either by adding CNI Layer extensions (described in the following paragraph) or by replacing the whole standard CNI Layer with a host-specific implementation, therefore the CNI Layer is located outside of the trusted region of the TTSoC architecture. Certification of the CNI Layer is though possible, if required by the application due to a host or a CNI Layer extension of high criticality.

The CNI Layer is the subject of chapter 4, together with the LLC Layer.

3.5.2.1 CNI Layer Extensions

During the development of the TTSoC architecture, possibilities to further improve the functional range of the NoC were evaluated. To keep possible certification procedures simple and to support the independent development of additional functions, it was decided to enhance the CNI Layer using optional CNI Layer extensions. Different kinds of CNI Layer extensions were discussed, some of which were implemented later.

Chapter 5 deals with the possible forms of CNI Layer extensions.

3.5.3 Interconnect

The Network-on-a-Chip Interconnect has to perform three main tasks, all of which are critical for TTSoC operation:

- Distribution of periodic and sporadic messages between the cores of the NoC
- Maintaining a consistent time base among the NoC cores
- Transporting the message schedule and additional configuration parameters from the TNA to the TISS configuration planning interfaces

To guarantee operation in a safety-critical environment, the NoC Interconnect has to be certified to the highest criticality level demanded by the respective application.

The message distribution service can be implemented using different network architecures, e. g., bus, star, mesh, or switched topology (figure 3.8). In addition, the Interconnect can be replicated to increase transmission bandwidth or network safety by using a second communication channel.

The chip-wide global real-time is accessible by all the TISSs in the NoC and has to be kept consistent among all cores without jitter and with minimal clock skew.

The configuration planning Interconnect can be slow, compared to the message distribution service, since configuration intervals are relatively large, compared to fragment sending periods. The need for the additional configuration planning Interconnect arises due to the requirement to guarantee that only the TNA can write the TISS message schedule and low-level configuration information (i. e., host mode, watchdog period).

The current TTSoC implementation uses a 128 bit wide bus network for message transportation, a simple memory interface for configuration planning purposes and a centralized time distribution system. Details can be found in [Eng07].

3.5.4 Resource Management Authority

The Resource Management Authority (RMA), although part of the TTSoC architecture, is implemented as a dedicated standard core. Since performing dynamic management of all available TTSoC resources (e. g., communication, computational, power resources) is a fairly complex task and can be highly application specific, the RMA is not assumed to be free of design faults and therefore is not certified.

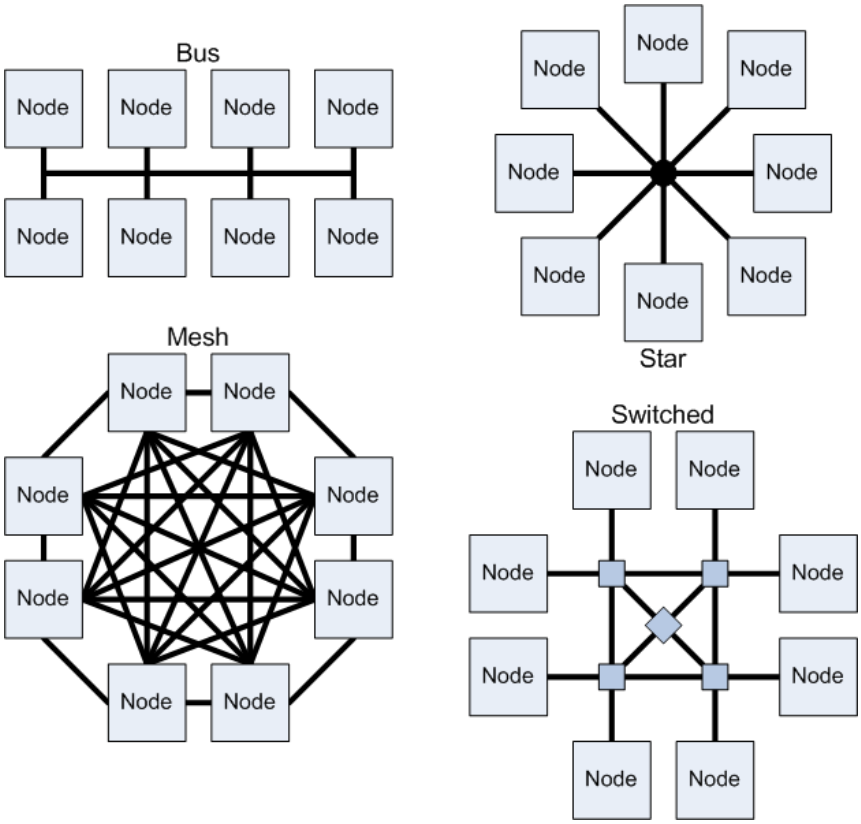

Figure 3.8: Different Interconnect Topologies

The RMA receives message (and/or resource) requests from the cores of the NoC, containing, among other things, the requested message length and message period. It tries to fulfill all message requests by composing a conflict-free schedule for fragment sending and receiving time slots across the NoC. In addition, host mode configurations for all cores have to be evaluated according to the current needs of the application and the available computational resources. The RMA tries to prepare a feasible message schedule and resource allocation plan in a predefined amount of time and transfers these plans to the TNA by means of the standard communication service. In the TNA the message schedule and the resource allocation plan are verified and, if necessary, altered to guarantee unhampered safety-critical operation.

By introducing this two-stage configuration mechanism (the RMA builds a provisional NoC configuration, the TNA verifies this configuration according to safety-critical requirements), the application system designers gain the freedom to implement arbitrary scheduling algorithms best suited for a specific appli-

cation while the TTSoC architecture ensures validity and dependability of the determined configuration under all circumstances.

3.5.5 Trusted Network Authority

The Trusted Network Authority (TNA) is responsible for verifying and distributing a valid message schedule and resource allocation plan for the whole TTSoC. Therefore, the TNA protects all safety-critical resources of the TTSoC. The complexity of the TNA is kept as low as possible to avoid design and operational faults and to ease certification, the TNA is considered to be part of the trusted region of the TTSoC architecture.

After receiving the proposed message schedule and resource allocation plan from the RMA, the TNA verifies them. Should the message schedule or the resource allocation plan be invalid, for instance due to an erroneous RMA or a change in the availability of certain resources, the TNA alters the plan to ensure that certain application specific safety-critical DASs are able to maintain their operation while non-critical application services can be deteriorated. During the application system design process, designers should ensure that this situation is a rare event, only caused by core failures or due to rare peak-load scenarios.

Similar to network medium access, other resources can be scheduled by the RMA and protected by the TNA in future implementations.[6] Examples are computational resources (e. g., host processor time for a specific job, memory consumption), controlled by setting an appropriate host mode, or power consumption of the cores. By controlling allocation of these resources and the communication medium, the TNA gains the ability to ensure uninterrupted and correct operation of safety critical DASs in spite of failures or unforseen overload scenarios.

The verified message schedule and host mode configurations are distributed to the TISSs via the configuration planning Interconnect. Every single TISS receives only its own relevant part of the message schedule and its specific host mode and watchdog period configuration.

[6] The current TTSoC implementation is focused on protection of the communication medium, power control or complex host mode settings are considered but not implemented by now.

4 LLC and CNI Layer

4.1 Overview

As part of the Trusted Interface Subsystem (TISS), the purpose of the Logical Link Control (LLC) Layer is to function as the link between the Medium Access Control (MAC) Layer and the Communication Network Interface (CNI) Layer. The CNI Layer is directly connected to the host processor and stores the messages transmitted via the NoC.

The following ports, corresponding to the messages described in section 3.3.2, are provided:

State Ports of variable size (fragment size granularity) which hold a whole pulsed data stream

Event Ports of variable size (fragment size granularity) which hold a whole pulsed data stream and provide a queue of variable length

Streaming Ports which hold a single fragment of a pulsed data stream

In addition, the LLC and CNI Layers perform some configuration and administrative functions, e. g., interrupts, status registers, etc. Both, the LLC Layer and the CNI Layer work together to make the whole variety of NoC services accessible by the host, therefore they are described together in this chapter.

Different components are needed to fulfill these tasks. An overview of the LLC and CNI Layers is depicted in figure 4.1, which shows these components.

The following sections describe the various interfaces used by the LLC and CNI Layers (section 4.2) and the implementation details of the LLC Layer and CNI Layer hardware (section 4.3). Finally, section 4.2.2.2 deals with the details of the memory mapped host interface which is used by the host to access all functions of the NoC.

To extend the functional range of the CNI Layer, so-called Plug-Ins can be added to the CNI (4.3.2.1). These Plug-Ins are subject of chapter 5.

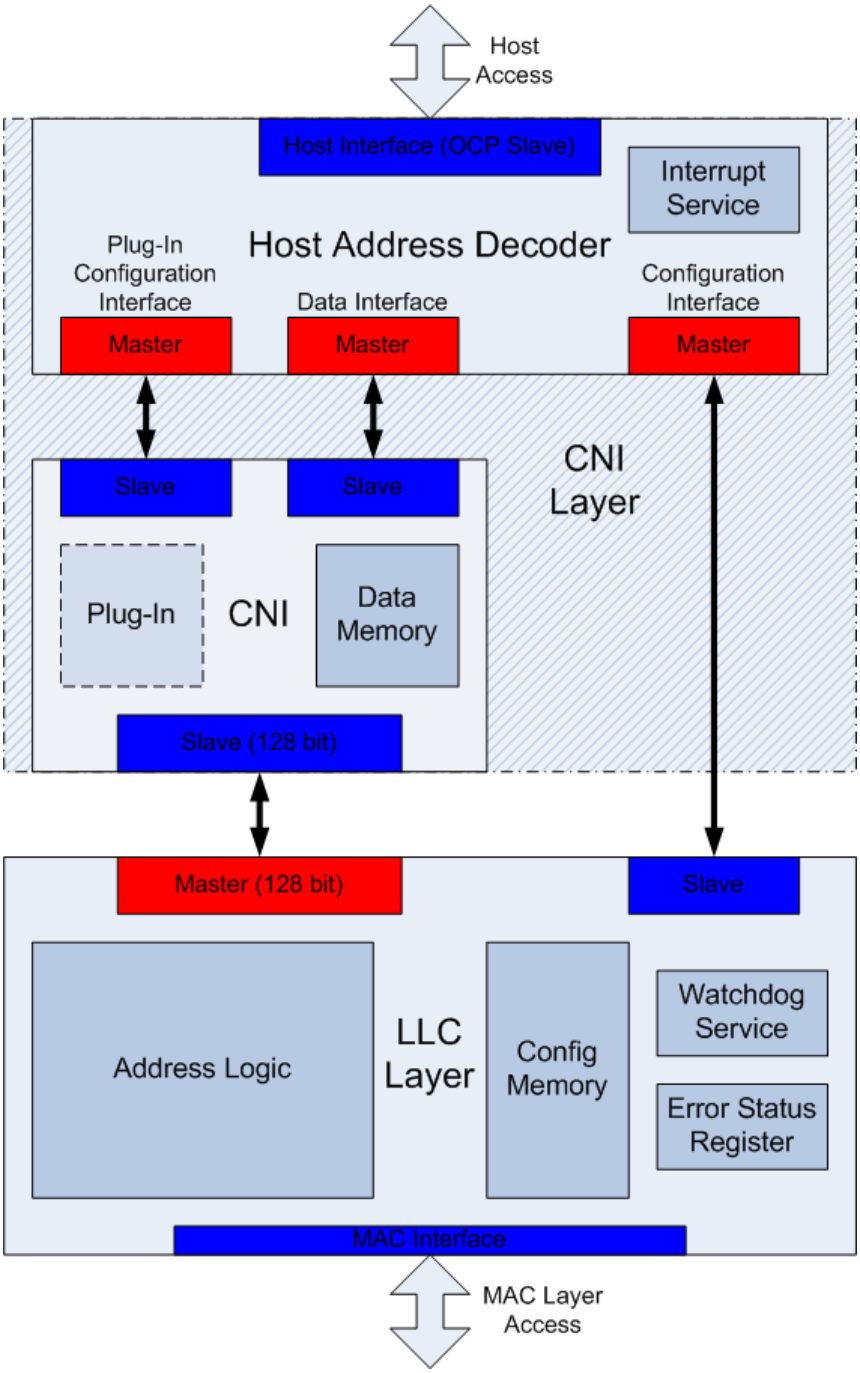

Figure 4.1: Overview of the LLC and CNI Layers

4.2 Interfaces

Two interfaces are implemented by the LLC and CNI Layers, both of them are described in the following sections: A **Medium Access Control (MAC)**

Interface to the MAC Layer and a **Host Interface** which provides access to all functions relevant to the host on a memory-mapped basis. In addition, some internal interfaces are used.

The internal interfaces are based on the Open Core Protocol (appendix A), therefore they are represented by a master and a slave instance. The data flow is controlled by the master in any case, whereas sideband signals (e. g., interrupts) may be used in both directions.

The purpose of these interfaces is to connect the LLC Layer, the CNI and the Host Address Decoder in a way that allows for an easy replacement of one of these parts, e. g., the exchange of the standard CNI with a CNI fitted with a Middleware Plug-In.

Most of the internal interfaces share the data word width with the host (32 bit), only the interface between the LLC Layer and the CNI uses the fragment size (128 bit) as data width.

4.2.1 MAC Interface

The Medium Access Control (MAC) interface is able to transfer one fragment as a whole in conjunction with the data needed to identify this fragment as a specific fragment in a specific port. In addition, some sideband signals are implemented, including configuration and time signals.

The following signals are used (input and output as seen by the LLC Layer):

PortID: input; identifies the port the data fragment belongs to (7 bit)

Fragment Position: input; identifies the position of the fragment in the pulsed data stream (8 bit)

Fragment Time: input; receiving time of the fragment (used for timestamping) (58 bit)

Receive Data: data input (128 bit)

Receive: input; signals the validity of the input data (1 bit)

Send Slot: input; used by the MAC Layer to request a specific fragment for transmission from the LLC Layer prior to reaching the send time of the fragment. When receiving this signal, the LLC Layer loads the fragment from the CNI memory and presents it on the Send Data lines while setting the Send signal. (1 bit)

Send Data: data output (128 bit)

Send: output; signals the validity of the output data (1 bit)

Time: input; provides the global real-time (58 bit)

MAC Layer Configuration Data: input; MAC Layer Configuration data read by the host (32 bit)

MAC Layer Configuration Address: output; used to address the MAC Layer Configuration memory which contains the local message schedule (9 bit)

Host Mode: input; host mode set by the TNA (1 bit)

Watchdog Period: input; the configured watchdog period (5 bit)

Reconfiguration Interrupt: input; signals the switching of MAC Layer configurations (Hostmode and Watchdog Period signals are considered valid while set) (1 bit)

All signals are sampled at the rising edge of the system clock.

Receiving A Fragment

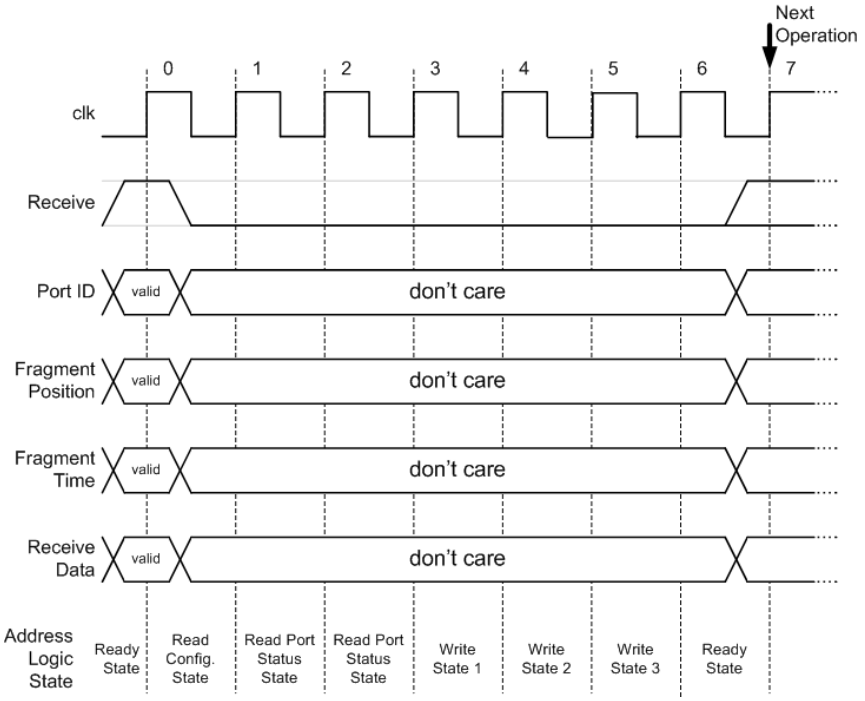

Figure 4.2: MAC Interface Receive Timing

The reception of a fragment starts with the MAC Layer asserting the *Receive* signal while the *PortID*, *Fragment Position*, *Fragment Time* and *Receive Data* signals are valid. The LLC Layer latches the data and the Address Logic starts a receive operation as described in section 4.3.1.1. After completion of the whole receive operation, which needs seven clock cycles (see figure 4.2), the LLC Layer is ready to receive or send the next fragment.

Sending A Fragment

Transmission of a fragment is initiated by the MAC Layer by setting *Send_Slot* along with *PortID* and *Fragment Position*. After latching this data, the Address Logic of the LLC Layer fetches the requested fragment from memory (see section 4.3.1.1) and drives the *Send Data* lines and the *Send* signal when finished. The whole sending procedure needs seven clock cycles to be performed (see figure 4.3).

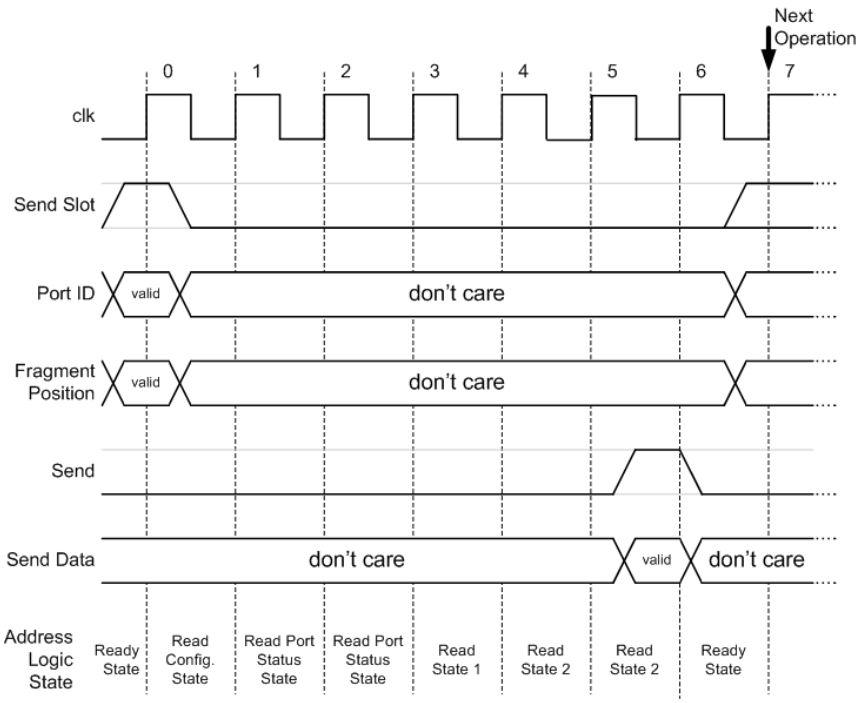

Figure 4.3: MAC Interface Send Timing

4.2.2 Host Interface

The Host Interface is explained by splitting this section into a physical and a logical description. While signal connections and timings are part of the physical Host Interface, the logical Host Interface describes addressing issues and the semantics of read or written data.

4.2.2.1 Physical Host Interface

The physical Host Interface is implemented as a 32 bit data word width Open Core Protocol (OCP) slave with 15 address lines. Since all addresses are in-

terpreted as word addresses, $2^{15} = 32768$ 32 bit words can be accessed, corresponding to an address space of 128 kilobytes. In addition, a single interrupt request (IRQ) line, supplementary flags to identify six different IRQs, and a host reset request line are provided.

The Open Core Protocol is described in detail in appendix A, table 4.1 summarizes the used OCP signals.

Name	**Signal**	**Width**	**Note**
OCP Clock	*clk*	1 bit	clock signal
Master Address	*MAddr*	15 bit	word addresses
Master Command	*MCmd*	3 bit	only "Idle", "Read" and "Write" are supported
Master Data	*MData*	32 bit	
Master Byte Enable	*MByteEn*	4 bit	byte write enable signals
Slave Command Accept	*SCmdAccept*	1 bit	
Slave Data	*SData*	32 bit	
Slave Response	*SResp*	2 bit	"Request Failed" is not used
Slave Error	*SError*	1 bit	issued on read or write errors
Slave Interrupt	*SInterrupt*	1 bit	interrupt request line
Slave Flags	*SFlag*	6 bit	IRQ selector
Slave Reset	*SReset_n*	1 bit	low-active

Table 4.1: OCP Signals used by the CNI Layer

Although the implemented Host Interface uses data handshake signals like Slave Command Accept (*SCmdAccept*) and Slave Response (*SResp*), the CNI Layer maintains a fixed read and write timing: On a "Read" command, the CNI Layer presents the requested data on the *SData* lines with a latency of one clock cycle, while a "Write" command takes immediate effect.

Interrupt Requests The CNI Layer sends an interrupt request (IRQ) to the host whenever an interrupt is generated by the MAC Layer (Reconfiguration Interrupt) or a component of the LLC Layer or the CNI Layer (i. e., the according bit in the interrupt status register (figure 4.14) is set) and the interrupt is not masked out (i. e., the according bit in the interrupt mask register is set too). The IRQ remains active until the host services the

interrupt and acknowledges it by either writing to the according bit in the interrupt status register (clearing only the current interrupt request) or the interrupt mask register (masking out all future interrupt requests).

Because of the interrupt request implementation being highly host specific (concerning mainly the number of possible IRQs), the CNI Layer implements one interrupt request line and six IRQ selector lines by using the OCP *SFlag* signal. While the main IRQ line (*SInterrupt*) is asserted on any interrupt, the six IRQ selector lines are activated only on specific interrupts, corresponding to the first six bits in the interrupt status register.

A host which allows only one IRQ per slave may only use the *SInterrupt* signal (as implemented by the Avalon Adapter, see appendix B), while a host supporting multiple interrupt sources per slave can take advantage of the *SFlag* signal to trigger different Interrupt Service Routines (ISRs), according to the occurred interrupt (as used by the Powerlink Bus Adapter, appendix C). In any case, the host processor can differentiate IRQs by examining the interrupt status register (section 4.2.2.2.3).

Reset Request The CNI Layer uses the *SReset_n* line to request the host to enter a reset state. This can become necessary after a host failure caused by a transient fault, or on startup of the whole Time-Triggered System-on-a-Chip.

Therefore, the host reset request is triggered by the watchdog service on a missed lifesign update as well as by issuing a chip-wide reset. According to the OCP specification, the reset request line is held active for at least 16 clock cycles.

4.2.2.2 Logical Host Interface

The main purpose of the LLC and CNI Layers is to provide safe real time communication services to the hosts. Furthermore, additional services are presented by the LLC Layer (e. g., a watchdog service), the CNI Layer (e. g., interrupt services), and the Plug-Ins (e. g., voting, DMA, etc.). All of these communication and additional services are accessed by the host through a memory mapped interface, the Logical Host Interface.

All memory locations used for communication, configuration and status information are subdivided into four main categories, described further in the sections below:

Ports are used for communication. Different types of input and output ports are provided, which use different synchronisation and signaling mechanisms. Port access is described in section 4.2.2.2.1.

Register File The register file informs the host about the current status of the LLC and CNI Layers (and the TTSoC in general) and is used to configure its additional services. See section 4.2.2.2.3 for details.

Port Configuration Ports are locally (at a specific core of the TTSoC) identified by their port identification number (PortID). The TNA defines send and receive parameters for every port, as well as its size and direction. All other parameters (type, address, etc.) have to be configured by the host itself, explained in section 4.2.2.2.4.

MAC Layer Configuration The MAC Layer Configuration memory stores the temporal configuration for every port and can be read by the host through the memory mapped interface (section 4.2.2.2.5).

The memory mapped interface uses 15 bit word addresses to access these locations and incorporates a 32 bit data word size.

4.2.2.2.1 Ports

All ports are stored in the CNI memory and accessed by the host through 32 bit words with consecutive 11 bit word addresses. This memory not only houses the input and output communication data, but also timestamps and some synchronization information (the port flags (PF)). This additional information may differ according to the direction, type, and the configuration of the port, but this data is located at the base address of the port (and the consecutive addresses in case of timestamps) in any case.

To simplify synchronization mechanisms, three bits of the 15 bit word address act as additional high active byte write enable flags, attached to the lower three bytes of a data word. Furthermore, the most significant address bit has to be set to `1` to distinguish between a port access and a register/port configuration/MAC Layer Configuration read or write operation. Figure 4.4 shows the layout of a port address.

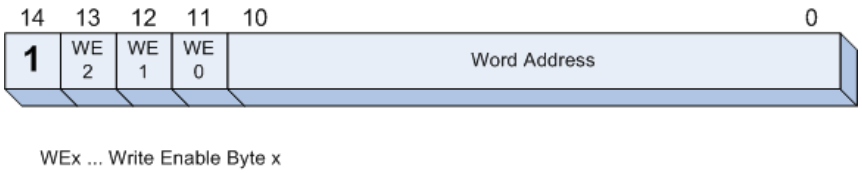

Figure 4.4: Port Address

State Ports

Figure 4.5 shows a state port of length N (fragments). The timestamp is only implemented for input state ports while the shadow fragments are only used by output state ports.

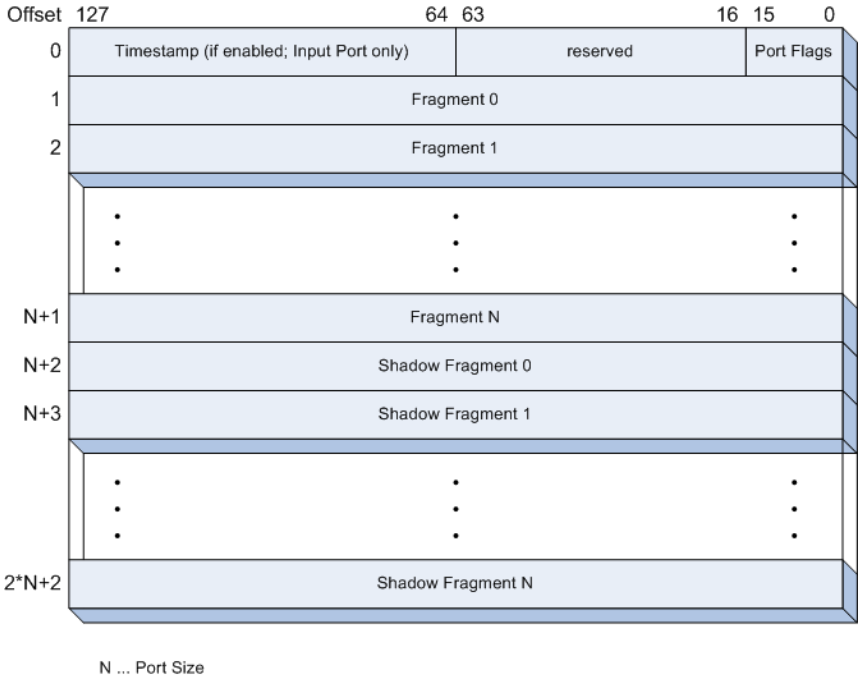

Figure 4.5: State Port

Input State Ports Input state ports use a 4 bit sequencer and a low-active empty flag for synchronization (shown in figure 4.6), both exclusively written by the Address Logic. An empty flag with value `0` signals an empty port, the memory contents are invalid.

The sequencer is used to avoid the reading of incomplete messages by the host. The Address Logic increments the sequencer at the beginning and after a complete port update (all fragments). Before any read attempt, the host has to check the sequencer for an odd value, which means an update is in progress, rendering the current port data invalid. In case of an even sequencer value, the host has to recheck the sequencer after the reading operation, to ensure that no update has taken place (and invalidated the read data) in the meantime.

Every input state port can be configured to store a timestamp on reception of the last fragment. This timestamp is located in the upper 64 bits of the 128

bit data word at the port base address, and is valid the same time as the port data.

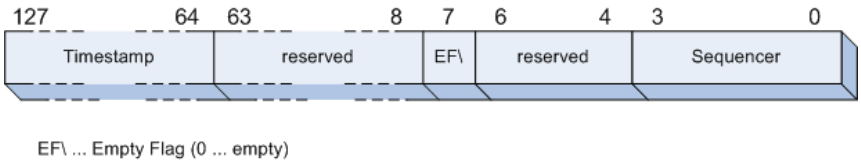

Figure 4.6: Input State Port Port Flags

Output State Ports To prevent the sending of invalid messages, output state ports use a "shadow register" and two flags, a "valid" flag written by the host, and a "using" flag written by the Address Logic. Figure 4.7 shows the location of these flags.

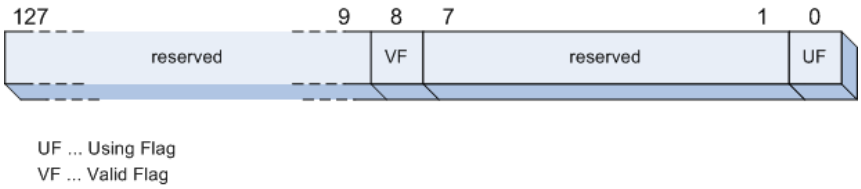

Figure 4.7: Output State Port Port Flags

The "using" flag indicates which register is used for transmission by the LLC Layer (`0` ... standard, `1` ... shadow). It is set to the "valid" flag value at the beginning of the sending of a whole port.

When sending a message, the host writes the message to the register *not* indicated by the two flags, i. e., the host is only allowed to update a register if *both* flags point to the other register. After updating the register, the host sets the valid flag accordingly. The next time the LLC Layer starts the transmission of the port, it sets the "using" flag and the host is allowed to update the other register.

To set/reset the "valid" flag without unintentionally changing the "using" flag, the byte write enable bits in the address have to be set to "`010`".

Streaming Ports

Streaming ports store only one fragment of a port at any time. The number of the last received or transmitted fragment is stored with the portflags, as seen in figures 4.8 and 4.9, respectively.

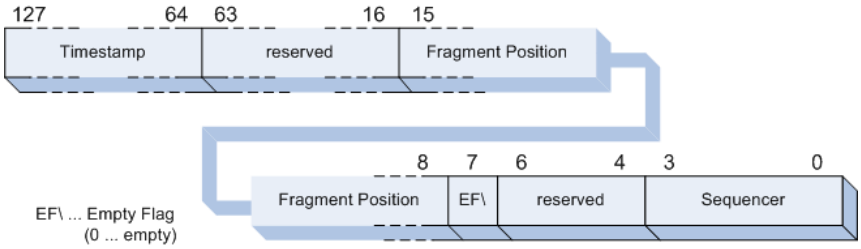

Figure 4.8: Input Streaming Port Port Flags

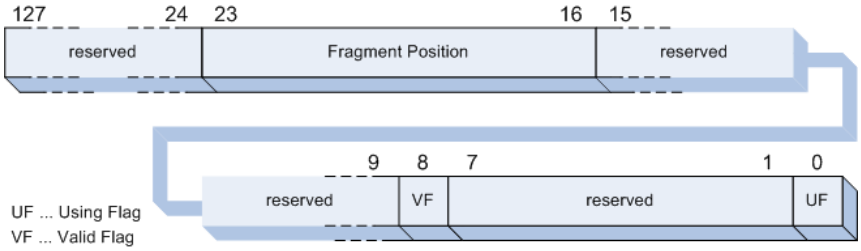

Figure 4.9: Output Streaming Port Port Flags

In all other concerns, streaming ports are treated like state ports with length 0. They use the same mechanisms for synchronization and timestamps. See section 4.2.2.2.1 for details.

Event Ports

Event Ports provide queues to allow reception or sending of multiple event messages, without the need for immediate host processing.

These event port queues are implemented as ringbuffers with configurable length (a maximum of 16 queue positions can be maintained, see section 4.2.2.2.4 for port configuration details). The port data is stored successively in the different queue positions after the port flags (or the input port timestamps, if enabled). See figure 4.11 for details.

Event ports use the same synchronization mechanisms for input and output ports, except for timestamps, which are only used on input ports (if enabled), and written exclusively by the Address Logic. Either the host (in case of an output port) or the Address Logic (input port) functions as the *writer*, the other one is called the *reader*.

The port flags of an event port (figure 4.10) store the current read and write positions in the queue as well as the Position Changed Flag (PCF), which

indicates if the *reader* or the *writer* was the last one to change the queue positions.

The Internal Flags (IFs) are used by the Address Logic for queue overflow and queue empty checking (bit 15 if the Address Logic is acting as *reader*, bit 7 if the Address Logic is the *writer*), and should be ignored by the host.

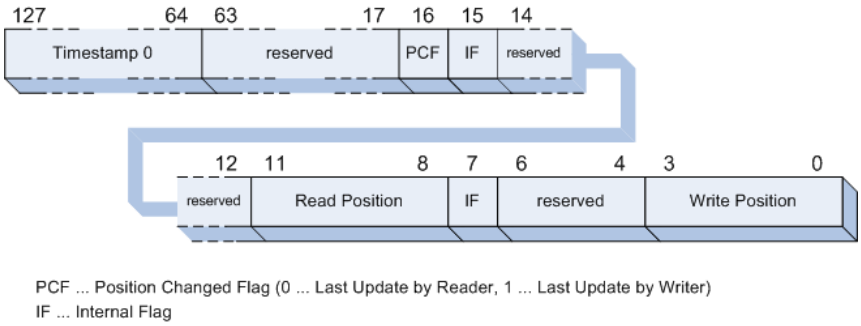

Figure 4.10: Event Port Port Flags

A read or write operation is started by reading the port flags. Two different situations are possible:

- If the read and write positions are different, the port data can be read from/written to the location indicated by the according position. After the operation is complete (all fragments have been read/written), the according position is incremented (modulo queue length), the PCF is set to `0` in case of a read operation or `1` in case of a write operation, and both are stored in the port flags. The host has to set the *byte write enable* bits to "`101`" to write the write position or "`110`" to write the read position.[1]
- If both positions are equal, the queue is either full or empty. These two conditions can be distinguished by the Position Changed Flag (PCF). A PCF value of `0` means the last operation was done by the *reader*, so the queue is empty and no read operation is permitted, but a write operation can occur. If the last operation was done by the *writer* (PCF equals `1`), the queue is full. No writes are allowed, whereas a read operation may take place.

An input event port is able to store timestamps, one for every queue position. The first timestamp (attached to queue position 0) is stored in the upper 64 bits of the 128 bit data word at the port base address. All other timestamps are located in the succeeding data words, two timestamps per 128 bit word

[1]The *reader* is not allowed to change the write position (and vice versa). This would lead to uncontrolled behavior and invalidation of the port data.

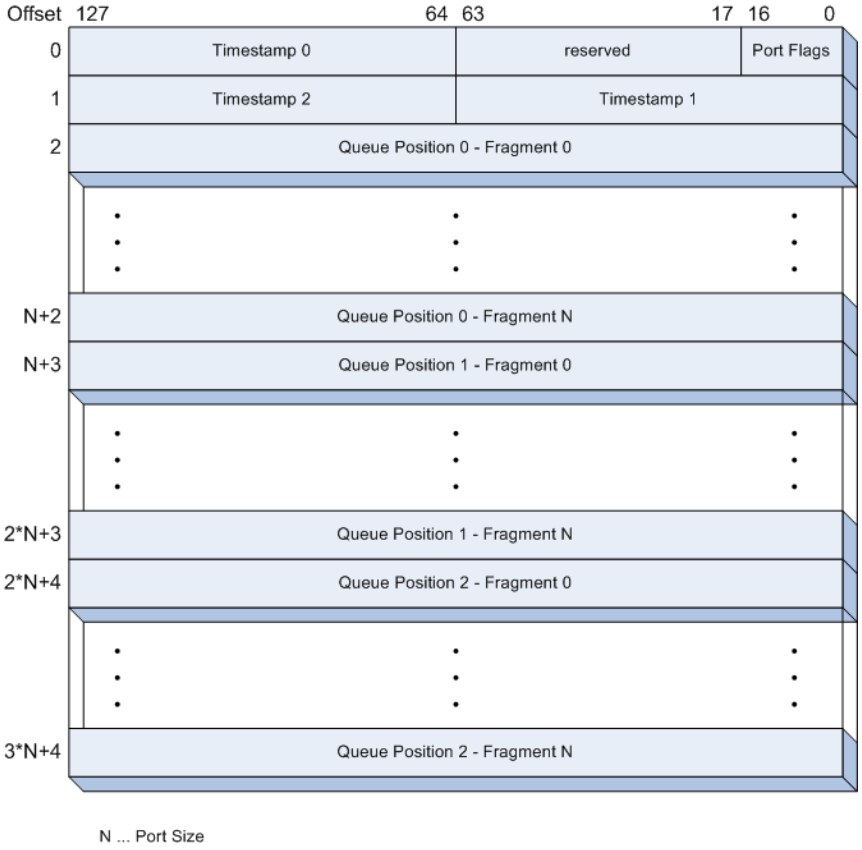

Figure 4.11: Event Port (Queue Length 2, Timestamps Enabled)

(i. e., the data word with the address $PORT\_BASE\_ADDRESS + 1$ houses the timestamp for queue position 1 in the lower 64 bits and the timestamp for queue position 2 in the higher 64 bits, etc.).

Figure 4.11 shows an input event port with queue length 2 and enabled timestamps.

4.2.2.2.2 Port Memory Usage

Every port uses a different amount of memory, depending on its configuration (size (N), type, timestamps (TS) and queue length (QL)). Table 4.2 summarizes the calculation of port memory usage.

	Output
Streaming	3
State	$2 * (N + 1) + 1$
Event	$(N + 1) * (QL + 1) + 1$

	Input (without TS)	Input (with TS)
Streaming	2	2
State	$N + 2$	$N + 2$
Event	$(N + 1) * (QL + 1) + 1$	$(N + 1) * (QL + 1) + \lceil QL/2 \rceil + 1$

Table 4.2: Port Memory Usage (128 Bit Fragments)

4.2.2.2.3 Register File

The register file provides status information about the NoC and is used to configure some additional services. To access the register file, the three most significant bits of the word address are set to a value of "000". Bits 4 to 7 indicate the Plug-In address, so every Plug-In can implement its own register file (the LLC Layer and CNI Layer register file is accessed by Plug-In address 0 ("0000")). The lowest four bits address a specific register in the LLC and CNI Layers or a Plug-In register file. Register addressing is shown in figure 4.12.

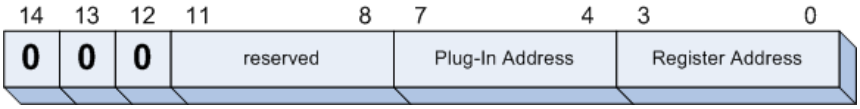

Figure 4.12: Register Addressing

LLC Layer and CNI Layer Registers

The LLC Layer and CNI Layer register file is depicted in figure 4.13. All registers are 32 bit wide and considered as read/write accessible unless mentioned otherwise. The following paragraphs describe them in detail.

Global Real-Time The first two (read only) registers store the 64 bit wide global real-time. Whenever the lower 32 bits of the time are read by the host (register address "0000"), the upper 32 bits of the time (register address "0001") are stored and frozen to ensure consistency.

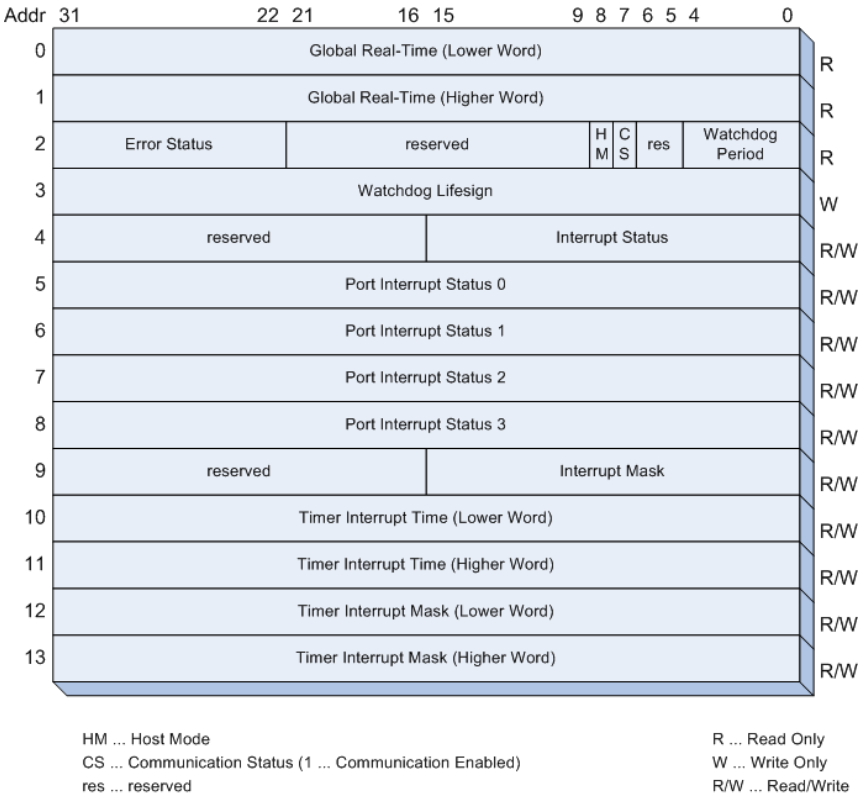

Figure 4.13: LLC Layer and CNI Layer Register File

Error Status, Host Mode, Communication Status & Watchdog Period The (read only) register at address 2 ("0010") stores the watchdog period (a value of "11111" indicates a disabled watchdog), the Host Mode set by the TNA and the current communication status (1 ... communication enabled), as well as the LLC Layer and CNI Layer error status (as described in section 4.3.1.4).

Watchdog Lifesign This write only register (register address 3 ("0011")) has to be updated regularly by the host with a value of 0x55555555 according to the watchdog period. Failure to do so results in a reset of the host and the LLC and CNI Layers (except the Error Status Register and the Watchdog Service).

Interrupt Status All interrupts triggered by the TISS and the CNI Layer are recorded and stored in this register (register address 4 ("0100")). The interrupt status register is shown in detail in figure 4.14. Every interrupt is represented by its own bit in the register, a value of 1 indicates a pending interrupt. Writing

1 to a specific position immediately clears the according interrupt (no matter if it was set or not).

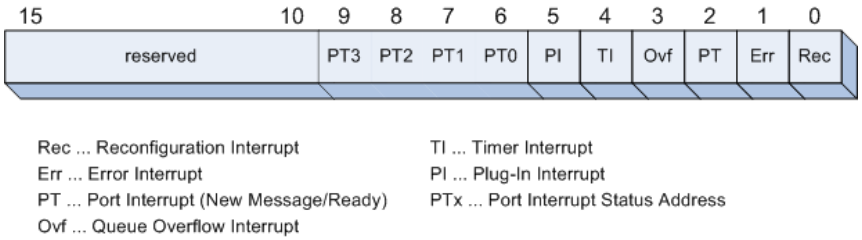

Figure 4.14: Interrupt Status Register

The port interrupt status address bits are set together with the port interrupt bit. They indicate the appropriate port interrupt status register for the activated port interrupt (i. e., PT0 . . . port interrupt status register 0, etc.) and have to be cleared explicitely (by writing 1), but do not trigger any interrupt requests.

Port Interrupt Status (0-3) Every port can generate a new message (input ports) or a ready (output ports) interrupt (see section 4.3.1.1 for details). In addition to activating a general port interrupt, any occurrence of these interrupts is stored on a one-bit-per-port basis (i. e., bit 1 of word 0: port 1, bit 2 of word 0: port 2, . . . bit 31 of word 3: port 127). The port interrupt status address bits indicate the matching word for the activated port interrupt. The port interrupt status bits are cleared by writing 1 to them after clearing the port interrupt bit in the interrupt status register.

Interrupt Mask This register is used to mask out specific interrupts not needed by the host. The layout matches the interrupt status register shown in figure 4.14. A set bit allows the according interrupt request, a cleared one masks it out. This only affects the IRQs sent to the host, the triggered interrupts are recorded in the interrupt status register in any case.

Timer Interrupt Time & Timer Interrupt Mask The timer interrupt service of the CNI Layer (described in section 4.3.2.3) is configured through these four registers.

To avoid wrong interrupts while configuring the timer interrupt service, the timer interrupt is disabled when the mask register (lower & higher word) or the

lower word of the time register is written and (re-)enabled after the complete higher word[2] of the time register is written.

Plug-In Registers The various Plug-In register files are accessed by setting the address bits 4 to 7 to the corresponding Plug-In address. The specific Plug-In register files are described in detail in chapter 5.

4.2.2.2.4 Port Configuration

A port is locally identified by its seven bit wide PortID. The TNA defines the size and the send/receive parameters for every port, while the following parameters have to be defined by the host:

Direction: Defines the direction of a port (1 bit) (`0` ... input port; `1` ... output port)

Size: Port size in fragments (8 bit)

Type: Port type (2 bit) ("`00`" ... not configured; "`01`" ... Streaming Port; "`10`" ... State Port; "`11`" ... Event Port)

Timestamp Enable: Enables timestamps for an input port (1 bit) (`1` ... timestamps enabled)

Interrupt Enable: Enables the port interrupt (new message/ready interrupt) (1 bit) (`1` ... interrupt enabled)

Queue Length: Defines the queue length of an event port (4 bit)

Base Address: Specifies the base address of a port (9 bit). Since the CNI memory is organized in 128 bit words, this address is two bits shorter than the address used by the host to read port data (using 32 bit words). To avoid overlapping of ports, port base addresses have to be calculated carefully, taking the memory usage of all preceding ports (in CNI memory, not in the configuration) into account (see section 4.2.2.2.2).

The LLC Layer implements two configuration memories, an "active" and a "shadow" configuration memory. The "active" configuration is used by the LLC Layer to maintain operation while the "shadow" memory can be written arbitrarily by the host. A specific bit (the Configuration Complete Flag (CCF)) in the configuration data triggers the switching of the two

[2]The timer interrupt is enabled after writing byte 0 of this 32 bit word, therefore the byte write enable lines have to be set accordingly.

configurations, immediately canceling any send/receive operation in progress.[3]

When switching to a new configuration, the host is responsible to clear the port flags of all ports in the new configuration to prevent the LLC Layer from sending or receiving invalid data over the NoC.

Two additional flags allow the host to suppress communication as long as the clearing operation is in progress: a Communication Disable Flag (CDF) and a Communication Enable Flag (CEF). Whenever the CDF is set to `1`, no sending or receiving of fragments takes place until communication is enabled again by setting the CEF to `1`. The current communication status is stored in a register and can be examined by the host at any time.

The layout of the configuration data for a single port as described above is depicted in figure 4.15.

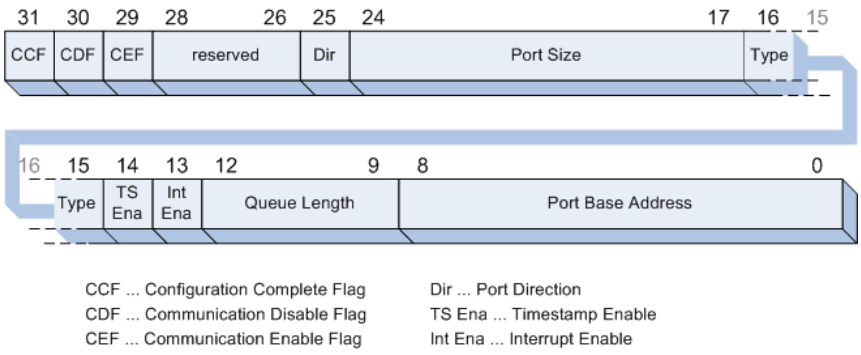

Figure 4.15: Port Configuration Data

Port Configuration Addressing A port configuration write access is characterized by the three most significant bits of the address set to a value of "`001`" and the PortID of the port to be configured in the least significant seven bits, as shown in figure 4.16. Read accesses to the configuration address space are ignored.

An exception is port number 0, which is reserved for diagnostic purposes. All configuration data written to this PortID is ignored, only the flags (CCF, CDF, and CEF) are recognized.

[3]It is recommended to set the CCF by writing to PortID 0 after the complete configuration is written, to prevent configuration data loss caused by setting the byte enable lines in the wrong order, which may lead to a configuration switch before the last port configuration data is written completely.

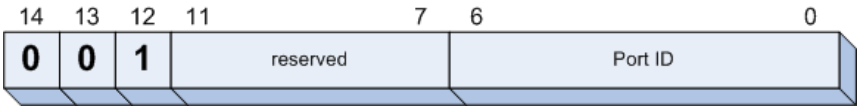

Figure 4.16: Port Configuration Addressing

4.2.2.2.5 MAC Layer Configuration

The host has to be able to read the MAC Layer Configuration data stored in the MAC Layer of the TISS. Therefore, a special address space is reserved for MAC Layer Configuration access, depicted in figure 4.17.

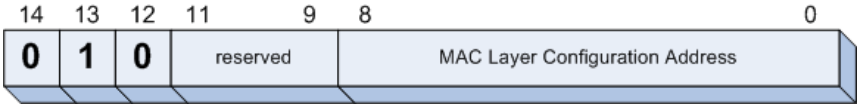

Figure 4.17: MAC Layer Configuration Address Space

The three most significant bits must have a value of "`010`" to identify a MAC Layer Configuration access. The MAC Layer Configuration address itself is 9 bit wide, the result of a read access is a 32 bit data word. Write attempts to the MAC Layer Configuration address space are ignored.

See [Eng07] for a detailed description of the MAC Layer.

4.3 Implementation

The LLC and CNI Layers consist of several components, each of which has to fulfill specific tasks to provide the whole variety of services of the Network-on-a-Chip (NoC) to the host. These components and their tasks are explained in detail in the following sections.

4.3.1 LLC Layer

The LLC Layer is considered to be static, in contrast to the CNI Layer, which may be changed to support a different host, or house different Plug-Ins. Some of the most elementary functions of the NoC are located in the LLC Layer, e. g., providing port synchronization mechanics, or the possibility to configure the way incoming pulsed data streams are interpreted. In addition, some functions to ensure the correct operation of the whole Time-Triggered

System-on-a-Chip are part of the LLC Layer. For instance, the watchdog service, the host mode register, and an error status register are used to record, report, and react to faulty behavior of the host attached to the CNI Layer.

Figure 4.18 shows control and data flow between the components of the LLC Layer.

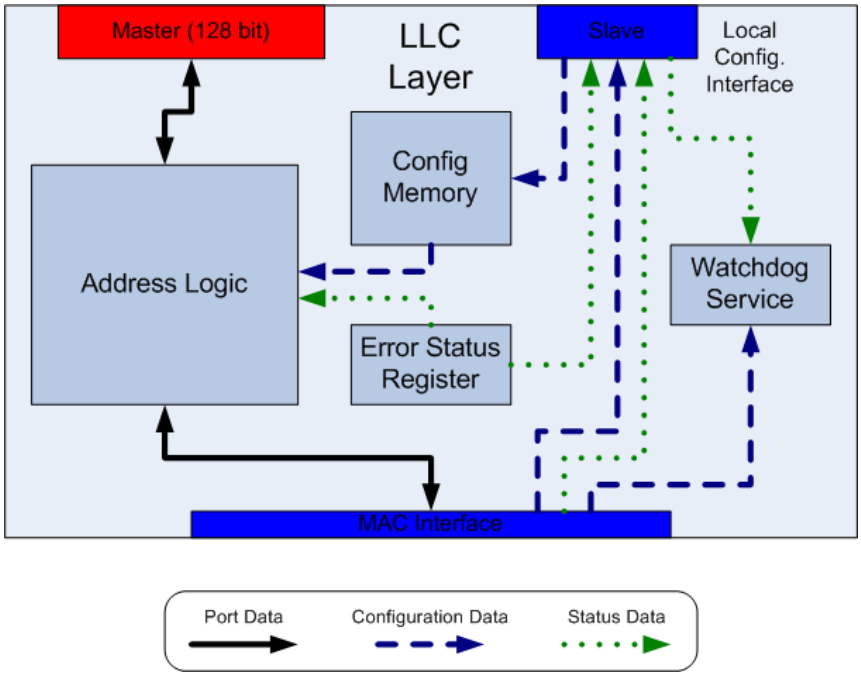

Figure 4.18: Control and Data Flow inside of the LLC Layer

Besides connecting these components, the LLC Layer stores the Host Mode set by the TNA and is responsible to route “Write” and “Read” commands issued via the local Configuration Interface to the addressed component.

4.3.1.1 Address Logic

The main purpose of the Address Logic is to store incomming fragments at their designated memory address as well as loading fragments requested for transmission from memory. In addition, the Address Logic maintains the necessary synchronization information for every single port, which is stored together with the port data in the CNI memory. Section 4.2.2.2.1 explains the operation of the synchronization mechanisms in detail.

The Address Logic is implemented as a Finite State Machine (FSM), depicted in figure 4.19. Every received or requested fragment is processed by cycling through the states as indicated by the arrows in the diagram. Depending on the type of the port the fragment belongs to, different actions are taken in each state until the store or load operation is completed and the port synchronization information is updated accordingly.

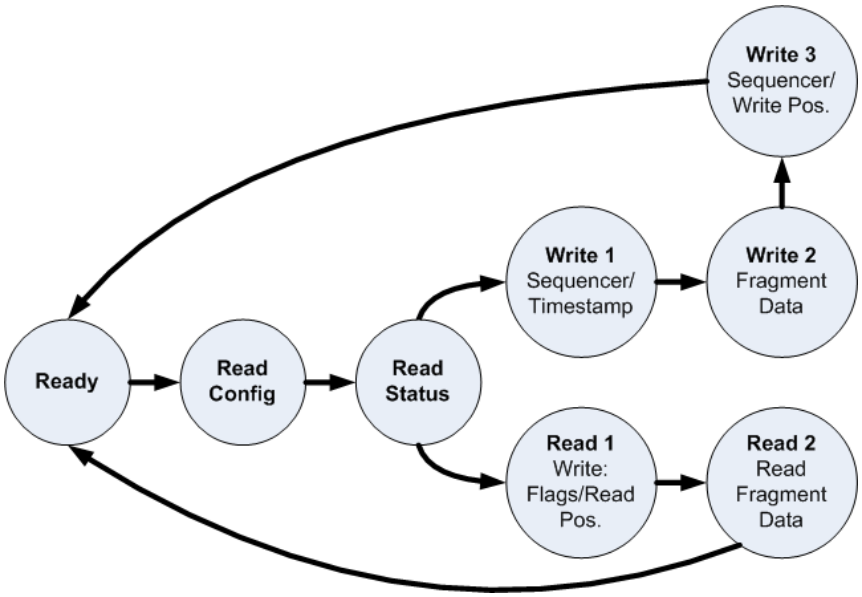

Figure 4.19: Address Logic Finite State Machine

This cycle is interrupted only if an error is detected (queue overflow, CNI memory error, or port configuration error), the host validates a new port configuration, or the chip-wide reset signal is asserted. In any of these cases, the current transfer operation is aborted immediately and the Address Logic enters the ready state.

Some remarks concerning the operation of the Address Logic have to be mentioned:

- Any send or receive request, with the exception of a send request to the diagnostic port, is ignored by the Address Logic while the communication is disabled by the host (communication status ... `0`, see section 4.2.2.2.4). As a reaction, a communication error is triggered to indicate the failed communication attempt by the MAC Layer.
- A send request to port 0, the designated diagnostic port (see section 4.3.1.4 for details), is answered by providing the contents of the error status register in the least significant bits of the send data and clearing

the error status register immediately afterwards, regardless of the current communication status.

- The Address Logic expects the CNI to confirm a "Write" command immediately (in the same clock cycle) by setting *SCmdAccept* and to respond to a "Read" command with a latency of one clock cycle by driving the "Data Valid" signal on the *SResp* lines. A "Write" command not accepted immediately leads to an abortion of the currently active fragment transfer, as well as a "Response Error" signal on the *SResp* lines in case of a "Read" transfer does, both indicated by triggering a memory error. Since the "Read" command response time is not measured by the Address Logic, the CNI is responsible to react within the given latency to prevent a disturbance of the Medium Access Control Interface timing.
- The correct fragment addresses in the CNI memory are calculated after reading the port flags using the formulas listed in table 4.3.

The following paragraphs describe the states of the Address Logic FSM and the different actions taken according to the port type of the currently transfered fragment. In any case, the transfer is completed after seven clock cylces to meet the Medium Access Control Interface timing requirements.

Ready State The Address Logic waits for a *Receive* or a *Send_Slot* signal from the MAC Layer to start a new fragment transfer. On reception of one of these signals, the *PortID* and *Fragment Position* signals, as well as the *Fragment Time* and the *Receive Data* signals in case of a fragment reception, are stored and a transition to the Read Configuration State is initiated. If the communication was disabled by the host, a receive transfer is aborted immediately, resulting in a communication error.

Read Configuration State If the communication was disabled by the host, the active transfer is aborted, a communication error is triggered, and the Ready State is entered again. Otherwise, before advancing to the Read Port Status State, the configuration data of the port indicated by the PortID is read and stored.

A send request to the diagnostic port is answered in any case while the Address Logic is in this state, followed by an error-free transition to the Ready State.

Read Port Status State The port flags, located at the port base address, are read from the CNI memory in this state. Afterwards, either Read State 1

Output Port	
Streaming	$BASE\_ADDR + UF + 1$
State	$BASE\_ADDR + (PS + 1) * UF + FP + 1$
Event	$BASE\_ADDR + (PS + 1) * RP + FP + 1$

Input Port (without Timestamps)	
Streaming	$BASE\_ADDR + 1$
State	$BASE\_ADDR + FP + 1$
Event	$BASE\_ADDR + (PS + 1) * WP + FP + 1$

Input Port (with Timestamps)	
Streaming	$BASE\_ADDR + 1$
State	$BASE\_ADDR + FP + 1$
Event	$BASE\_ADDR + (PS + 1) * WP + TSO + FP + 1$

Abbreviations		
$BASE\_ADDR$	port base address	0 - 512
UF	"using" flag	0/1
PS	port size (fragments)	0 - 255
FP	fragment position	0 - 255
RP	read position	0 - 15
WP	write position	0 - 15
TSO	time stamp offset	$\lfloor (QL + 1)/2 \rfloor$
QL	queue length	0 - 15

Table 4.3: Fragment Address Calculation

(output port, send transfer) or Write State 1 (input port, receive transfer) is entered.

The following states perform different actions according to the port type addressed by the transfer.

Read State 1

Streaming Port: The "using" flag, set to the value of the "valid" flag, and the fragment position are written to the CNI memory. In addition, if enabled, a ready interrupt is triggered.

State Port: When processing the first fragment of a pulsed data stream, the "using" flag is set to the value of the "valid" flag and written to the CNI

memory and a ready interrupt is triggered if enabled. No action is taken on other fragments.

Event Port: Six possible port states, indicated by the port flags, have to be considered at this point. Table 4.4 summarizes these states and the actions taken as a consequence.

Queue Status	IF	First Frag.	Last Frag.	Action
not empty	0	X	no	continue
not empty	0	X	yes	write read position & PCF, continue
empty	X	yes	no	set IF, abort
empty	X	yes	yes	abort[4]
X	1	no	no	abort
X	1	no	yes	clear IF, abort

Table 4.4: Output Event Port Behavior (X . . . don't care)

The **Queue Status** is determined by comparing the read and write positions. If both positions are equal, signaling a full or an empty queue, the PCF is checked. A PCF value of `0` indicates that the last operation performed on the queue was a read operation by the Address Logic, so the queue is empty.

Since the Queue Status is calculated on every fragment request, a change of the queue contents (e. g., the host writing a new message and adjusting the write position) could lead to a change of the Queue Status before a whole pulsed data stream is processed completely. To prevent the Address Logic from sending corrupted data in such a case, an **Internal Flag (IF)**, stored with the port flags, preserves the Queue Status calculated on the first fragment of a pulsed data stream until it is completed. No port data is transmitted while this flag is active.

When processing the last fragment of a pulsed data stream, the new read position is actually written *before* the fragment is read to gain an extra clock cycle for address calculation. The last fragment is still valid at the next clock cycle, since the host has to read the updated port flags before overwriting the port data, which takes at least one clock cycle.

The aborted Event Port transfers due to an empty queue, as shown in table 4.4, force the FSM into the Ready State, while the regular next state is Read State 2, regardless of the processed port type.

[4]Port Size is 0 (one fragment), so no Internal Flag is needed.

Read State 2 The requested fragment is read from the CNI memory, which takes two clock cycles, and transferred to the MAC Layer. An Event Port can trigger a ready interrupt in this state, depending on the port configuration.

Since a sending operation is completed after performing these actions, the Address Logic FSM enters the Ready State again, awaiting the next send or receive request.

Write State 1

Streaming Port: The sequencer is incremented by 1 (to an odd value) and stored in the CNI memory together with the requested fragment position.

State Port: On the first fragment of a pulsed data stream, the sequencer is incremented by 1 (to an odd value) and stored in the CNI memory. Since the sequencer still contains an odd value, no action is taken when processing other fragments.

Event Port: As in Read State 1, six possible port states can be distinguished in this state, listed in table 4.5.

Queue Status	IF	First Frag.	Last Frag.	Action
not full	0	X	no	continue
not full	0	X	yes	write timestamp (if enabled), continue
full	X	yes	no	set IF, signal overflow error & interrupt, abort
full	X	yes	yes	signal overflow error & interrupt, abort[5]
X	1	no	no	abort
X	1	no	yes	clear IF, abort

Table 4.5: Input Event Port Behavior (X . . . don't care)

The **Queue Status** is determined by comparing the read and write position of the queue and examining the Position Changed Flag (PCF). Equal read and write positions and a PCF of value `1` (last operation was a write operation performed by the Address Logic) indicate a full queue.

To prevent port data corruption, an **Internal Flag (IF)** is set if the target queue is full when processing the first fragment of a pulsed data stream, similar to Read State 1. No fragment data is stored if this flag is

[5]Port Size is 0 (one fragment), so no Internal Flag is needed

set. In addition, an overflow error and an overflow interrupt are triggered to inform the host and an optional external diagnostic entity about the data loss. The Internal Flag is cleared upon completion of processing the last fragment of the pulsed data stream.

Since the timestamp represents the event of receiving the fragment by the MAC Layer, it can be written *before* the fragment data is stored in the CNI memory.

The next state in the Address Logic FSM is Write State 2, unless the operation is aborted due to a full Event Port queue.

Write State 2 After the fragment data is written to the appropriate address, Write State 3 is entered.

Write State 3

Streaming Port: The sequencer is incremented again (to an even value) and stored in the CNI memory together with the fragment position and the fragment timestamp. If enabled, a new message interrupt is generated.

State Port: If processing the last fragment of the pulsed data stream, the same procedures as if processing a Streaming Port are performed, except writing the fragment position. Otherwise, no action is taken.

Event Port: The updated write position and the PCF is written to the CNI memory if processing the last fragment of the pulsed data stream. In addition, a new message interrupt is triggered if enabled. No action is taken if processing any other fragment.

Write State 3 is completed by a transition to Ready State.

4.3.1.2 Configuration Memory

Configuration of the TISS is a two-stage process. The message schedule is configured by the TNA and stored in the MAC Layer. Furthermore, the TNA configures the host mode, stored in the LLC Layer, and the watchdog period used by the Watchdog Service (4.3.1.3). All other configuration data is set by the host.

The Port Configuration Memory is written by the host and stores the port configuration data (i. e., port base address, queue length (event ports only), port interrupt enable, timestamp enable (input ports only), port type, port size (in fragments), port direction (input/output)) and the current communication status (communication enabled/disabled).

Since the Configuration Memory has to provide valid port configuration data to the Address Logic at every fragment send or receive event, which are not necessarily synchronized with the host operation, the Configuration Memory uses two logical memories: An "active" memory, which keeps a valid configuration and can be read by the Address Logic at any time, and a "shadow" memory written by the host to build a new configuration. By setting the Configuration Complete Flag (CCF), the host declares the new configuration valid, and the two memories are swapped. Port configuration details are explained in section 4.2.2.2.4.

To preserve on-chip resources (e. g., die area, power consumption), the two logic memories are implemented by using a single Simple Dual-Port Memory twice the size of one complete port configuration. This Simple Dual-Port Memory provides one read port (for the Address Logic) and one write port (for the host), which can be accessed concurrently.

The distinction of the "active" and the "shadow" logical memory is managed by the Most Significant Bit (MSB) of the memory address. The value of this bit, pointing to the "active" half of the memory, is stored in a register and toggled by the host by setting the Configuration Complete Flag.

The 26 bit configuration data of a single port is addressed by its seven bit PortID, providing a maximum of 127 active ports and the diagnostic port with PortID 0. As a consequence, the Configuration Memory uses a 256x26 bit memory with 8 bit addresses (1 bit "active"/"shadow", 7 bit PortID) to maintain all port configuration issues, depicted in figure 4.20.

The drawback of this solution is the inability of the host to read any configuration data, once stored. Therefore, the host has to keep a mirror image of the port configuration data in its main memory if this information is needed after completing the port configuration process.

Since most applications will not need this information anyway, this solution was chosen in favour of a much more complex implementation consisting of two Dual-Port Memories, a set of multiplexers, and a write protection logic to prevent the host from accidentially overwriting an active configuration.

The communication status, stored in a register, is set to "enabled" after startup and can be changed by the host by setting the Communication Disable Flag (CDF) while writing port configuration data. When disabled, communication can be re-enabled by setting the Communication Enable Flag (CEF). Clearing both flags simultaneously leaves the communication status unchanged, while setting both flags simultaneously disables communication.

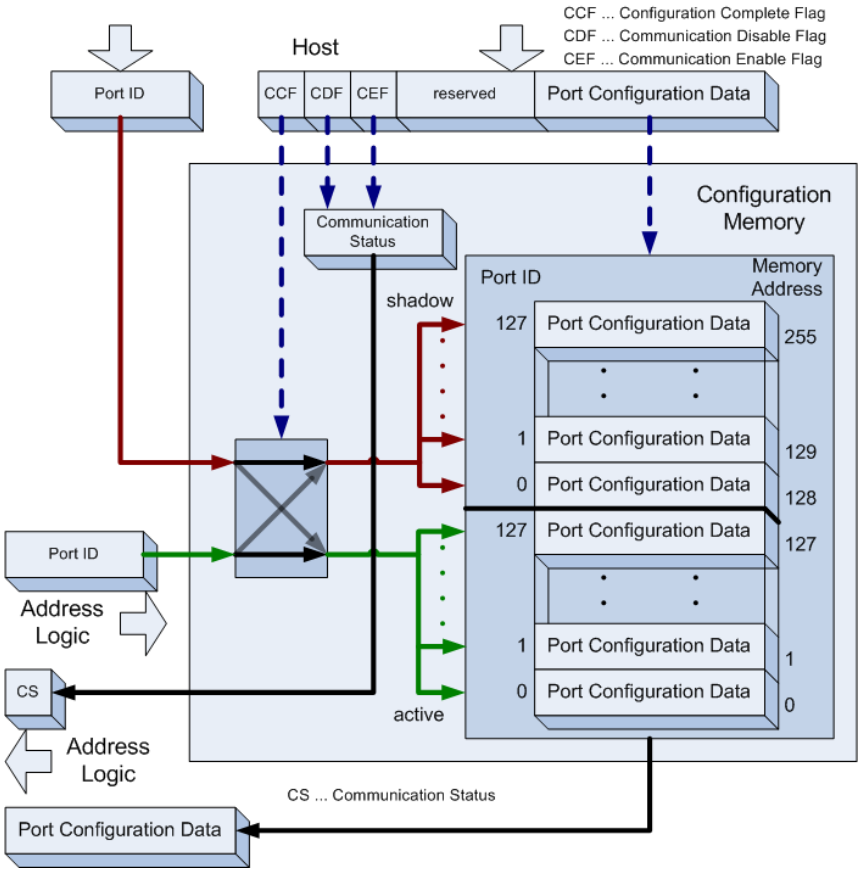

Figure 4.20: Operation of the Configuration Memory

4.3.1.3 Watchdog Service

The Watchdog Service is used to detect a crashed host processor. Since all other services of the LLC and CNI Layers do not require the host to respond (i. e., all LLC Layer operations continue despite of a failed host), the Watchdog Service is the only possibility of the NoC to identify a crash or omission failure [Lap92, p. 17] of the host.

The Watchdog Service requests the host to update a 32 bit register (the lifesign register) with a value of `0x55555555` on a periodic basis. Failure to do so will result in an immediate reset of the host and the LLC and CNI Layers (except the Error Status Register and the Watchdog Service), to overcome transient faults. In addition, the Watchdog Miss error flag in the Error Status Register is set in such a case.

The time period, in which the lifesign register has to be updated at least

once (called the Watchdog Period), is configured by the TNA. On every Reconfiguration Interrupt assigned by the MAC Layer, the new Watchdog Period is stored in a register and validated at the next period start.

If not needed, the Watchdog Service can be disabled entirely by the TNA by setting the Watchdog Period to 31 ("`11111`").

4.3.1.4 Error Status Register

Every error detected by the LLC and CNI Layers is stored in the Error Status Register to inform the host and an optional diagnostic entity about it, so that corrective actions can be taken. All error flags are set on occurrence of the error and cleared by the Address Logic every time they are read through the diagnostic port (PortID 0) and a diagnostic message is sent. The host can read the error flags at any time, but it has no possibility to clear the Error Status Register.

The following error status flags are implemented (as shown in figure 4.21):

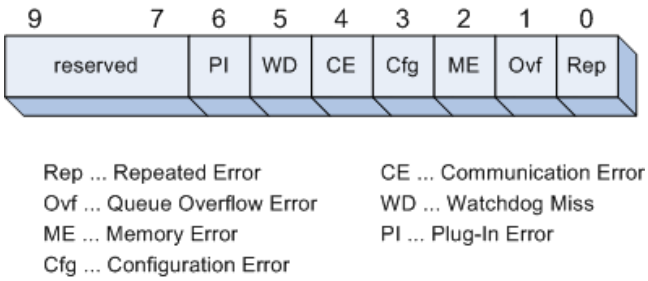

Figure 4.21: Error Status Flags

Repeated Error (Rep): an error occurred more than once.

Overflow Error (Ovf): an event port overflow occurred.

Memory Error (ME): the CNI memory did not react according to specification.

Configuration Error (Cfg): a port addressed by the MAC Layer was not configured correctly.

Communication Error (CE): the MAC Layer initiated a communication attempt while communication was disabled.

Watchdog Miss (WD): the host failed to update the watchdog lifesign according to specification.

Plug-In Error (PI): a Plug-In error occurred (see chapter 5 for details).

4.3.2 CNI Layer

The CNI Layer consists of the Communication Network Interface (CNI), responsible for storing all port data, and the Host Address Decoder, which provides an interface to the host processor and houses an Interrupt Service. The CNI Layer may be adapted to the requirements of different applications by exchanging the standard CNI (4.3.2.1) with an extended CNI fitted with one or more Plug-Ins, as described in chapter 5.

The control and data flow between the components of the CNI Layer is depicted in figure 4.22.

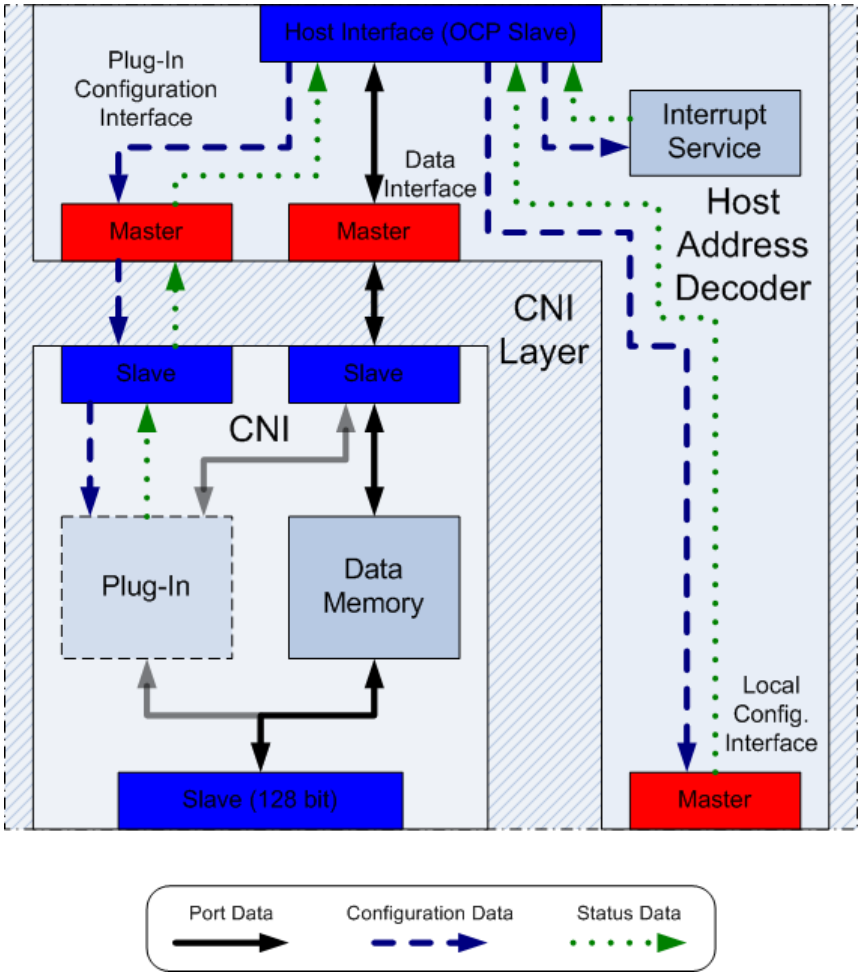

Figure 4.22: Control and Data Flow inside the CNI Layer

4.3.2.1 Communication Network Interface

The Communication Network Interface (CNI) stores all incoming port data written by the Address Logic, all outgoing port data written by the host, and the respective port flags in the CNI memory. By supporting the information pull paradigm for input data on the host side and the information push paradigm for output data on the Address Logic side [DeL99], it acts as a temporal firewall [KO02] for the Network-on-a-Chip.

Furthermore, the CNI has to interface the different port data widths used by the Address Logic (fragment width, 128 bit) and the host interface (32 bit).

The following paragraphs describe the implementation of the standard CNI, other CNI implementations may differ due to additional Plug-Ins and/or optimized hardware resource usage (data word width, memory size, etc.).

To match the Address Logic timing requirements, the CNI memory features a 128 bit data word width, so write or read accesses by the Address Logic can be serviced in one or two clock cycles, respectively. As a consequence, host write accesses are handled by duplicating the write data three times and assigning the appropriate byte write enable lines. In case of a host read access, a multiplexer is used to select the requested 32 bit word from the 128 bit memory output, determined by the two lowest bits of the host address. Figure 4.23 provides a schematic overview of these operations.

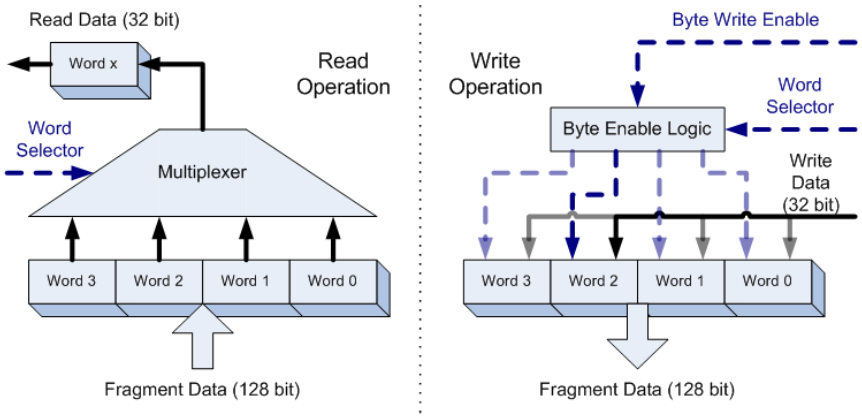

Figure 4.23: Data Word Width Reduction in the CNI

The CNI memory itself is implemeted by means of a 512x128 bit Dual-Port Memory to eliminate all concurrency problems arising from possible simultaneous host and Address Logic accesses.

The size of this memory was determined with respect to two requirements:

1. A core of the NoC should be able to perform at least one incoming and one outgoing Time-Triggered Ethernet [Ste06] state message.

   Such a message is 1500 byte (= 12000 bit) long, that equals 93.75, or, rounded up, 94 TTSoC fragments. Therefore, an appropriate input state port uses 94 fragments for the port data and one extra fragment for the port flags. An output state port needs twice the memory for the port data (one "standard" and one "shadow" register) and one fragment for the port flags too.

   Hence, the total memory usage to receive and send a Real-Time Ethernet state message is $(94 + 1) + (2 * 94 + 1) = 284$ fragments, which is the minimum memory size for this implementation.

2. The implementation of the TTSoC should be resource efficient, with respect to the limited on-chip resources of the used Field-Programmable Gate Array (FPGA), like logic elements (LEs) and memory.

   The used FPGA, an Altera Cyclone II [Alt07b], combines standard M4K memory blocks, which provide 4096 memory bits each and support a data width of up to 32 bit,[6] to provide any custom memory size and data width.

   As a consequence, when using this particular FPGA, memory resource usage can be calculated in M4K blocks instead of used memory bits.

   To build a 284x128 bit Dual-Port Memory, as requested by the first requirement, 16 M4K blocks are needed. Since a 512x128 bit Dual-Port Memory requires the same amount of M4K blocks while providing approximately 44% more storage capacity, a memory size of 512 fragments was chosen for this implementation.

The implemented memory is equipped with byte write enable signals, which are used by the CNI for data width reduction and to simplify the port flag synchronization mechanisms.

The CNI Layer provides two sets of byte write enables to the host: The standard byte write enable lines, which are driven by the host hardware to perform a selective byte write, and three extra byte write enable bits in the port address, set by the host software. Although the hardware byte write enable lines would be sufficient for normal host access operations, it can be a fairly complex task to write host software which is able to set them accordingly, i. e., machine code instructions or special source code constructs may be necessary, depending on the used host processor. Certain byte write enable settings are needed for port synchronization operations (i. e., setting or clearing the "valid" flag of a

[6] Or 4608 memory bits each and up to 36 bit data width including parity bits.

State or Streaming Port without changing the "using" flag, writing the Position Changed Flag (PCF) and the read or write position of an Event Port without changing the other one), therefore the CNI Layer offers the possibility to control byte write enables by means of three extra bits of the port address which can be set by the host software with reasonable programming effort. Since only the lower three bytes of a 32 bit host word can contain synchronization information, only three byte write enable bits are needed, the highest byte can only be masked out by means of the hardware byte write enable lines. Different hardware and software byte write enable signals concerning the same byte result in not writing this particular byte.

The Address Logic in the LLC Layer uses five byte write enable lines. The lowest three of them correspond to the lowest three bytes of the written fragment, similar to the software driven extra byte write enable bits provided to the host. The fourth byte write enable line controls the writing of bytes 7 down to 3 (bits 63 down to 24), while the highest 8 bytes (64 bits) of the fragment are affected by the fifth byte write enable line. These two byte write enable lines are used when writing timestamps.

The Plug-In (PI) Configuration Interface is not used by the standard CNI, hence it ignores any host read or write accesses concerning this interface.

4.3.2.2 Host Address Decoder

The Host Address Decoder is responsible for distribution of all host accesses to the addressed component or internal interface of the CNI Layer, i. e., the Data Interface, the Plug-In Configuration Interface, the Configuration Interface and the Interrupt Service. Furthermore, it houses the global real-time register and manages all OCP Host Interface functions, as mentioned in section 4.2.2.

Global Real-Time Register When the lower 32 bits of the global real-time register, which are directly connected to the chip-wide global real-time are read, the higher 32 bits are stored in a register to ensure consistent access of the real-time for the host.

4.3.2.3 Interrupt Service

Besides providing an Interrupt Status and an Interrupt Mask Register, as described in section 4.2.2.2.3, the Interrupt Service incorporates a timer interrupt. A timer interrupt is triggered every time the following equation is true:

(Timer Interrupt Time) == (Timer Interrupt Mask) & (Global Real-Time)

This mechanism allows for periodic timer interrupts as well as interrupts at a specific point in time.

All possible interrupt events may lead to an interrupt request (IRQ). Section 4.2.2.1 explains this mechanism in detail.

5 CNI Layer Extensions

The LLC Layer of the TISS and the CNI Layer provide the necessary basic functions to allow a host to communicate over the NoC. When designing the LLC and CNI Layers, we decided that only a minimum of additional features should be integrated in the LLC and CNI Layers since the TTSoC architecture should provide a generic architecture for a variety of different applications. In addition to simplifying a possible certification process, this approach benefits smaller applications, which do not need any other functions than simple event or state message communication and thus can take advantage of a compact design, regarding die area and power consumption.

By reducing the LLC and CNI Layers to their essential role, more complex applications are forced to implement supplementary functions by means of host software, which consumes processing power.

To bridge this gap we made the decision that it should be possible to improve the CNI Layer with custom hardware functions to support a wider range of application requirements.

During the design process, two possibilities to extend the functional range of the LLC and CNI Layers were identified:

- **Hardware Middleware Bricks**, located between the CNI Layer and the host (section 5.1), and
- Middleware **Plug-Ins**, implemented directly in the CNI (section 5.2).

5.1 Hardware Middleware Bricks

The concept of Hardware Middleware Bricks was strongly motivated by the possibility to easily compose different supplementary components, by physically "stacking" them to gain the requested functions.

Middleware, in the original meaning of the word, is low-level software located logically between an application program and the underlying hardware. Its purpose is to extend the capabilities of the hardware and/or provide a standardized application programming interface (API). Some examples of middleware are

communication protocol stacks, data security functions, or human interface device (HID) APIs.

Since hardware modules can be more efficient than software modules concerning processing time, chip area, and power consumption,[1] realizing some middleware functions as hardware modules can benefit an application significantly.

The first approach was to simply replace the middleware software with hardware modules located in the data path between the host processor and the CNI Layer. These modules, called Hardware Middleware Bricks, are not part of the CNI Layer, but connect to the Host Interface of the CNI Layer. On the host side, they incorporate their own host interface, similar to the CNI Layer Host Interface, but with an extended functional range. Figure 5.1 shows a schematic overview of this layout.

The advantage of the Hardware Middleware Bricks approach is, that this concept allows an arbitrary number of Hardware Middleware Bricks to be stacked between the CNI Layer and the host. Furthermore, since the Hardware Middleware Bricks can be designed independently from the LLC and CNI Layers, an already compiled NoC can be easily equipped with additional functions, without changing the original design. Even a physical separation of the NoC, the individual Hardware Middleware Bricks, and the host can be accomplished, by locating each part on its own microchip.

One of the disadvantages of this solution is the change of the temporal properties of the Host Interface imposed by every single Hardware Middleware Brick. Three sources of latency and jitter can be identified in a Hardware Middleware Brick:

Signal Delay is the time an electrical impulse needs to travel the distance between a source, like a host output pin, and a sink, like the data memory in the CNI. Since every wire and logic element adds to the signal delay, a Hardware Middleware Brick inherently prolongs the time a signal between the host and the CNI Layer needs to become valid. This may reduce the maximum clock frequency of the whole TTSoC or raise the need for extra clock cycles (waitstates) to wait for certain data to become valid.

Data Modifications are all actions a Hardware Middleware Brick performs on data transferred from the host to the CNI Layer or vice versa, to fulfill its

[1] Chip area and power consumption of software modules can be calculated by taking the extra processing power of a host processor needed by the additional software module into account. Providing this extra processing power without deteriorating the other host services can lead to an increase of the host processor's chip area and/or power consumption.

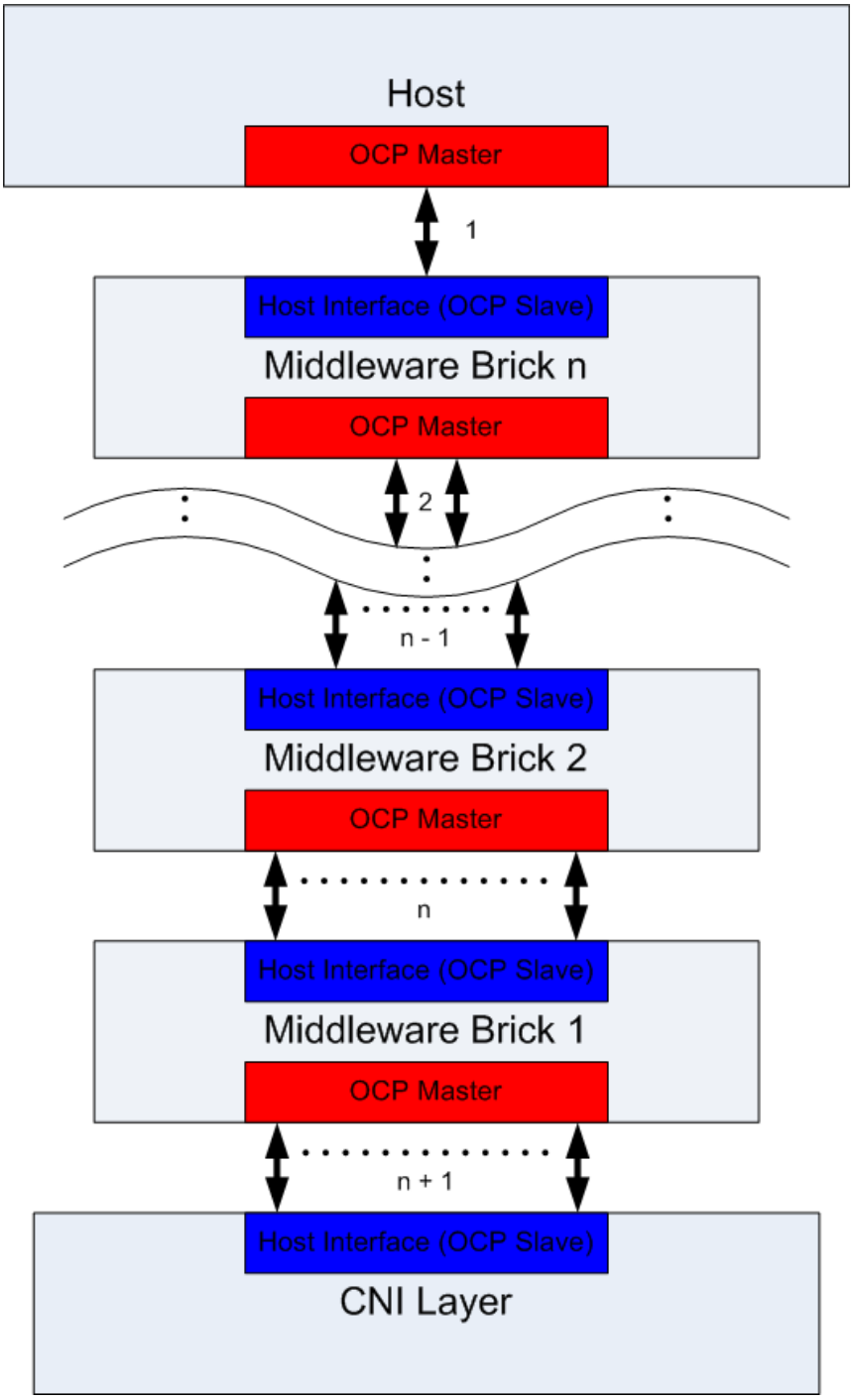

Figure 5.1: Stacking of Hardware Middleware Bricks

purpose. Examples of such operations would be port data encryption and decryption, done by a security middleware. Every such action slows down

the data transfer, by introducing additional logic elements in the data path, each of which adds to the accumulated signal delay. Furthermore, sequential logic may be needed to perform certain actions, causing a latency of some clock cycles, or, even worse, jitter may be introduced by a data-dependent behavior[2] of the operation to be performed.

Concurrency is introduced by Hardware Middleware Bricks which transfer data over the CNI Layer Host Interface by themselves, independently from any host access. An example would be a Hardware Voter, which reads the data of three ports from the CNI memory to compare them. Such a data transfer is initialized by the Hardware Voter itself as soon as all three ports are received completely. The concurrency problem arises if the host and the Hardware Middleware Brick try to access the CNI Layer simultaneously. Since the CNI Layer Host Interface only supports one read or write operation at a time, either the host or the Hardware Middleware Brick has to be delayed until the first transaction is completed. This access arbitration, besides consuming on-chip resources, leads to latency and/or jitter and has to be performed very carefully. Any deadlock [Sta01, p. 204] (e. g., a misbehaving host program, constantly reading from (i. e., blocking) the CNI memory while waiting for a Hardware Middleware Brick to write the requested data) or livelock situations [Sta01, p. 211] (e. g., a host and a Hardware Middleware Brick alternately reading and overwriting the same CNI memory location, waiting for a response) and deadline misses (e. g., watchdog lifesign, communication schedule) have to be avoided, which makes such an arbiter a fairly complex component.

Although the latency imposed by the Hardware Middleware Bricks and, up to a certain level, the jitter can be compensated by the host software, the host software has to be aware of the Hardware Middleware Bricks attached between the host and the CNI Layer, even if they are currently not in use. As a consequence, it is not possible to migrate host software between different cores of the TTSoC without changing it, or, at least, reevaluating its execution time and timely behavior in general. Furthermore, replica determinism cannot be guaranteed between cores equipped with different Hardware Middleware Bricks, cores using the same Hardware Middleware Bricks with different startup states, or cores fitted with Hardware Middleware Bricks using any indeterministic access arbitration.

To make matters worse, every additional Hardware Middleware Brick added to a specific core increases the problems mentioned above. In a core equipped

[2] Some operations may consume a varying amount of time, depending on the data they are performed on.

with n Hardware Middleware Bricks, $n+1$ instances (host, n Hardware Middleware Bricks) may try to access the CNI Layer concurrently. In addition, signal delay is significantly increased by multiple succeeding Hardware Middleware Bricks, while the latency and/or jitter caused by data modifications can be held nearly constant, as long as different Hardware Middleware Bricks modify different data words.

Considering these drawbacks of Hardware Middleware Bricks and the effort needed to overcome them, an implementation in the scope of this thesis was not eligible. Further development was suspended in favour for Middleware Plug-Ins (section 5.2).

Still, a future implementation of Hardware Middleware Bricks is possible. Some examples for functions which may be implemented as Hardware Middleware Bricks are listed below.

Encryption/Decryption A security middleware can encrypt port data written to a specific port and decrypt port data read from a specific port, respectively.

Independent Functions Functions providing a service not directly related to the operation of the Network-on-a-Chip can also be designed as Hardware Middleware Bricks. In this way, these functions could take advantage of the services provided by the NoC and the LLC and CNI Layers, e. g., the global real-time register, without disturbing normal host operation. An example of such a function would be a Hardware Middleware Brick providing additional timer interrupts.

Adapter In general, any kind of adapter between the OCP Host Interface of the CNI Layer and a host using another peripheral interface than the Open Core Protocol could be seen as some sort of Hardware Middleware Brick.

5.2 Middleware Plug-Ins

Middleware Plug-Ins are hardware modules extending the functional range of the LLC and CNI Layers. In contrast to Hardware Middleware Bricks, Plug-Ins are an integral part of the CNI Layer.

To add additional functions to a core of the Network-on-a-Chip, the standard CNI can be replaced by a CNI fitted with a Plug-In. These Plug-In-equipped CNIs can be customized completely to fulfill the intended purpose, even a CNI memory is not necessarily required. The only requirement for any CNI is to correspond to all logical and temporal constraints introduced by the three internal CNI interfaces, the 32 bit Host Data Interface, the Plug-In Configuration

Interface, and the 128 bit interface to the LLC Layer, as documented in section 4.3.2.1.

Although completely customized CNIs without a CNI memory can be designed, it is recommended to preserve a minimum of the standard functionality (i. e., at least a small CNI memory, capable of storing standard Streaming, State, or Event ports without interacting with the fitted Plug-In) since not all host software can interact with such specialized CNI Layers, making any future upgrades or host software changes difficult or even impossible. Therefore, only CNIs equipped with a CNI memory and an additional Plug-In, as seen in figure 5.2, are discussed in this document.

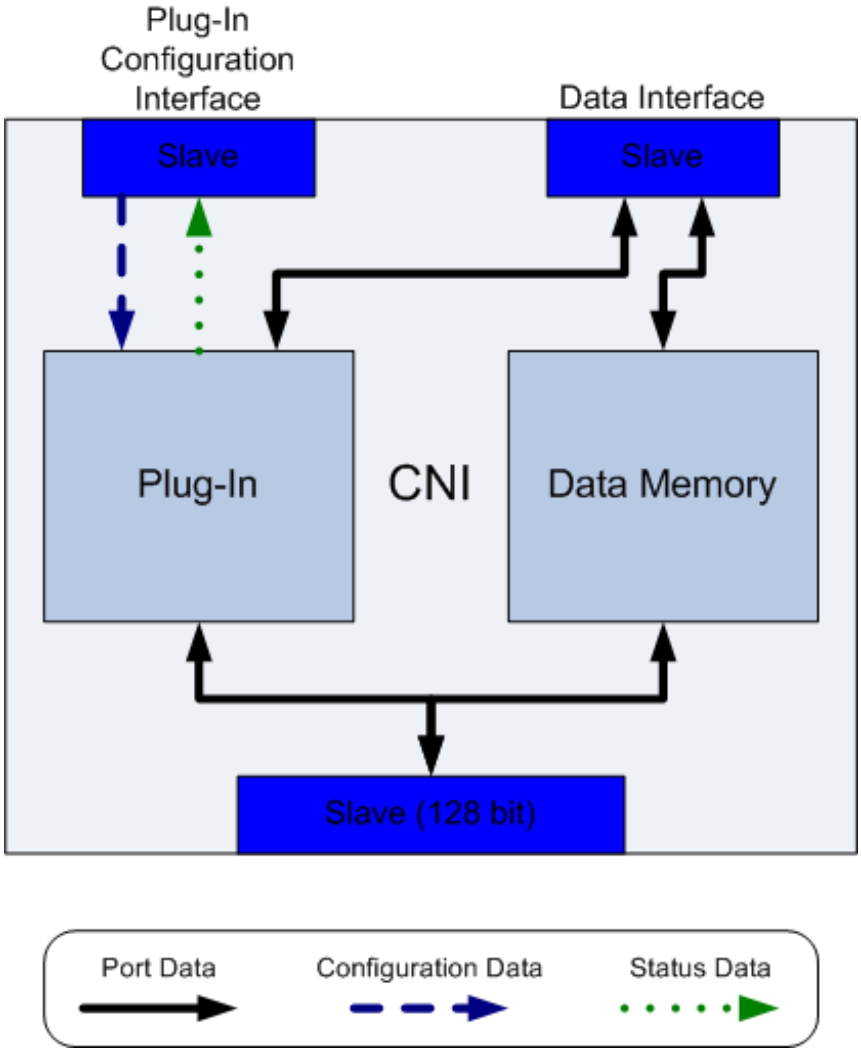

Figure 5.2: CNI fitted with a Plug-In

As part of a custom CNI, a Plug-In can be implemented between the host

and the CNI memory, between the LLC Layer and the CNI memory, or wrapped around the CNI memory. Even a Plug-In mounted parallel to the CNI memory, or multiple Plug-Ins in a single CNI (parallel to or succeeding one another) are possible.

By interfacing directly with the Address Logic in the LLC Layer, a Plug-In is constantly aware of incoming messages, since any received fragments are directly written to the Plug-In. On the other hand, fragments requested for transmission by the Address Logic are provided by the Plug-In instantly. Therefore, any data a Plug-In processes is as current as possible, as well as the data sent to other cores.

This implementation allows for a timely and predictable Plug-In behavior without interfering the operation of other components of the LLC and CNI Layers, in contrast to Hardware Middleware Bricks, which need an external source of information about received or sent messages (e. g., interrupts, timers), and have to read (or write) the port data over the CNI Layer Host Interface before (or after) performing any operations.

Another advantage of this solution is that Plug-Ins do not consume any resources concurrently with the host processor. When designed properly, a Plug-In does not affect the normal LLC Layer or CNI Layer operation at all. Most important, aside from minor changes in signal delay caused by the necessary routing functions of a custom CNI, no additional latency or jitter is introduced to the LLC and CNI Layers by a Plug-In.

Since the routing functions in a custom CNI, needed to distribute port data between the CNI memory and an attached Plug-In, can be designed to operate within the same latency boundaries as the memory access functions of a standard CNI, replica determinism between a core fitted with a Plug-In and a core using an unmodified CNI Layer can be maintained. In addition, by designing a Plug-In in a way that allows it to be disabled completely, a CNI Layer equipped with a Plug-In can be used by all kinds of host software originally intended to run on a core without a Plug-In, without any change. It is not necessary for the software to be aware of any installed Plug-Ins, as long as the standard CNI memory range is still accessible.

Custom CNIs equipped with one or multiple Plug-Ins can be designed a priori, with known temporal properties and resource consumption (die area, power, etc). As a consequence, a CNI Layer can be built by simply inserting a pre-designed CNI module housing a Plug-In which provides the requested functions.

Although recompiling (parts of) the system in case of a Plug-In-change is

inevitable, it is possible to fit a system with additional Plug-Ins not necessarily requested at compile time, by using Plug-Ins which can be disabled completely by the host software. Despite of requiring additional on-chip resources, this approach is promising, since no hardware upgrade becomes necessary in case of a change in host software. Using this technique, performant systems requiring limited maintenance effort can be provided with reasonable expenses.

The following sections describe some Plug-In examples. A Voter Plug-In (section 5.2.2) and a Direct Memory Access (DMA) Plug-In (5.2.1) were implemented in the scope of this thesis to demonstrate the concept of extending the LLC and CNI Layers with Middleware Plug-Ins.

5.2.1 Direct Memory Access Plug-In

The purpose of the Direct Memory Access (DMA) Plug-In is to transfer data fragments directly to or from the host main memory (or another appropriate host peripheral device) to a Streaming Port when received or requested for sending, respectively. These operations are performed without interrupting the normal host operation.

When set to input mode, the DMA Plug-In, once initialized, stores data received on the selected port directly to an a priori configured address in the host memory space. On output mode, after reading the first data fragment from the configured address in the host main memory, the DMA Plug-In reads the next data fragment as soon as the previous data fragment has been sent over the NoC, until the whole port is transmitted.

To gain access to the host main memory (or any other appropriate peripheral device), the DMA Plug-In implements a second OCP Host Interface, designed as an OCP master (see appendix A). The DMA Plug-In issues "Write" commands over this interface to store received port data, as well as "Read" commands to load requested data for transmission. No memory access arbitration or address checking operations are performed with respect to the variety of different host processors that can be used for the Time-Triggered System-on-a-Chip. In fact, not all host processors may be able to take advantage of the DMA Plug-In since not all host processors allow a peripheral communication device like the NoC to directly access their main memory.

5.2.1.1 Implementation

This implementation of a DMA Plug-In was tested using the Altera NIOS II [Alt07c] as host processor and the Avalon Master Adapter (appendix B.3)

to translate the DMA "Read" and "Write" commands to regular Avalon Memory-Mapped interface accesses.

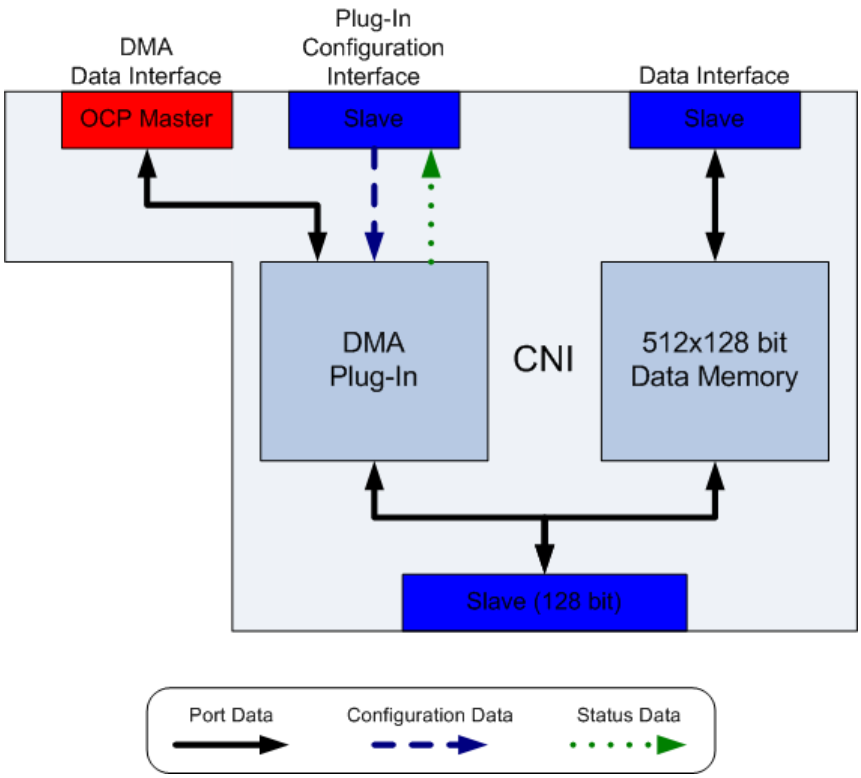

Figure 5.3: CNI fitted with a Direct Memory Access (DMA) Plug-In

The DMA Plug-In is integrated into a custom CNI, fitted with the required extra DMA OCP master interface, parallel to a standard size 512x128 bit CNI memory (figure 5.3). This implementation allows the host software to use the full range of port data memory, like on a CNI Layer with a standard CNI, if the DMA Plug-In is disabled. If enabled, the DMA Plug-In occupies the highest four fragments of the port data memory (fragment addresses 508 to 511) for its operation. The host is still able to read data from these addresses, but this data has to be considered invalid while the DMA Plug-In is enabled.

The Streaming Port to be used for DMA transfers has to be configured to use *PORT_BASE_ADDRESS* 508. Table 5.1 summarizes the memory usage of the DMA Plug-In CNI.

By configuring a Streaming Port to use *PORT_BASE_ADDRESS* 508, the host software allows the DMA Plug-In to read from or write to this particular Streaming Port. A different configuration of the port used by the

Memory Address	Usage
DMA Plug-In disabled	
0-511	normal port data
DMA Plug-In enabled	
0-507	normal port data
508	DMA port port flags
509	fragment data register
510	input: reserved
	output: fragment data shadow register
511	reserved

Table 5.1: DMA Plug-In Memory Usage (Fragment Addresses)

DMA Plug-In (e. g., wrong *PORT_BASE_ADDRESS*, wrong port type, etc.) will most likely lead to an access violation (e. g., writing to a reserved address) by the Address Logic, resulting in a Configuration Error indicated by the appropriate flag in the DMA Plug-In register file and a Plug-In Error recorded by the Error Status Register of the LLC Layer.

The DMA OCP master interface is implemented as a 32 bit data width interface using 32 bit wide byte addresses and no sideband signals, like interrupts.

The DMA Plug-In awaits the OCP slave on the host side to accept a "Write" command by setting the *SCmdAccept* signal. If a write operation is delayed too long by the host (e. g., due to memory access arbitration), the DMA Plug-In may not be able to store a whole fragment (128 bit, so four 32 bit write accesses are needed) before the next fragment is received by the Address Logic. This situation forces the DMA Plug-In to disable itself and to signal a DMA Busy Error. The received data is lost in that case.

When issueing a "Read" command, the OCP slave has to provide the requested data on the *SData* lines along with an OCP *SResp* signal set to "Data Valid" similar to a "Write" command. Again, failure to do so, or a "Response Error" or "Request Failed" signal on the *SResp* lines leads to a shutdown of the DMA Plug-In and a Host Memory Error or DMA Busy Error signal, respectively. Unless this problem is taken care of by the host before the next send slot for a fragment of the DMA port is due, the Address Logic reads the next fragment of the DMA port from the CNI memory instead of the DMA Plug-In, rendering the sent data invalid.

A Finite State Machine (FSM) with 12 states is used to perform all data transfer functions of the DMA Plug-In. The states of this FSM, shown in

figure 5.4, are explained in detail in the paragraphs below.

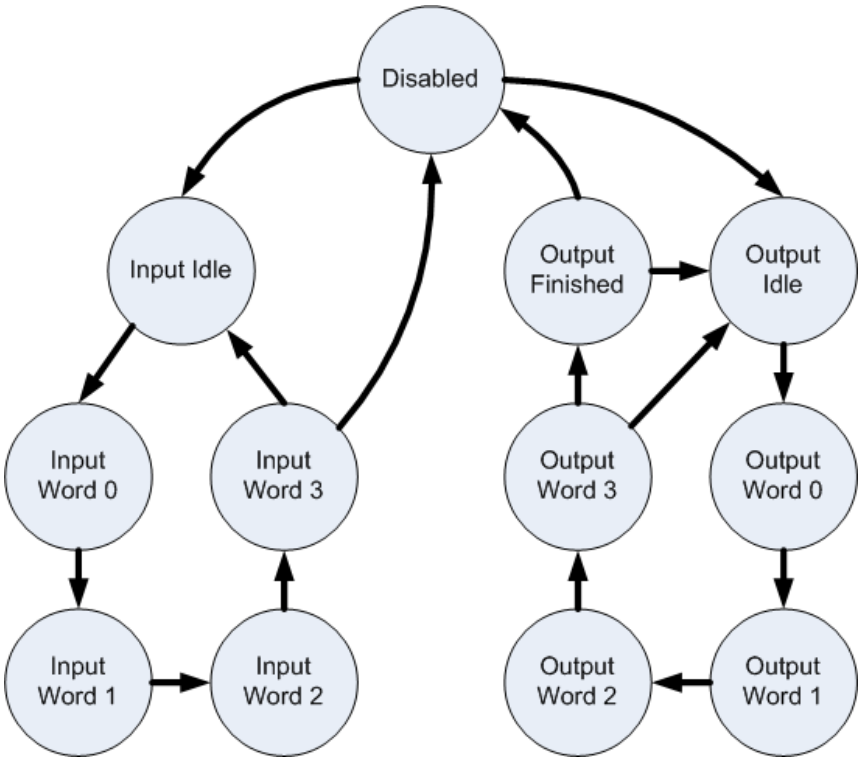

Figure 5.4: DMA Plug-In Finite State Machine

Disabled State The DMA Plug-In performs no action while in this state. Once enabled by the host, it stores the start address configured by the host in a current address register and transitions to either Input or Output Idle State, depending on the configured port direction.

Input Idle State The DMA Plug-In waits for the next fragment of the Streaming Port to be received. When the DMA Plug-In was just enabled or the last fragment stored was the last fragment of the port, the DMA Plug-In waits for the first fragment of the Streaming Port, determined by the Fragment Position written to the port flags by the Address Logic. This is done to prevent port data corruption in the host main memory caused by enabling the DMA Plug-In before the pulsed data stream belonging to the DMA port is completed, or by losing a fragment due to errors of the sending core or the NoC. Not synchronizing to the first fragment of a pulsed data stream in such situations could result in disarranged fragments. It would not be possible to determine the start

and the end of a pulsed data stream later, rendering any received port data, present and future, useless.

After receiving and storing the next fragment to a register, Input Word State 0 is enterend.

Input Word State 0-3 The four Input Word States are entered one after the other. In every Input Word State, the next 4 bytes (32 bit) of the received fragment are stored to the current main memory address, which is incremented by 4 afterwards.

In Input Word State 3, after writing the last word of the current fragment, the size of the port is checked to determine if the fragment just written was the last fragment of the pulsed data stream. If the pulsed data stream is completed, the DMA Plug-In can trigger an interrupt (if enabled) and enters either the Disabled State or the Input Idle State, depending on the Repeat Flag set by the host. The current address register is reset to the start address and the Complete Flag in the first DMA Plug-In register is set to indicate the successful completion of the DMA transfer.

If there is still port data to be received, the Input Idle State is entered and the DMA Plug-In waits for the next fragment of the pulsed data stream.

Output Idle State The Output Idle State waits for the "using" and "valid" flags of the Streaming Port port flags to be equal before starting a fragment fetch process. This condition indicates that the Address Logic has commenced sending of the current fragment stored in either the "standard" or in the "shadow register", so the other register is free to be filled with the next fragment from memory. Output Word State 3 inverts the "valid" flag after fetching the next complete fragment to force the FSM to wait for transmission again. See section 4.2.2.2.1 for Streaming Port synchronization details.

Once enabled, the DMA Plug-In immediately starts to fetch the first fragment at the start address in the host main memory, to be ready for the first send request of the Address Logic. Still, the actual sending of the first fragment is prolonged until the beginning of a pulsed data stream to prevent disarranged port data on the NoC, similar to the mechanism used in the Input Idle State.

The next state of the DMA Plug-In FSM is Output Word State 0.

Output Word State 0-3 While in these states, the DMA Plug-In reads the next fragment to transfer word by word from the host main memory. The current address is increased by 4 after every word read.

In Output Word State 3, the "valid" flag of the Streaming Port port flags is inverted to signal that a complete fragment is ready for transfer. If it is the last fragment of the pulsed data stream, Output Finished State is entered, otherwise the FSM transitions back to Output Idle State to wait for the Address Logic to transfer the fragment and the opportunity to fetch the next fragment.

Output Finished State This state is needed to stall the DMA Plug-In until the last fragment of a pulsed data stream is actually sent by the Address Logic. Once done, the Complete Flag is set, an interrupt is triggered if enabled, and the current address is reset to the start address. Depending on the Repeat Flag, the DMA Plug-In is either disabled or the FSM enters Output Idle State to wait for and perform the next round of the pulsed data stream.

All states mentioned above return to Disabled State immediately, aborting any operation in progress, if the host disables the DMA Plug-In.

5.2.1.2 Direct Memory Access Plug-In Register File

The DMA Plug-In provides two 32 bit configuration and status registers (figure 5.5), located at register addresses 0 ("0000") and 1 ("0001") of Plug-In address 2 ("0010").

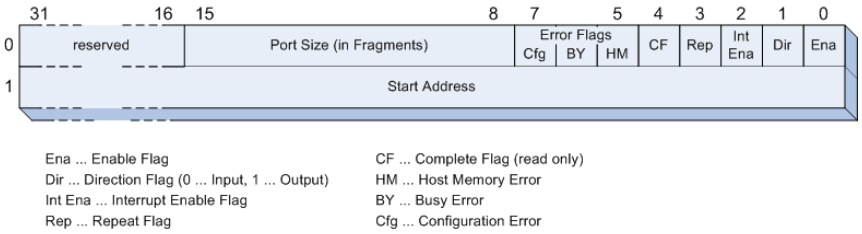

Figure 5.5: DMA Plug-In Register File

The following configuration and status flags are implemented:

Direct Memory Access Register 0

Enable Flag This flag is used by the host to enable or disable the DMA Plug-In. Once set, every change of the configuration registers by the host is prohibited by the DMA Plug-In, with the exception of clearing the Error Status Flags. Therefore, care has to be taken if the byte write enable lines of the CNI Layer are used. The host has to ensure that all relevant configuration data is already written before the Enable Flag is set.

The DMA Plug-In disables itself by clearing the Enable Flag if a DMA transfer is completed and the Repeat Flag is not set, or if any error occurs.

Direction Flag The DMA Plug-In can be set to input mode (`0`) or output mode (`1`) using this flag.

Interrupt Enable Flag After completing a DMA transfer by storing or sending the last fragment of a pulsed data stream, an interrupt can be triggered if enabled by this flag. The host can determine the current interrupt status by examining the Plug-In Interrupt flag in the Interrupt Status Register of the CNI Layer (section 4.2.2.2.3).

Repeat Flag The host can set this flag to force the DMA Plug-In to start a new DMA transfer as soon as the previous transfer is completed, using the same settings (start address, port size, etc.) as before.

Complete Flag The read-only Complete Flag is set after a whole pulsed data stream has been transferred by the DMA Plug-In, and cleared at the beginning of a DMA transfer, i. e., when the DMA Plug-In FSM is in Input or Output Idle State.

Error Status Flags These flags are set on DMA Plug-In errors. They can be cleared at any time, even if the DMA Plug-In is enabled, by writing `1` to them. Every time one of these flags is set due to an error, the Plug-In Error Status Flag in the LLC Layer Error Status Register (section 4.3.1.4) is set too.

Host Memory Error Flag Set if the host main memory reacts to a "Read" command of the DMA OCP master interface with a "Response Error" or "Request Failed" signal on the *SResp* lines.

Busy Error Flag This flag is set if the Address Logic tries to write or receive a fragment while storing or loading of the previous fragment to or from host main memory is still in progress (i. e., the DMA Plug-In is busy).

Configuration Error Flag If the Address Logic tries to access one of the addresses occupied by the DMA Plug-In in a not intended way (e. g., writing to a reserved address) due to erroneous port configuration, this flag is set.

Port Size The port size in fragments has to be configured by the host before enabling the DMA Plug-In. Since the DMA Plug-In counts the transferred fragments and stops once the port size configured herein is reached, the port size stored in this register has to match the port size set in the Port Configuration Memory (section 4.2.2.2.4) to prevent data corruption.

Direct Memory Access Register 1 This register is used for the 32 bit start address for every DMA transfer. This byte address is incremented by 4 after every 32 bit word transfer by the DMA Plug-In, until the whole pulsed data stream is completely transmitted or received.

The DMA Plug-In is restricted to operate on consecutive byte addresses in the host main memory, an address wrap-around, static addresses, or non-consecutive address regions are not supported. The host software is responsible for providing the send data or the memory space for data to be received in an appropriate form, to prevent the DMA Plug-In from sending invalid data or overwriting restricted memory regions.

5.2.2 Voter Plug-In

To detect and tolerate single-core failures in a dependable system, Triple Modular Redundancy (TMR) can be implemented. When using Triple Modular Redundancy, a core is replicated and a voter is used to perform a majority vote over the results of the replicated cores. Thus, the intended service can be provided even in case of a failure of one of the replicated cores. Such a voter was implemented in the scope of this thesis in the form of the CNI Layer Voter Plug-In introduced in this section.

A voter round, seen at TMR system level, is performed in four stages (figure 5.6):

1. The three replicated cores receive their input data.
2. The cores perform the desired operation on the input data.
3. The operation results of all three cores are transmitted to the voter.
4. The voter compares the results and determines the correct result along with the current status of the three replicated cores, to allow possible fault correction operations (e. g., core reset, maintenance notification, etc.).

The Voter Plug-In takes advantage of the time-triggered nature of the TTSoC architecture, which allows the voter to determine exact time instances at which the input data from the three replicated cores should be valid and equal. Therefore, Voter Plug-In operation is organized in time slices, starting with the first state data sent by one of the replicated cores and lasting until the correct result is provided by the voter. Such a time slice is called a **voting round**. Since voting is performed only on state data, a Voter Plug-In voting round consists of receiving three State Ports of equal length from the three replicated cores, determining the correct voting result, and forwarding this result and the current State Port status (valid data/corrupted data) to the attached host.

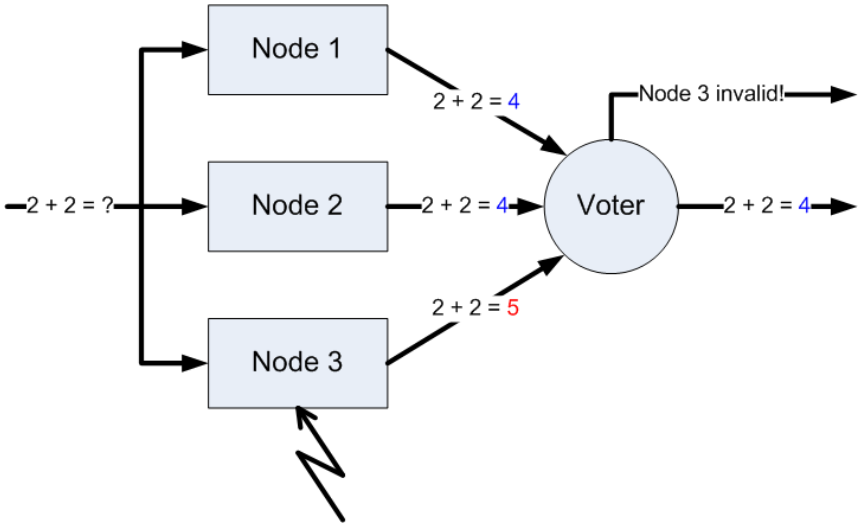

Figure 5.6: Triple Modular Redundancy (TMR) System with Voter

The following paragraphs are focused on the voter itself, replication of cores, scheduling issues, or the constraints the host software in the replicated cores has to fulfill are not discussed in this document.

Different possibilities to improve the NoC with a hardware voter were discussed during the design phase of the TTSoC project. A voter implemented as a Hardware Middleware Brick, performing all voting operations outside of the CNI Layer, introduces some serious drawbacks which have to be considered:

Resource Usage Since the voter has to wait for all three ports to be received completely, some sort of signal is needed to inform the voter of this condition. This signal can be an interrupt triggered upon completion of the last of the three ports or the voter itself may implement a timer service to wait for a calculated time instant after complete reception. In any case, additional resources (e. g., timer service) are needed or existing resources (e. g., port interrupts) are occupied by a Hardware Middleware Brick voter.

Bandwidth Consumption A Hardware Middleware Brick voter has to load the port data of all three ports, one data word after the other, from CNI memory over the CNI Layer Host Interface once reception of all three ports is complete, thus increasing the amount of data transported over the CNI Layer Host Interface and reducing the bandwidth available for other accesses.

Voting Delay Before the host processor can access the valid port data, the actual voting has to be performed, introducing a significant delay. After

receiving and loading the port data of all three ports, the voter subsequently compares the data to verify equality of the ports. Finally, if at least two ports were equal, the valid port data and the port status are stored in internal memory (further increasing the delay) or the CNI memory (increasing the delay and the bandwidth consumption) to be read by the host.

Scheduling A Hardware Middleware Brick voter seriously complicates message scheduling by requesting the three ports to be transmitted in order (to determine the complete reception of all three ports) and with a considerable delay between the completion of the last port and the start of the first port (to perform the actual voting and the storing operation).

Hardware Middleware Brick Voter operation is shown in figure 5.7 (left side).

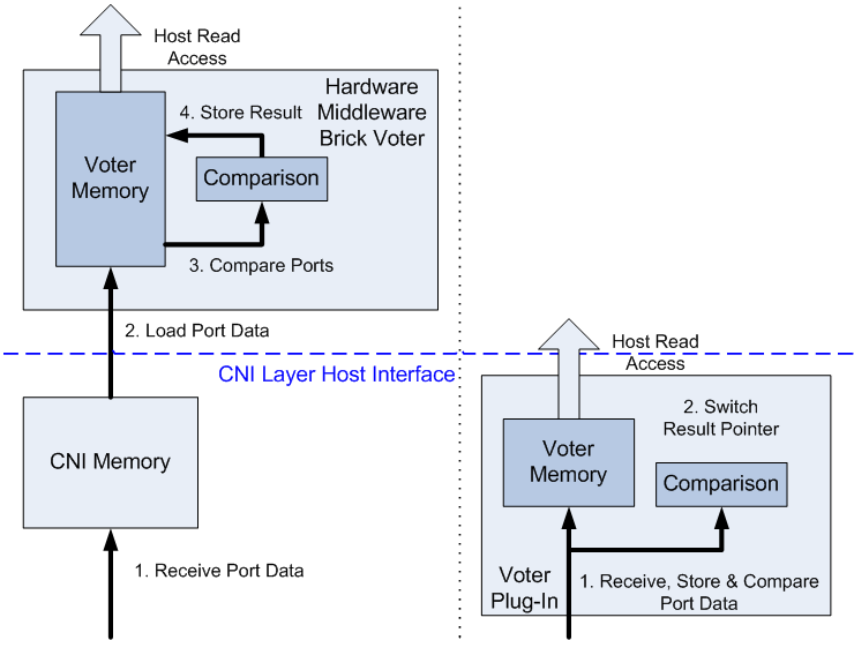

Figure 5.7: Comparison: Hardware Middleware Brick Voter / Voter Plug-In

By taking advantage of the Middleware Plug-In concept, which allows a Plug-In to immediately process received data without the need of any additional memory read operation, the Voter Plug-In introduced herein eliminates any additional CNI Layer Host Interface bandwidth usage and reduces the latency generated by the voting process to a minimum (right side of figure 5.7).

The Voter Plug-In replaces half of the CNI memory by its own, internal memory of the same size. When the Voter Plug-In is disabled, this memory

can be accessed by the Address Logic and the host like normal CNI memory. As a consequence, the custom CNI equipped with a Voter Plug-In behaves the same way as a standard CNI whenever the Voter Plug-In is disabled. Hence, the host software is not needed to be aware of the Voter Plug-In if not using it.

If enabled, the internal voter memory is reserved for the three ports to be compared and the result port, which presents the valid data to the host. Each of these ports is accessed through a pointer and occupies a quarter of the available memory. The Voter Plug-In performs any comparing and storing operations on every single fragment as soon as it is received. Voting operations are not delayed until complete reception of the three ports.

Three counters, one for every port, are maintained by the Voter Plug-In. Once a fragment is received by the Address Logic and stored to one of the three port memory regions, the Voter Plug-In increases the counter belonging to that port.

If the counter of the just received port is lower than the counter of one of the other ports, the just received fragment is compared to the fragment at the same fragment position of the other port, which already contains valid port data, and a status register is set if the two compared fragments are not equal.

If the counter of the just received port is higher than the counter of one of the other ports, the fragment of the other port at the fragment position in question was not received by now. No comparison is performed with that port.

Figure 5.8 shows this mechanism.

Once all ports are completely received, i. e., after the last fragment of the voting round has been received, indicated by all three counters equaling the port size, all three ports have been compared fragment by fragment and the status register can be used to identify a possible erroneous port. The host has access to a copy of this status register, as well as to the valid port data, which is located in the result port. To provide the data to the host, the Voter Plug-In does not perform any copy operations, it just exchanges the result port pointer with one of the pointers to a valid port.

Before a new voting round starts, the Voter Plug-In resets all three counters to zero and clears the internal status register. If the port pointers were changed during the previous round, the port data is stored in different physical memorys in the next round, but still accessed through the same memory addresses by the LLC Layer and the host.

By using this technique, the Voter Plug-In is able to provide valid port data after a very short latency (< 10 clock cycles after receiving the last fragment of

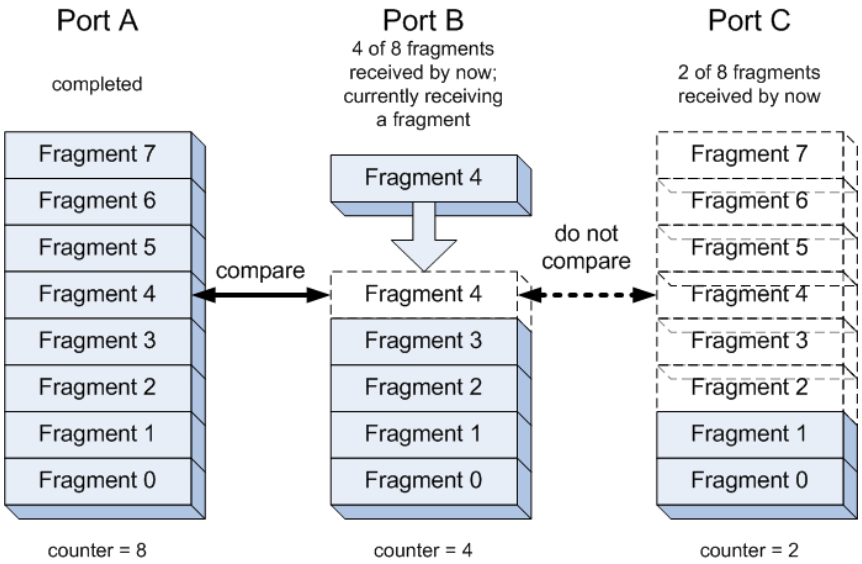

Figure 5.8: Voter Plug-In Fragment Comparison

the voting round) and to keep this data stable until the next voting operation is completed successfully. After an unsuccessful voting attempt (i. e., all three ports are different), indicated by the status register, the result port pointer is not altered, leaving the contents of the result port unchanged.

In contrast to a standard software or Hardware Middleware Brick voter, the Voter Plug-In operates without knowing the sequence of the ports to be transmitted in every voting round, since round completion is not determined by waiting for reception of the last fragment of the last port of the round, but by counting the received fragments of all ports. Therefore, "mixing" the fragments of different ports by transmitting them in interlaced pulsed data streams (i. e., fragment 0 of port A, fragment 1 of port A, fragment 0 of port B, fragment 2 of port A, ...) is permitted. Even a change of the schedule can be tolerated without reconfiguration or interruption of the Voter Plug-In, as long as voting rounds remain separated (i. e., one voting round is completed before the next voting round starts), and the internal fragment order of every port is preserved (i. e., fragment position 0, 1, ..., N). No further scheduling constraints are required.

5.2.2.1 Implementation

The Voter Plug-In is integrated into a custom CNI and provides memory space for four limited size ports, if enabled. The three ports to be voted upon are called ports A, B & C, the fourth port, used to present the valid data to the host,

is called result port. This fourth port is set up automatically when enabling the Voter Plug-In, no port configuration or port flags clearing is necessary. To perform the voting operations, the A, B & C ports have to be configured as input State Ports with specific base addresses, their port flags are cleared by the Voter Plug-In when enabled. Wrong base addresses or other ports configured to use the Voter Plug-In address space will lead to unpredictable behavior of the Voter Plug-In, resulting in errors and/or corrupted or lost port data. If the Voter Plug-In is disabled, the internal Voter Plug-In memory can be used like standard CNI memory, without any further restrictions. Table 5.2 summarizes the memory space of the implemented custom CNI for the Voter Plug-In.

Memory Address	**Usage**
Voter Plug-In disabled	
0-511	normal port data
Voter Plug-In enabled	
0-255	normal port data
256	port A base address
257-319	port A data
320	port B base address
321-383	port B data
384	port C base address
385-447	port C data
448	result port base address
449-511	result port data

Table 5.2: Voter Plug-In Memory Usage (Fragment Addresses)

The port size of the ports to be compared is limited to 63 fragments to preserve on-chip resources. Even if the three ports are smaller, their base addresses and memory consumption remain constant. This drawback is necessary to allow the Voter Plug-In to store every port in its own physical memory and, as a consequence, read a fragment at a specific fragment position from all ports concurrently. Comparison of a just received fragment with the fragments at the same fragment positions in the other ports can be performed simultaneously in this way.

The Voter Plug-In, as depicted in figure 5.9, can be divided into three components, which are discussed in detail in the following paragraphs: The Voter Plug-In Main Entity, the Voter Plug-In Memory, and the Voter Plug-In Comparator.

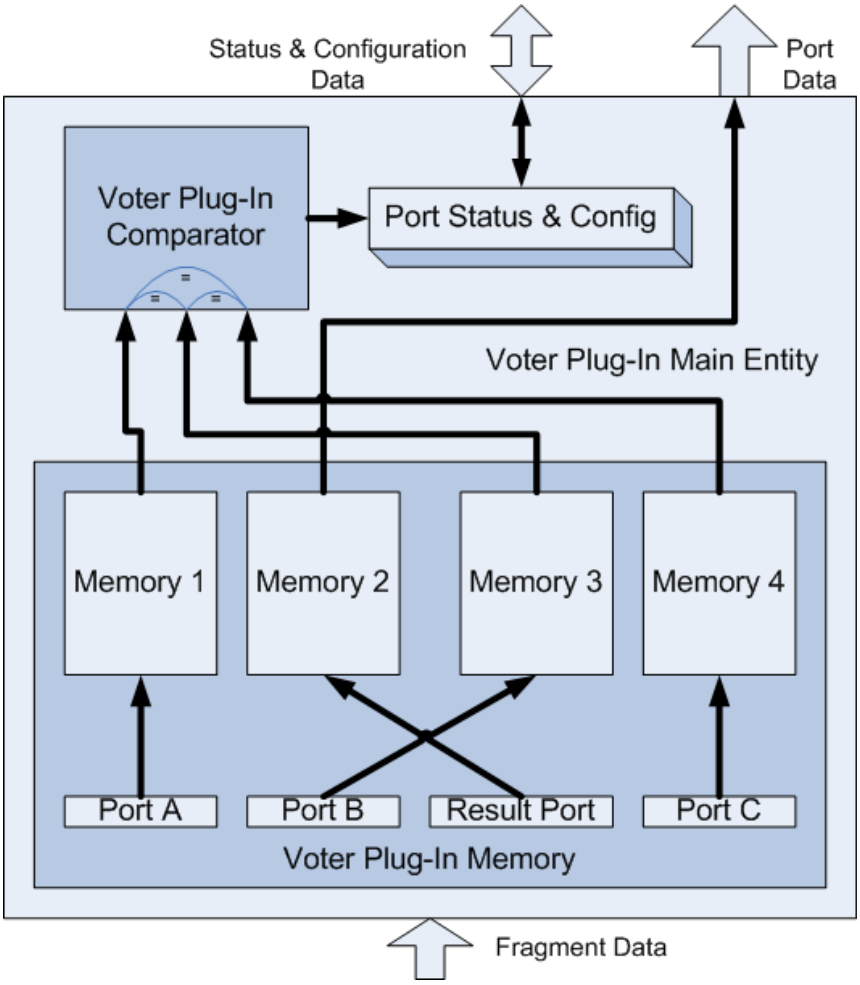

Figure 5.9: Voter Plug-In Overview

Voter Plug-In Main Entity The Voter Plug-In Main Entity connects the other two components of the Voter Plug-In, houses the configuration & status register, stores and alters the result port port flags (section 5.2.2.2), and is responsible for keeping the different memory pointers for the four ports.

Once a voting round is finished, the Voter Plug-In Main Entity resets the Comparator, and the memory pointers are exchanged to provide the new port data in the result port. Since no copy operations take place, the data of one of the input ports is invalidated during this process. The decision which input port is exchanged and invalidated is taken depending on the voting results, as shown in table 5.3.

Status	Pointer Exchange	Invalid
A = B = C	port B ↔ result port	port B
A = B ≠ C	port B ↔ result port	ports B & C
A ≠ B = C	port B ↔ result port	ports A & B
A = C ≠ B	port A ↔ result port	ports A & B
A ≠ B ≠ C	no exchange	ports A, B & C; result port may be outdated

Table 5.3: Pointer Exchange due to Voting Results

Voter Plug-In Memory All port data is stored in the Voter Plug-In Memory component. It houses four 256x32 bit Dual-Port Memories, which equals half of the standard CNI memory size. This memory setting was chosen to meet the following three requirements:

1. The integrated Voter Plug-In memory should be able to act as a standard port data memory while the voter is disabled, to preserve on-chip memory resources. Furthermore, a host software not aware of the Voter Plug-In should be able to use the CNI memory without any restrictions.
2. The Voter Plug-In implementation should be resource efficient, the size of the used on-chip memory has to be a tradeoff between usable port size (for the voting) and additionally required memory blocks.
3. The Voter Plug-In requires a separate physical memory for every port to function correctly, i. e., four independently accessible Dual-Port Memories are needed (ports A, B & C and the result port).

To maintain a feasible resource usage, the maximum voteable port size had to be reduced to approximately a quarter of the standard maximum port size of 256 fragments. A voter allowing ports of length 256 to be voted would need memory space for $4 * 256 = 1024$ fragments (or 131072 bits), in addition to at least a small standard CNI port data memory for other ports. By reducing the supported port size to 63 fragments (+1 fragment for the port flags), the Voter Plug-In requires $4 * 64 = 256$ fragments (or 32768 bits) of on-chip memory, exactly half of the standard CNI memory size.

An Altera Cyclone II FPGA was used during the implementation, which provides M4K memory blocks to assemble any custom memory size (see [Alt07b] and section 4.3.2.1). Since any Dual-Port Memory which is able to store 256 or less 128 bit words requires 8 M4K memory blocks, a word width of 32 bits was choosen, while the total amount of 64 fragments per port was kept, resulting in a 256x32 bit Dual-Port Memory using 2 M4K memory blocks for every port. In this way, $4 * 2 = 8$ M4K memory

blocks are required for the Voter Plug-In and 8 M4K memory blocks are used by the 256x128 bit CNI port data Dual-Port Memory, resulting in a total of 16 M4K memory blocks, the same amount as used by the standard CNI.

When the Voter Plug-In is disabled, the four 32 bit wide memories are accessed parallel to allow the Address Logic to write and read 128 bit fragments in the same amount of time a standard CNI memory operation would take. Every 32 bit memory stores a quarter of a fragment at the same address, emulating the standard 128 bit word width memory.

While not operating, the Voter Plug-In handles host accesses exactly the same way a standard CNI memory does.

No read operations are performed by the Address Logic once the Voter Plug-In is enabled, since the voter operates only on input State Ports. Any fragment written by the Address Logic is kept in a register and stored to the appropriate port memory, determined by the fragment address, in four consecutive clock cycles. The Address Logic needs seven clock cycles to finish a receive operation, therefore the Voter Plug-In completes serializing the received data before the next fragment can be received.

At the beginning and when finishing a State Port receive operation, the Address Logic updates the port flags. This is done while the Voter Plug-In is still busy serializing and storing the actual fragment data, so all three empty flags and sequencers (one set for every one of the three input ports) are stored in registers instead of the memories, allowing port flag read or update operations without interrupting the voting process.

While storing the received port data to the appropriate port memory, the 32 bit words at the same positions in the other two ports are loaded, and all three 32 bit words are transferred to the Comparator component to perform the actual voting. In addition, the Voter Plug-In Memory component determines the beginning of a port (fragment 0) by monitoring the write addresses and signals this event to the Comparator for synchronization purposes.

The host can read all stored data, the three input ports, the result port, and their portflags, at any time while the Voter Plug-In is enabled, but host write accesses are ignored.

Voter Plug-In Comparator The Voter Plug-In Comparator component performs the actual port data comparison. It maintains three word counters, one for every input State Port, to keep track of the already received amount of data of every port, three status flags to store the relationships between the ports

(equal/not equal), and three finished flags to detect the completion of a voting round.

Whenever a 32 bit word of port data is stored to the Voter Plug-In memory, the counter of the port it belongs to is compared to the other two counters, to determine if a status flag update is due. If the counter value of the just written port is smaller than one of the others, the written port data is compared to the already stored data of this other port (or to the data of both other ports, if the counter value is smaller than both other counter values). Otherwise, no comparison or status update takes place, since the data word of the other port (or ports) at the position to be compared was not received by now. The word counter belonging to the just received port is incremented afterwards in any case.

After a comparison, the appropriate port relationship status flag is updated according to the comparison result. These port relationship status flags are all set to "equal" on every Comparator restart, i. e., at the beginning of a voting round. Once a flag is set to "not equal" due to different port data, it cannot be reverted to "equal" before a Comparator restart is performed, even if the port data received later is equal again.

The value configured in the port size register (see section 5.2.2.2) defines the size of all three input ports, therefore completion of a port can be detected by examining the three word counters. If all three ports are finished, the communication round is completed, which is signaled to the other components of the Voter Plug-In, and the port relationship status flags can be considered valid. The Voter Plug-In Main Entity uses these port relationship status flags to update the port status flags and the different memory pointers, and finally initiates a Comparator restart to await the next voting round.

Aside from comparing the port data, the Comparator has to monitor two other possible cases of communication errors:

- Port data could be lost during the transmission, due to an erroneous sending core or because the sending core failed completely. This would make it impossible to detect the end of a voting round, since the counter of the port which the lost data belongs to could never reach the configured port size. The Comparator detects such a situation by monitoring the port start signal issued by the Voter Plug-In Memory component on the reception of fragment 0 of a port, and checks if the word counter value of the belonging port is zero. No Comparator restart has taken place if the word counter holds any other value than zero on reception of the first fragment of a port, therefore port data was lost in the previous voting round. As a consequence, the Comparator immediately restarts itself, clears all counters, adjusts the port relationship status flags to consider the incomplete

port, signals the end of the previous voting round to the Voter Plug-In Main Entity, and starts processing of the new communication round.

- The Address Logic can receive a fragment belonging to a port already finished in the current voting round. This error is most likely due to a faulty port size configuration, and is signaled to the host by setting the error flag in the Voter Plug-In configuration & status register and the Plug-In error flag in the LLC Layer Error Status Register.

5.2.2.2 Voter Plug-In Register File

The Voter Plug-In uses a single 32 bit register for voter configuration, port status information, and an error flag. It can be read or written by the host software at Plug-In address 1 ("0001"), register address 0 ("0000"). The Voter Plug-In register is shown in figure 5.10, the implemented flags are described below.

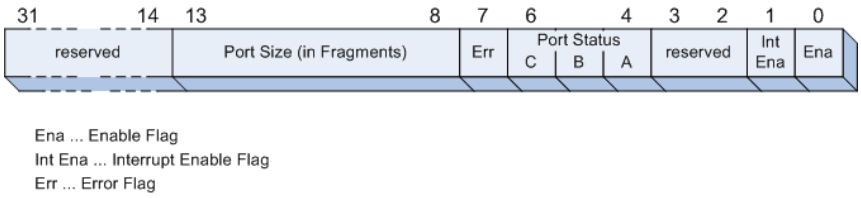

Figure 5.10: Voter Plug-In Register

Enable Flag The host enables the Voter Plug-In by setting this flag. When enabled, the Voter Plug-In prohibits any changes of the configuration flags, with the exceptions of clearing the error flag and disabling the Voter Plug-In. Therefore, the byte write enable lines of the CNI Layer Host Interface have to be set in the correct order to prevent a configuration data loss. When the Voter Plug-In is disabled, the port status flags and the error flag are not valid and should be ignored by the host. The interrupt enable flag and the port size configuration have no effects on a disabled Voter Plug-In.

The host software is responsible to enable the Voter Plug-In only in between voting rounds. Although the Voter Plug-In is able to synchronize itself to the start of a voting round by monitoring the completion of all three ports, it has no possibility to allocate the port data to the corresponding rounds. The Voter Plug-In declares the start of a voting round if all three ports are completed before one of them starts again (reception of fragment 0). Therefore, if the Voter Plug-In is enabled *during* a voting round *after* one or two ports are already completed, it may try

(and succeed) to synchronize itself to a wrong round start *inside* of an actual voting round, with the result of comparing port data from different voting rounds (figure 5.11).

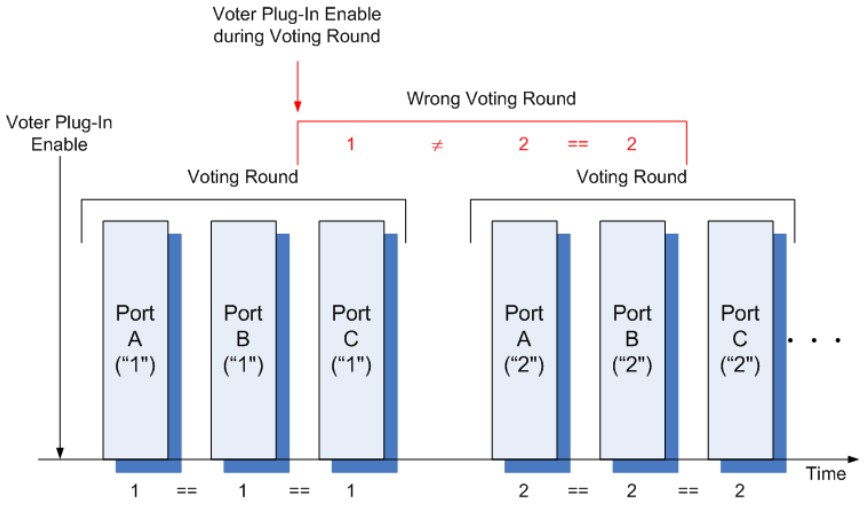

Figure 5.11: Voter Plug-In Voting Rounds

Interrupt Enable Flag By setting this flag, the Voter Plug-In is enabled to trigger a Plug-In interrupt in the Interrupt Status Register (section 4.2.2.2.3) whenever a communication round is completed, regardless of the results of the voting.

Port Status Flags These flags signal the status of the three input ports after a completed voting round. They are valid until the next round is finished. A set flag indicates that the corresponding port received valid data. Since a port is considered valid if its data is equal to another port, at least two of the three flags are set in this case. Table 5.4 lists all possible voting results.

PSF	Explanation
000	all three ports contain different data
011	ports A & B are equal, port C is different
110	ports B & C are equal, port A is different
101	ports A & C are equal, port B is different
111	all three ports are equal

Table 5.4: Voting Results (PSF . . . Port Status Flags)

Error Flag This flag is set if a fragment belonging to an already finished port is received before the next communication round started. This situation

occurs if the port size was not configured correctly by the host.[3] The error flag register can be cleared by the host by writing `1` to it, even while the Voter Plug-In is enabled.

Port Size The host has to configure the correct size of the A, B, and C ports in this register,[4] to allow the Voter Plug-In to detect the completion of a received pulsed data stream. Since the port size of the three ports is restricted to 63 fragments, this port size register is only six bit wide (a port size of 64 fragments ("`111111`") is prohibited).

In addition to the status register, the Voter Plug-In provides the port status and error information in the result port port flags to simplify host access. Behavior and meaning of the flags in the result port port flags, shown in figure 5.12, are the same as described above. The empty flag is `0` (port empty) only before completing the first successful (i. e., at least two ports are equal) voting round voting. The sequencer is even at all times, since the result port data is validated by switching the appropriate memory pointer simultaneously to increasing the sequencer.

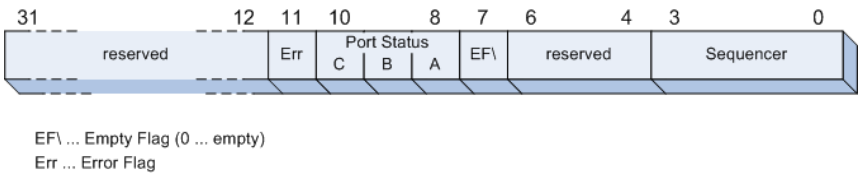

Figure 5.12: Voter Plug-In Result Port Port Flags

5.2.3 Other Plug-Ins

A wide range of functions improving the LLC and CNI Layers can be designed as Plug-In. Some possible future extensions are listed below.

Security/Safety Plug-Ins Various security and data safety functions can be implemented transparently to the host by means of a Plug-In. These functions range from simple message counters to preventing unauthorized communication. Other useful mechanisms would include data encryption

[3] Under some circumstances, loosing the first fragment of a voting round immediately after loosing a fragment in the previous voting round could lead to the same situation, even if the port size was configured correctly. Since there is no possibility to detect such an unlikely case, the Voter Plug-In does not distinguish between those two faults.

[4] The result port is automatically configured to the same size.

and decryption, checksum generation, parity bits, Cyclic Redundancy Checks (CRCs), port data feasability control (e. g., range checks, temporal validity), etc.

Timer Interrupt Plug-In Additional or other kinds of timer interrupts may be needed by certain applications. By integrating them into a Plug-In, in contrast to a Hardware Middleware Brick, the timer interrupt service gains the possibility to constantly access the current global real-time. No register reading is required, therefore missing a time instant (while performing the read operation) is prohibited. Furthermore, the Interrupt Service of the CNI Layer can be used to trigger the generated interrupt requests (IRQs), improving the usability by rendering further hardware changes (e. g., additional IRQ lines) obsolete.

I/O Plug-In When equipped with an external interface, a Plug-In may read sensor data or control an actuator without interfering (or being interfered by) the host. Control data can be received from another host or a Plug-In in another core, as well as the sensor data can be distributed over the NoC to another core immediately. Such a Plug-In could keep its own host informed by means of status registers, or receive control information through configuration registers.

CNI Memory Even the standard CNI memory can be categorized as a Plug-In. Since the CNI memory is mandatory, it is not considered as a Plug-In throughout this document, but a different version of the port data memory can definitely be seen as a Plug-In. Possible modifications would include a bigger (or smaller) memory, different types of memory, external (i. e., off-chip) memory, etc.

6 Conclusion

By implementing the time-triggered concept for communication over the Network-on-a-Chip, the Time-Triggered System-on-a-Chip architecture maintains strict encapsulation of different cores. The rising complexity inherent to today's distributed embedded systems can be controlled using this encapsulation, by means of allowing the application system to be broken down into different Distributed Application Subsystems which can be designed independently. No side effects or mutual dependencies between different DASs are caused by integrating them into the overall system.

While the TNA and the TISSs are, among other tasks, responsible to protect the NoC from unrestricted access (i. e., temporal faults), thus guaranteeing encapsulation, the extendible host interface is capable of providing all functions necessary for a host processor to communicate over the time-triggered Network-on-a-Chip. The extendible host interface not only fulfills this basic requirement in a resource efficient way, but also maintains a replica deterministic behavior while achieving a reasonable data throughput. Support for diagnosis is provided as well as a real-time service, which, in conjunction with other supporting functions and a well defined host interface, improve the (re-)usability of the whole TTSoC architecture. In case an application demands other additional functions to be performed by the NoC hardware, interface extensions can be implemented.

To guarantee sufficient extendability of the TTSoC architecture with reasonable resource usage and design effort, the possibility to add custom Middleware Plug-Ins was introduced. These Plug-Ins benefit from direct access to the port data they operate on, without the need to adapt host access procedures or installed host application software. A DMA Plug-In, intended for multimedia applications with high data transfer requirements, and a Voter Plug-In, dedicated to support Triple Modular Redundancy with minimal voting latency, are used to demonstrate the capabilities of the Middleware Plug-In approach.

Taking the characteristics of all components forming the TTSoC architecture and the way they are integrated and interact with each other into account, one can determine that the Time-Triggered System-on-a-Chip architecture is able to meet its requirements and proves to be a valuable asset concerning future distributed embedded application systems.

The Trusted Interface Subsystem and the Middleware Plug-Ins, as well as the whole Time-Triggered System-on-a-Chip architecture will undergo future improvements and changes. Possible future developments are outlined below. They are meant as an impulse for future design considerations.

- Various new Middleware Plug-Ins can be designed to further improve the usability of the TTSoC architecture.
- A great number of different host processors could be used with the TTSoC architecture. Additional adapters for different, host-specific peripheral interfaces could be implemented to allow them to communicate using the Open Core Protocol.
- Different types of interconnect topologies of the NoC could be adapted for different application requirements (large bandwidth, power efficiency, preserving on-chip resources, etc.).

A Open Core Protocol

The Open Core Protocol (OCP) is administered by the OCP International Partnership (OCP-IP) Association [OCP] and defines a master-slave interface designed for on-chip peer-to-peer communication between IP-cores, buses, etc. A large variety of signals and operational modes are available with OCP, although only those which are needed by the SoC design have to be implemented.

The possibility to easily adapt the OCP interfaces to match the requirements of the implemented SoC, along with a broad acceptance of the Open Core Protocol among various device manufacturers and IP-core developers, were the driving arguments to use the Open Core Protocol for the CNI Layer Host Interfaces of the TTSoC architecture. Some of the capabilities of the Open Core Protocol, which influenced the decision for the OCP Host Interface are listed below (taken from [Inc04, p. 1]).

- *Small set of mandatory signals, with a wide range of optional signals*
- *Synchronous, unidirectional signaling allows simplified implementation, integration and timing analysis*
- *Configurable address and data word width*
- *Structured method for inclusion of sideband signals: high-level flow control, interrupts, power control, device configuration registers, test modes, etc.*

This chapter focuses on the OCP signals and operational modes used by the CNI Layer Host Interface, the complete OCP specification can be found in [OCP05].

A.1 OCP Signals

Only two OCP signals, the OCP clock (*clk*) and the master command (*MCmd*) signals are mandatory for every OCP implementation, all other signals can be chosen to meet the specific requirements. The following signals are used by the OCP Host Interface of the CNI Layer, implemented as an OCP slave. A preceding "M" in the signal name identifies a master-to-slave signal, while a preceding "S" stands for a slave-to-master signal.

Dataflow Signals

Basic Signals

OCP Clock (clk): The on-chip clock signal is used as OCP clock, all dataflow signals are sampled at the rising edge of this clock signal.

Master Address (MAddr): 15 bit address, specifies the memory location to read from or write to.

Master Command (MCmd): 3 bit command signal, specifies the OCP master command ("Read", "Write", etc.). Table A.1 lists all possible OCP commands. Only the commands marked with * are supported by the implemented OCP Host Interface, all other commands are ignored by the CNI Layer.

MCmd[2:0]	Function
0 0 0	Idle *
0 0 1	Write *
0 1 0	Read *
0 1 1	ReadEx
1 0 0	ReadLinked
1 0 1	WriteNonPost
1 1 0	WriteConditional
1 1 1	Broadcast

Table A.1: OCP Master Commands

Master Data (MData): 32 bit write data

Slave Command Accept (SCmdAccept): 1 bit slave command accept, set by the slave (the CNI Layer) on accepting the issued OCP Master Command ("Read" or "Write").

Slave Data (SData): 32 bit read data

Slave Response (SResp): 2 bit slave response, set by the slave (the CNI Layer) when presenting the read data (*SData*) after performing a "Read" command to indicate the validity of the data. Note that only a "Read" command triggers a response, "Write" commands are not confirmed by the CNI Layer. Table A.2 lists all possible OCP responses.

Simple Extensions

Master Byte Enable (MByteEn): 4 bit byte write enable signal, indicates which bytes of the 32 bit write data (*MData*) are to be written.

SResp[1:0]	Response	Mnemonic
0 0	No response	NULL
0 1	Data valid / accept	DVA
1 0	Request failed	FAIL
1 1	Response error	ERR

Table A.2: OCP Slave Response Encoding

Sideband Signals

These sideband signals are independent from any data transfer, the CNI Layer can assert them at any time.

Slave Error (SError): This signal is asserted by the slave (the CNI Layer) whenever a read or write error occurs (e. g., wrong register write, unsupported OCP command, etc.).

Slave Interrupt (SInterrupt): An interrupt request (IRQ) is sent to the OCP master using this signal. The *SFlag* signal is used to indicate the interrupt source simultaneously.

Slave Flags (SFlag): 6 bit interrupt information, used to determine the source of an IRQ.

Slave Reset (SReset_n): This low-active reset signal forces the OCP master (e. g., the host processor) to enter a defined powerup state. According to the OCP specification, the OCP slave (the CNI Layer) holds this signal active for at least 16 OCP clock cycles to ensure a proper reset.

A.2 OCP Operation

Although the Open Core Protocol supports a large variety of operational modes, the CNI Layer OCP Host interface supports "Read" and "Write" commands only, no broadcasts, bursts, threading, etc. are available. This decision was made mainly for simplicity reasons. Every additional, sophisticated form of communication would afford additional logic elements, wiring, memory, etc., increasing the complexity of the CNI Layer, and, as a consequence, the whole TTSoC. With simple read and write commands all required functions can be performed with reasonable effort. Any higher level communication protocols (e. g., bursts) have to be implemented by the host software.

A simple OCP read or write data transfer starts with a **request phase** initiated by the OCP master. The OCP master switches the *MCmd* signal from

"Idle" to "Read" or "Write" and, at the same time, presents the target address of the transfer on the *MAddr* lines. In case of a "Write" command, valid data is presented on the *MData* signal simultaneously. The OCP slave captures the values from the *MData* and/or *MAddr* signals and starts the internal read or write operation, according to the asserted OCP Master Command. The Slave Command Accept (*SCmdAccept*) signal is set to inform the OCP master that the OCP slave accepted the command and that the request phase has ended. As a consequence, the *MCmd* signal is set to "Idle" again.

A write transfer ends after the request phase, while a read transfer continues with its **response phase**. Once the OCP slave has prepared the requested data (e. g., loaded it from internal memory), the Slave Response (*SResp*) signal is set to "Data Valid", indicating valid data on the *SData* signal. The OCP master stores this data, and the response phase is completed.

The CNI Layer OCP Host interface uses the *SError* signal, together with the "Response Error" signal on the *SResp* lines in case of a read transfer, to inform the OCP master about the occurrence of an error.

Further OCP signal and operation details are explained in the Open Core Protocol specification ([OCP05]).

B Avalon Adapter

An Altera NIOS II Processor [Alt07c] was used to test some components of the NoC implementation. Since the CNI Layer Host Interface was designed as an OCP Slave interface, but the NIOS II processor uses the so called "system interconnect fabric" and "Avalon Memory-Mapped interfaces" to connect any peripherals, an OCP-to-Avalon adapter was needed. Its implementation is described in this appendix.

B.1 Avalon Memory-Mapped Interface

The Altera Avalon Memory-Mapped interface, as specified in [Alt07a], is used by the Altera system-on-a-programmable-chip (SOPC) Builder of the Quartus II software to connect predefined components (e. g., the NIOS II processor) and/or custom components using a graphical user interface. It defines master- and slave-type interfaces which can be connected in arbitrary way. The system interconnect fabric takes care of all administrative tasks arising from customization, e. g., access arbitration, address mapping, chip select assignment, data width reduction/expansion, etc.

The Avalon Memory-Mapped interface defines a variety of signals and their behavior and different supported transfer types (e. g., read, write, burst, etc.). Not all signals and transfer types have to be implemented by a peripheral, only *clk*, *waitrequest*, and *address* are mandatory for an Avalon master interface. Any used signal is also available in its inverted form, highlighted by adding $\_n$ to the signal name.

In the following sections only those signals and transfer types actually used by this implementation are described, any further details of the Avalon Memory-Mapped Interface can be looked up in [Alt07a].

B.2 Avalon Slave Adapter

The purpose of the Avalon Slave Adapter is to allow an easy integration of the NoC into systems designed using the Altera SOPC Builder. It implements an

Avalon Memory-Mapped slave-type interface, an OCP Master interface tailored to match the OCP Host Interface of the CNI Layer, and the necessary glue logic to translate the requests and responses and some sideband signals between these two interfaces.

Since intended for testing purposes only, the Avalon Slave Adapter provides only simple read/write functionality and a minimal IRQ support. The propagation of OCP interface error signals and the differentiation of the TTSoC IRQs were neglected, resulting in a single IRQ line for all possible IRQs and the read/write error detection delegated to the system interconnect fabric and the host processor. Read and write timing was fixed using the SOPC Builder, no Avalon response signals (e. g., *readdatavalid*) were used.

The host reset initiated by the LLC Layer watchdog service is routed directly to the *resetrequest_n* signal of the (probably) connected NIOS II processor via the OCP *SReset_n* signal outside the Avalon Slave Adapter. This is done to prevent an automatic reset of all systems in the core connected to the system interconnect fabric in case of a watchdog miss. The necessity to reset the whole core is application dependent, therefore only the host processor is reset, other peripherals, like off-chip communication systems, are allowed to continue operation unless the host processor itself propagates the reset signal.

The signals used by the Avalon Slave Adapter and their mapping to the OCP signals are summarized in table B.1.

Avalon Signal	Width	Note	↔	OCP Signal	Width
address	15 bit	directly connected	→	*MAddr*	15 bit
readdata	32 bit	directly connected	←	*SData*	32 bit
writedata	32 bit	directly connected	→	*MData*	32 bit
read *write*	1 bit 1 bit	start read/write operation, OCP command is selected accordingly (*MCmd*=“Idle” otherwise)	→	*MCmd*	3 bit
byteenable	4 bit	directly connected	→	*MByteEn*	4 bit
irq_n	1 bit	inverted	←	*SInterrupt*	1 bit
		used to set *MCmd* back to “Idle” after successful transaction	← ←	*SResp* *SError*	2 bit 1 bit

Table B.1: Avalon Slave Adapter Signals

B.3 Avalon Master Adapter

Similar to the Avalon Slave Adapter, the Avalon Master Adapter allows integration of the DMA Middleware Plug-In into an SOPC Builder system. Both operate in parallel, the Avalon Master Adapter is used exclusively by the DMA Plug-In to store or load data to or from the host's main memory. Any memory and/or access arbitration inside the host processor system is carried out by the system interconnect fabric.

An OCP Slave interface and an Avalon Memory-Mapped master-type interface are incorporated by the Avalon Master Adapter, both support only the minimal functionality to transfer single data words. No error detection is performed and no sideband signals are used.

Table B.2 lists the implemented signals and their relationships.

Avalon Signal	Width	Note	↔	OCP Signal	Width
waitrequest	1 bit	used by the system interconnect fabric to delay a transaction, stalls *SResp* and/or *SCmdAccept*			
address	32 bit	directly connected	←	*MAddr*	32 bit
readdata	32 bit	directly connected	→	*SData*	32 bit
writedata	32 bit	directly connected	←	*MData*	32 bit
read *write*	1 bit 1 bit	start read/write operation, triggered according to OCP command	←	*MCmd*	3 bit
		set if read/write is accepted by the system interconnect fabric	→	*SCmdAccept*	1 bit
		set to "Data Valid" if *readdata* is valid	→	*SResp*	2 bit

Table B.2: Avalon Master Adapter Signals

C Powerlink Bus Adapter

During the development of the Time-Triggered System-on-a-Chip architecture, a test environment will be created, consisting of an FPGA device (housing the Network-on-a-Chip, the TNA, the TISSs and the CNI Layers of all cores) which is connected to external host processors. These processors are located on ^{TTP}Powerlink CPU modules (e. g., as described in [TTT02] and [TTT05]) and connected to the FPGA through the PowerLink Connection Bus version 1 (PLCB1), specified in [TTT01].

For this test implementation of the TTSoC an adapter between the PowerLink Connection Bus (PLCB1) and the OCP Host interface is needed, which is the subject of this appendix.

C.1 PowerLink Connection Bus

The PowerLink Connection Bus (PLCB1) is a *space saving stacked-PCB* [Printed Circuit Board] *micro controller solution for TTP/C, well suited for Motorola PowerPC controllers like the MPC555.* [TTT01, p. 4] The PLCB1 specification defines mechanical properties of PLCB1 conformant Printed Circuit Boards as well as the electrical issues of the board connectors.

A complete list of all PLCB1 signals is available in [TTT01], the Powerlink Bus Adapter uses only a subset of the following MPC555-bus signals that are available through the Powerlink bus: $D31$-$D0$, $A31$-$A8$, $\overline{CS2}$-$\overline{CS3}$, $\overline{OE}$, $RD/\overline{WR}$, $\overline{WE/BE0}$-$\overline{WE/BE3}$, $\overline{TS}$, $\overline{TA}$, $\overline{TEA}$, $\overline{BDIP}$, $\overline{RESET}$, $EPEEBUS$, $RCFB$, $\overline{BR}$, $\overline{BG}$, $\overline{BB}$, $\overline{IRQ0}$-$\overline{IRQ4}$, $\overline{IRQOUT}$, CLK. A description of the Motorola MPC555 CPU and the MPC555-bus signals and operation can be found in [Mot00]. Other processor boards can be adapted to use these signals of the PLCB1 bus.

C.2 Implementation

The following PLCB1 bus signals are used by the Powerlink Bus Adapter. Input or output lines, as stated in the description, only concern the Powerlink

Bus Adapter, in general, all PLCB1 bus lines are bi-directional.[1]

Data Bus ($D31$-$D0$): 32 inout data lines to transport a single 32-bit double word. These data lines are also driven by other components and therefore have to be tri-stated if not in use to prevent interferences. They are connected to the *MData* or *SData* signals depending on the type of transfer and the $\overline{OE}$ signal.

Address Bus ($A31$-$A8$): 24 address input lines. For the CNI Layer Host interface address space only 15 address lines are required, directly connected to the *MAddr* signal.

Chip Select ($\overline{CSx}$): This low-active signal is used to distinguish between accesses to other external devices and the CNI Layer. Furthermore, the start and the end of a transfer is determined by this signal. Either one of the two available chip select lines can be connected to the Powerlink Bus Adapter. (input)

Output Enable ($\overline{OE}$): Asynchronous low-active output enable. Has to be asserted to allow the Powerlink Bus Adapter to lay data on the PLCB1 bus. (input)

Read/Write ($RD/\overline{WR}$): Determines the type of a host access, read (high) or write (low). The *MCmd* signal is set accordingly. (input)

Write Enable/Byte Enable ($\overline{WE/BE0}$-$\overline{WE/BE3}$): These low-active input signals are used as byte write enable signals, therefore they are inverted and connected to the *MByteEn* signal.

Transfer Acknowledge ($\overline{TA}$): Informs the host processor about the completion of a transfer. Once a transfer is initiated, the Powerlink Bus Adapter sets this signal to `1` to stall the host. After completion of the transfer, i. e., after the data is written in case of a write access or the data is loaded and ready to be stored by the host in case of a read access, this signal is pulled to `0` and held low until the chip select ($\overline{CSx}$) is de-asserted. Otherwise, this output signal is tri-stated.

Reset ($\overline{RESET}$): This low-active signal can be used to reset the host processor. It is asserted on a Powerlink Bus Adapter reset, initiated by the chip-wide reset signal or the OCP Slave Reset (*SReset_n*) signal. (output)

Interrupt Request ($\overline{IRQ0}$-$\overline{IRQ4}$): One or more of these low-active interrupt request (IRQ) lines are triggered by an OCP *SInterrupt* signal. The *SFlag* signal is used to determine the interrupt source and set the $\overline{IRQ0}$-$\overline{IRQ4}$ lines accordingly.

[1] Exceptions: $\overline{IRQ0}$-$\overline{IRQ4}$, $\overline{RESET}$: output; CLK, $\overline{IRQOUT}$: input

Transfer Start, Transfer Error Acknowledge, Burst Data In Progress ($\overline{TS}$, $\overline{TEA}$, $\overline{BDIP}$): These signals are not in use, but have to be tri-stated for compatibility reasons.

Clock (CLK): Host processor clock input. The host processor clock can be used to time the PLCB1 bus operation, allowing all PLCB1 bus signals (except $\overline{OE}$) to be sampled at the rising edge of this clock signal. *Note that CLK may or may not be connected to the MPC555's* [or another processor's] *internal clock on the CPU module - this depends on a hardware option.* [TTT01, p. 8]

The first TTSoC test beds dedicated for use with the Powerlink Bus Adapter did not feature an MPC555-bus clock signal. As a consequence, the Powerlink Bus Adapter was designed to operate independently from the PLCB1 bus clock, all signals were sampled using the on-chip clock signal. Although this implementation proofed to work correctly even if the TTSoC and the PLCB1 bus run with different clock frequencies, a second version using the MPC555-bus clock signal was built to be used in future test environments.

Bibliography

[Alt07a] Altera, 101 Innovation Drive, San Jose, CA 95134. *Avalon Memory-Mapped Interface Specification*, 3.3 edition, May 2007. www.altera.com.

[Alt07b] Altera, 101 Innovation Drive, San Jose, CA 95134. *Cyclone II Device Handbook, Volume 1*, 3.2 edition, 2007. CII5V1.

[Alt07c] Altera, 101 Innovation Drive, San Jose, CA 95134. *Nios II Processor Reference Handbook*, 7.0 edition, March 2007. NII5V1.

[CS04] Dependable Embedded COmponents and Systems. DECOS. www.decos.at/VideosAndPictures/DECOS_fs1.pdf, September 2004.

[DeL99] R. DeLine. A catalog of techniques for resolving packaging mismatch. In *SSR '99: Proceedings of the 1999 symposium on Software reusability*, pages 44–53, 1999.

[Dep] Dependable Embedded Components and Systems. *www.decos.at*.

[EEE+01] S. Eberle, C. Ebner, W. Elmenreich, G. Färber, P. Göhner, W. Haidinger, M. Holzmann, R. Huber, R. Schlatterbeck, H. Kopetz, and A. Stothert. Specification of the TTP/A protocol. Research Report 61/2001, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2001.

[Eng07] G.A. Engleder. Time-Triggered Network-on-a-Chip. Master's thesis, Computer Engineering, Real-Time Systems Group, Vienna University of Technology, 2007.

[Ham03] R. Hammett. Flight-critical distributed systems: Design considerations. *Aerospace and Electronic Systems Magazine, IEEE*, 18(6):30–36, June 2003. Charles Stark Draper Lab. Inc., Cambridge, MA, USA.

[Inc04] OCP-IP Association Inc. Open Core Protocol (OCP) datasheet. www.ocpip.org/socket/datasheets/OCP_2.2_Datasheet.pdf, 2004.

[KB03] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.

[KHO+06] H. Kopetz, B. Huber, R. Obermaisser, C. Salloum, and M. Schoeberl. *Design of System-on-a-Chip Component.* Vienna University of Technology, Austria, October 2006.

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

[Kop04a] H. Kopetz. An integrated architecture for dependable embedded systems. *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 160–161, October 2004.

[Kop04b] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer, 2004.

[Kop06] H. Kopetz. Pulsed data streams. Research Report 14/2006, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, February 2006.

[KR93] H. Kopetz and J. Reisinger. NBW: A non-blocking write protocol for task communication in real-time systems. January 1993.

[Lap92] J.C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems.* Springer Verlag, 1992.

[Mot00] Motorola. *MPC555 / MPC556 User's Manual*, October 2000.

[Obe06] R. Obermaisser. *DECOS System-on-a-Chip Component - Specification.* Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 0.2w edition, January 2006. Task 2.2.4.

[OCP] OCP International Partnership, 3855 SW 153rd Drive, Beaverton, Oregon 97006 USA. *www.ocpip.org.*

[OCP05] OCP International Partnership Association, Inc., 3855 SW 153rd Drive, Beaverton, Oregon 97006 USA. *Open Core Protocol Specification 2.1*, 1.0 edition, 2005.

[OPHS06] 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.

[PAB+06] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a

first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179–196, January 2006.

[Pol94] S. Poledna. *Replica determinism in fault-tolerant real-time systems*. PhD thesis, Technische Universität Wien, 1994.

[POT+05] P. Peti, R. Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An integrated architecture for future car generations. *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 2–13, May 2005. Vienna University of Technology, Austria.

[SK06] K. Steinhammer and H. Kopetz. Time-Triggered Ethernet. *Junior Scientist Conference (JSC 2006), Vienna, Austria*, April 2006.

[Sta01] W. Stallings. *Operating Systems*. Prentice-Hall International, Inc., fourth edition, 2001.

[Ste06] K. Steinhammer. *Design of an FPGA-Based Time-Triggered Ethernet System*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[TTTa] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP/C Specification*. Available at *www.tttech.com*.

[TTTb] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *www.tttech.com*.

[TTT01] TTTech, Schönbrunner Straße 7, A-1040 Vienna, Austria. *PLCB1 bus specification*, 1.0.00 edition, July 2001. Specification for TTTechs PLCB1 bus.

[TTT02] TTTech, Schönbrunner Straße 7, A-1040 Vienna, Austria. *TTP Powerlink CPU module*, 3.0.02 edition, October 2002. A MPC555 CPU board for the TTP Powerlink PLCB1 bus.

[TTT05] TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, Austria. *TTP Powerlink CPU TC1796*, 0.2.00 edition, October 2005. A CPU Board with the Infineon TC1796 TriCore CPU for the TTP Powerlink PLCB1 bus.