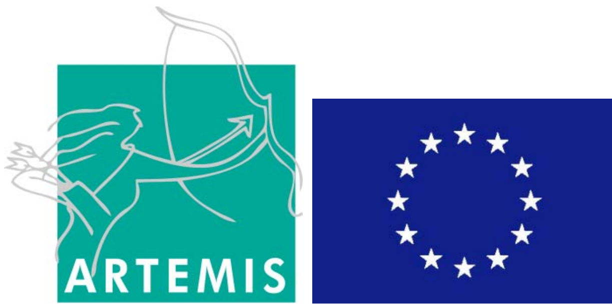

ACROSS
ARTEMIS Cross-Domain Architecture

Design Document of the MPSoC Core Services

Project Acronym	ACROSS		Grant Agreement Number	ARTEMIS-2009-1-100208	
Document Version	1.0	Date	2011-06-30	Deliverable No.	D1.3
Contact Person	Christian Paukovits		Organisation	TTTech Computertechnik AG	
Phone	+43 1 585 34 34 - 19		E-Mail	christian.paukovits@tttech.com	

Document versions

Version No.	Date	Change	Author(s)
0.1	2011-01-11	initial draft version	Christian Paukovits
0.2	2011-04-18	extended draft version	Christian Paukovits
0.3	2011-06-03	• added boot service from TU Vienna • time stamps revised, introduced time stamp counter • added note on inter-component channel configuration • introduced conditional ports • propagated changes to affected sections and figures	Christian Paukovits Armin Wasicek Martin Elshuber
0.4	2011-06-30	• included comments of Cassidian Electronics, added clarifications • changed behavior of register INTMASK (no longer write-only) • added clarifications for the registers of interrupt handling (INTMASK and INTUNMASK) • removed section of basic boot service	Christian Paukovits
1.0	2011-06-30	• final typos corrected	Andreas Eckel

Table of Contents

Document versions 2
1 About this Document 7
1.1 Role of the Deliverable 7
1.2 Relationship to other ACROSS Documents 7
1.3 Structure of this Document 7
2 Interface to Core Services 8
2.1 Layout of the Uniform Network Interface 8
2.1.1 Port Interface 9
2.1.2 Control Interface 9
2.2 Basic Communication Services 10
2.2.1 Memory Layout of Ports 11
2.2.2 Memory-consistent access to Ports 14
2.2.3 Port Configuration 20
2.2.4 Register File of the Basic Communication Services 21
2.3 Component Control Service 21
2.3.1 Component control channels 21
2.3.2 Roles of Component Control 22
2.3.3 Command Messages 22
2.3.4 Register File of Component Control Service 27
2.4 Task Trigger Service 27
2.4.1 Realization of Task Triggers 27
2.4.2 Register File of the Task Trigger Services 28
2.5 Common Time Service 28
2.5.1 Realization of Periods 28
2.5.2 Duration of Periods 29
2.5.3 Active Periods 29
2.5.4 Register File of the Common Time Service 30
2.5.5 Time stamps 30
2.6 Timer Interrupt Service 31
2.6.1 Realization of the Timer Interrupt Service 31
2.6.2 Running Modes 32
2.6.3 Activation 32
2.6.4 Register File of the Timer Interrupt Service 32
2.7 Error Detection Service 33

2.7.1 Error Sources and Error Flags 33
2.7.2 Clearing Error Flags 34
2.7.3 Watchdog 34
2.7.4 Register File of Error Detection Service 35
2.8 Health Reporting Service 35
2.8.1 Health Reporting Channels 35
2.8.2 Structure of Health State Messages 36
2.8.3 Omitting Application-Specific Data 37
2.8.4 Clearing Error Flags of the Error Detection Service 37
2.8.5 Register File of the Health Reporting Service 38
2.9 Identification Service 38
2.10 Inter-Component Channel Configuration 38
3 Application Programming Interface 39
3.1 The TISS Driver 39
3.2 Register Access Functions 39
3.2.1 Function: across_getRegPtr() 40
3.2.2 Function: across_getIRV() 41
3.2.3 Function: across_getComponentID() 41
3.3 Send & Receive Functions 42
3.3.1 Blocking Mode of Send & Receive Functions 42
3.3.2 Back-end function: across_getMsgPtr() 43
3.3.3 Back-end function: across_commitMsg() 44
3.3.4 Front-end function: across_recvMsg() 44
3.3.5 Front-end function: across_sendMsg() 45
3.4 Interrupt Callback Mechanism 46
3.4.1 Dispatching of interrupts 46
3.4.2 Set-up of hooks: across_setHook() 48
3.4.3 Set-up of interrupt masks 49
3.4.4 User-specified dispatcher functions 49
3.4.5 Limiting size of hooks 49
4 Protected Memories 50
4.1 Port Parameter Memory 50
4.1.1 Subscription of Ports 51
4.2 Time-Triggered Schedule 52
4.2.1 Entries of the Time-Triggered Schedule 52
4.2.2 Layout of the Time-Triggered Schedule 53
4.3 Burst Configuration Memory 55
4.4 Routing Information Memory 56

4.5 Register File of Protected Memories ... 56
4.6 The Exception – Write-Access to Protected Memories ... 57
5 The Interrupt Mechanism of the TISS ... 58
5.1 Interrupt sources ... 58
5.2 Register File of the Interrupt Control ... 58
5.3 Handling Interrupt Flags ... 59
5.3.1 Interrupt Status Register (INTSTAT) ... 59
5.3.2 Interrupt Acknowledge Register (INTACK) ... 60
5.3.3 Interrupt Mask Register (INTMASK) ... 60
5.3.4 Interrupt Unmask Register (INTUNMASK) ... 60
5.3.5 Message Complete Interrupts ... 61
5.4 Interrupt Lines ... 61
6 Data types, structures and constants ... 63
6.1 Platform-specific data types ... 63
6.2 Platform-independent data types ... 63
6.3 Error codes of driver functions ... 64

Table of Figures

Figure 1: Schematics on an ACROSS component 8
Figure 2: structure of the Control Interface's address space 9
Figure 3: memory layout of outgoing periodic ports 12
Figure 4: memory layout of ports with incoming periodic messages 13
Figure 5: memory layout of sporadic ports 13
Figure 6: layout of a memory entry of the Port Synchronization Memory 14
Figure 7: operation of synchronization flags of ports with outgoing periodic messages 17
Figure 8: layout of a memory entry of the Port Configuration Memory 20
Figure 9: layout of the register file of the basic communication services 21
Figure 10: structure of a command message (without time stamp) 23
Figure 11: structure of a command message (with time stamp) 23
Figure 12: trigger modes of component control 26
Figure 13: layout of the register file of the task trigger service 28
Figure 14: layout of the register file of the common time service 30
Figure 15: structure of a time stamp 31
Figure 16: layout of the register file of the timer interrupt service 33
Figure 17: layout of the register file of the error detection service 35
Figure 18: structure of a health state message 37
Figure 22: the interrupt callback mechanism 47
Figure 23: layout of a memory entry of the Port Parameter Memory 50
Figure 24: layout of an entry of the time-triggered schedule 52
Figure 25: layout of the time-triggered schedule 53
Figure 26: layout of a memory entry of the Burst Configuration Memory 55
Figure 27: layout of the register file of the protected memories 56
Figure 28: layout of register file of interrupt control 59

1 About this Document

1.1 Role of the Deliverable

This document elaborates on the design of the core services that are to be implemented by WP1. It presents the core services from the point of view of the interface to the host within an ACROSS component.

1.2 Relationship to other ACROSS Documents

The functional specification of the core services was described in prior deliverable D1.2. This document is the basis for the implementation phase, which results in the deliverables D1.6 and partly in D1.7.

1.3 Structure of this Document

Chapter 2 describes the design of all those core services associated with WP1, which operation is visible at the physical and logical interface between Trusted Information Subsystem (TISS) and the ACROSS component's host. It incorporates the mapping of functional behaviour to memories and register files, through which the operation of a given core services is controllable and observable.

Chapter 3 deals with the Application Programming Interface (API) how these core services are made usable in application software executed on the ACROSS component's host.

Chapter 4 contains supplementary information to chapter 2 on layout of memories within the TISS.

Chapter 5 explains hardware interrupts of core services. In this context, it lists the interrupt sources and how the handling of interrupts in register files has to be realized by the API, which is detailed in section 3.4.

Finally, chapter 6 lists source code data types and symbols used by the API.

2 Interface to Core Services

This chapter presents the realization of the core services in a concrete design and their interface from the point of view of a host based on their functional specification.

2.1 Layout of the Uniform Network Interface

The core services are available at the interface between the TISS and the host, which has the name Uniform Network Interface (UNI). If it comes to an implementation of ACROSS, a TISS is realized as a dedicated module embedded in a higher order entity of a hardware modeling language. This higher order entity embodies an ACROSS component, which in turn is a building block from an even higher hierarchical point of view. At the top level entity the conjunction of all building blocks forms the overall ACROSS MPSoC.

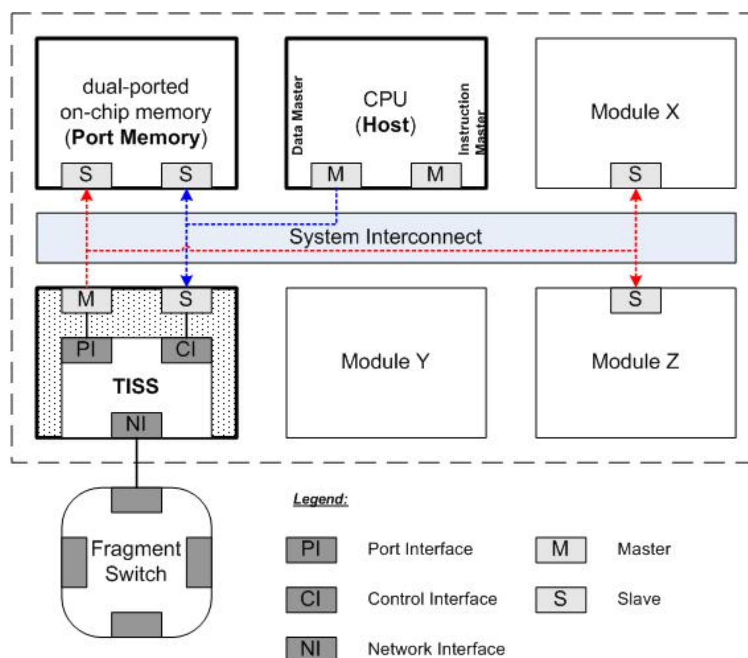

Figure 1: Schematics on an ACROSS component

Within an ACROSS component the TISS must be compatible to other modules, which are connected by means of a given system interconnect fabric. This implies that the TISS wraps its native UNI interface to the semantics of the component internal system interconnect's transactions and obeys signal and naming conventions. In case of ACROSS the *Altera AVALON Memory-Mapped Interconnect* has been chosen as the system interconnect within ACROSS components. Consequently, the TISS provides AVALON memory-mapped master and slave interfaces.

Figure 1 illustrates the schematic outline of an ACROSS component. In terms of AVALON, the UNI consists of a memory-mapped master and a memory-mapped slave. The master maps the TISS' Port Interface that conveys real communication data. The slave corresponds to the TISS' Control Interface, which provides a memory-mapped register file to report status information and to enable control and configuration of core services. Details about these interfaces with respect to signal

specifications and timing of the native UNI and the AVALON interface are described in deliverable D1.6.

2.1.1 Port Interface

We also learn from Figure 1 that the TISS' Port Interface has exclusive access to one port of a dual-ported on-chip memory, which takes the role of the *Port Memory*. The Port Memory serves as a buffer for incoming and outgoing messages transmitted by the basic communication services. Additionally, the Port Interface might perform direct access to other modules that offer payload data for transmission via the TTNoC. The other port of that dual-ported memory is reserved for other modules of the ACROSS component, i.e., the CPU. Similarly, the CPU and its peripherals take the role of the *host* and access the Control Interface of the TISS.

2.1.2 Control Interface

The Control Interface is used by the host to retrieve status and configuration information, synchronization flags, control registers of core services, which control the operation of a restricted set of functions of most core services. In general, the Control Interface is a 32-bit wide, memory-mapped interface that presents data in register files. For each core service a dedicated register file assigned to a given address range is reserved. Additionally, there are register files for the interrupt mechanism and the protected memories of the TISS. The address ranges to which register files are mapped are characterized by their start offsets and the length, which can be derived from the number of 32-bit data words in that register file.

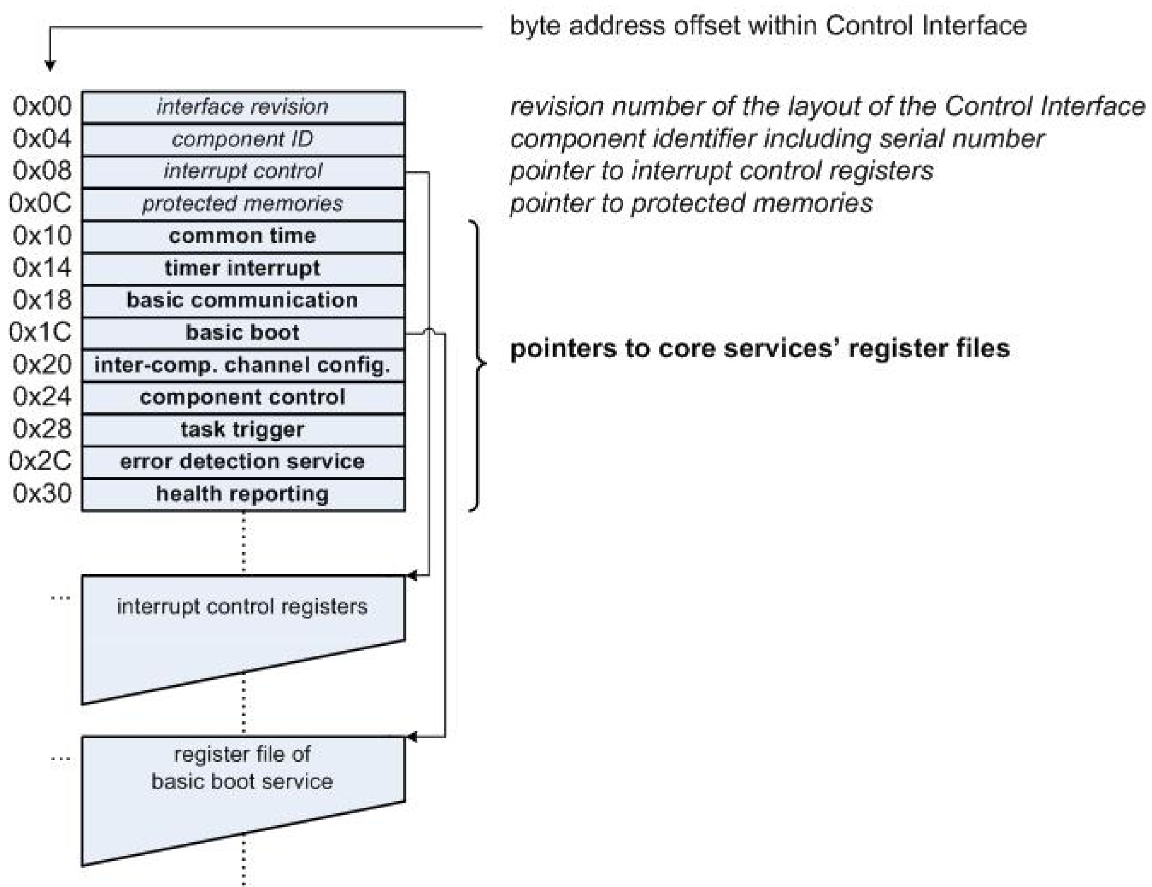

Figure 2: structure of the Control Interface's address space

2.1.2.1 Lookup-Table of Register Files

To avoid accesses to entities by hard-coded address offsets and thus facilitate changes of the Control Interface's layout without recompilation of the application software, the Control Interface introduces a look-up table of pointers that refer to the start offsets of register files. In fact, a pointer provides a 32-bit data word containing the **word-aligned byte address offset of the beginning of some register file**. Figure 2 illustrates the structure of the Control Interface's address space.

If the host accesses a given entity of a core service or an interrupt register or protected memory, it has to extract the start offset of the corresponding register file and then add that entity's offset within the register file to determine the relative address within the Control Interface. Also note that the Control Interface is a memory-mapped interface and incorporates a base address in a (AVALON) master's address space, which has to be included in the calculation of an entity's absolute address. As defined in section 3.2, this process of address translation is wrapped by the TISS driver and is usually abstracted from the application software.

As we can also see in Figure 2, the pointers to core services' register files begin at byte address offset 16 (0x10) of the Control Interface. The layout of each register file is described in the section of the corresponding core service.

2.1.2.2 Interface Revision Number & Component ID

At byte address offset 0 of the Control Interface we find the *interface revision number*. This is a 32-bit wide hexadecimal read-only string of a date in the format YYYY/MM/DD that identifies the layout of the look-up table and each register file plus the features of the TISS driver. We recommend checking the interface revision number for compatibility in the application software at start-up.

The register *component ID* at byte address offset 4 includes a numeric identifier of the ACROSS component, which enumerates that component in the MPSoC starting from number 1. That component identifier is 8-bit wide and located at the least significant bits of this 32-bit word. The remaining 24 bit embody the unique numeric serial number of the MPSoC.

These two registers realize the identification service of ACROSS.

2.1.2.3 Pointer to Interrupt Control Register File

Byte address offset 8 of the Control Interface contains a pointer to the interrupt control registers, which does not belong to core services directly but maintains the interrupts produced by the core services. Details of the interrupt mechanism of the TISS can be found in chapter 5.

2.1.2.4 Pointer to Protected Memories Register File

Byte address offset 12 (0xC) embodies a pointer to the start offset where the protected memories of the TISS are mapped into the Control Interface's scope. Protected memories contain intrinsic information about time-triggered schedule and routing information, which is not in the sphere of control of the host. Consequently, they are read-only for the host but can be used to retrieve elemental information for specialized application software, e.g., diagnosis and statistics. Protected memories are described in detail in chapter 4.

2.2 Basic Communication Services

This section gives information how the functional specification of the basic communication services (CORE_1_COMM, CORE_2_COMM) is mapped to a design including a description of the operation of these core services.

2.2.1 Memory Layout of Ports

This section explains the memory layout of ports that are the endpoints of ACROSS' encapsulated communication channels. From the implementation point of view, a *port* in the context of ACROSS is a dedicated memory location that resides in the Port Memory which is a dual-ported on-chip memory. There are four parameters that characterize the memory layout of a port:

1. The *type* of a port determines which semantics the containing message carry and how many complete messages a port can contain at maximum. The allowed values are *periodic*, *sporadic, and conditional*. While (outgoing) periodic ports usually involve double buffering of (state) messages and therefore two complete images of the same entity, sporadic ports establish FIFO queues with configurable number of complete (event) messages. Conditional ports are a combination of periodic and sporadic ports, as they usually convey state messages with aperiodic, "on-demand" characteristics.
2. The *direction* of a port (i.e., *incoming* or *outgoing*) has also considerable influence on the memory layout.
3. The *size* of a message (given in data words) derives the number of data words in the Port Memory, which are arranged in a dense sequence and reserved to fill in application-specific data.
4. The *port base address* defines the starting point from where this dense sequence of data words begins in the Port Memory. For more details on the port base address refer to section 2.2.3.

As the size of a message has direct impact on the bandwidth of a communication channel, it is part of the time-triggered schedule and therefore not in the sphere of control of a host, but the Trusted Resource Manager (TRM). In other words, a host is not allowed to change the size of a message directly without involvement of the TRM. The same applies to the direction, which is also part of the time-triggered schedule.

However, the host controls the type and the port base address of ports. By setting the corresponding fields in the Port Configuration Memory of the TISS (see section 2.2.3), the host is able to relocate ports and perform reallocation of address ranges at run-time. In this context, **the host is in charge of the correctness of the allocation of ports** in the Port Memory, i.e., to prevent overlapping memory ranges of ports. Indeed, a disarrangement of ports can not corrupt the determinism of the basic communication services, but the transmitted data might be corrupted in the value domain.

2.2.1.1 Periodic Ports

Generally, we categorize the memory layout by direction of the port: incoming and outgoing. Considering the type "periodic" of outgoing ports, we introduce an additional quality of semantics, namely *explicit and implicit synchronization*, which is a dedicated field in the Port Configuration Memory (see section 2.2.3), and can be enabled and disabled by the host. Thereby, "synchronization" denotes the rules how to access to the Port Memory in order to achieve memory consistency among TISS and host for the messages in the Port Memory.

Outgoing periodic ports with explicit synchronization establish a *double buffer* mechanism to assure memory consistency. In this case, a port must have twice the size of the corresponding message reserved, whereas the data words assigned to both buffers are aligned in a dense sequence.

The usage of both buffers involves a mechanism of alternating active and shadow buffers. The active buffer contains the message that is ready to be sent by the TISS upon the next time-triggered send operation of the corresponding port. The host is not allowed to update data words in the active buffer, but in the shadow buffer. After the TISS has completely transmitted the message from the active buffer and the host has supplied a new message in the shadow buffer, the TISS toggles the roles of the buffers: active becomes shadow, shadow becomes active. Details of port synchronization are described in section 2.2.2.2.2.

Figure 3(a) illustrates the memory layout of an outgoing periodic port that uses explicit synchronization without time stamping. Note that all allocated data words of a given buffer can be used for payload data. By contrast, Figure 3(b) depicts the usage of time stamping for the port. Hereby, the very first two data words of the message (in each buffer) are reserved to house the time stamp so that the **net number of data words for payload data decreases by the size of the time stamp**. For the structure of the time stamp refer to section 2.5.5.

Figure 3(c) shows the memory layout of an outgoing periodic port that uses implicit synchronization (without time stamping[1]). In such a case, the port does not include double buffers so that the host is not allowed to modify data words in parallel to the sending TISS. Conversely, host and TISS are forced to temporally interleave their read/write accesses to the port in order to establish memory consistency of the contained message. To sum up, implicit synchronization is only applicable if creation and transmission of a message are properly aligned in the temporal domain. It is the responsibility of the application designer to assure this property in the time-triggered schedule as well as the task execution of the task that creates the message.

Incoming periodic ports use a single buffer so that the memory consistency must be guaranteed by a dedicated mechanism of synchronization, i.e., the Non-Blocking Write (NBW) protocol presented in section 2.2.2.2.1. As shown in Figure 4, ports with incoming periodic message also support time stamping.

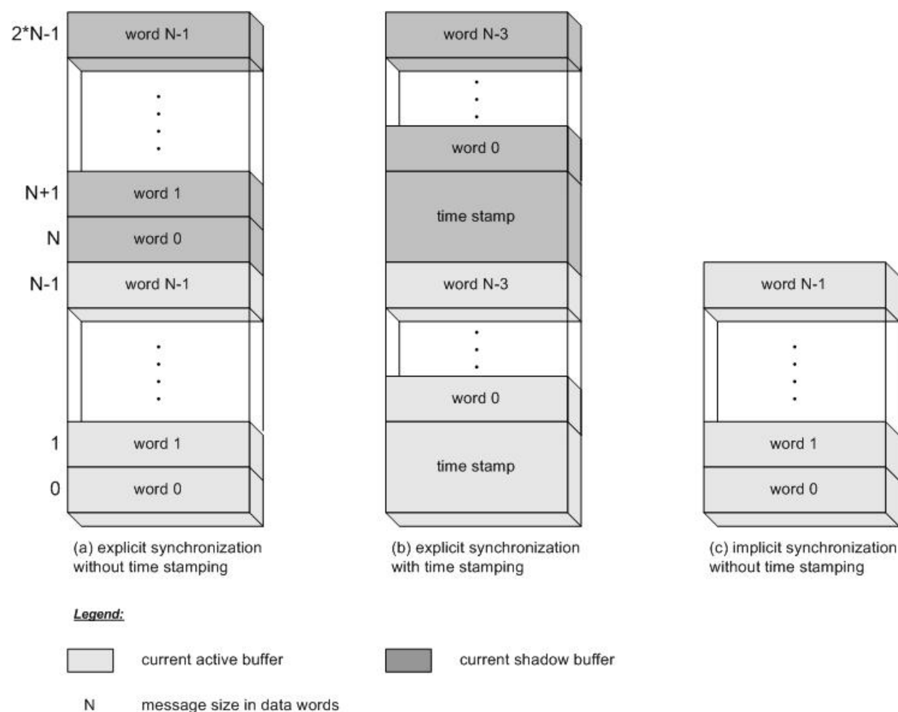

Figure 3: memory layout of outgoing periodic ports

2.2.1.2 Sporadic Ports

Unlike periodic ports that establish an update-in-place semantics, sporadic ports provide an exactly-once semantics by means of queuing of messages. While the size of a single message in the queue is predefined, the host is able to decide the length of the queue expressed by the maximum number of complete messages (see section 2.2.3). As we learn from Figure 5, the data words of all messages in

[1] Of course, ports with outgoing periodic messages and implicit synchronization also support time stamping.

the queue are arranged in a dense sequence. Like periodic ports, time stamps are appended in front of net payload data.

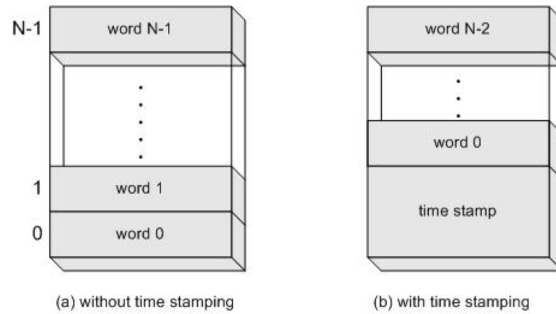

Figure 4: memory layout of ports with incoming periodic messages

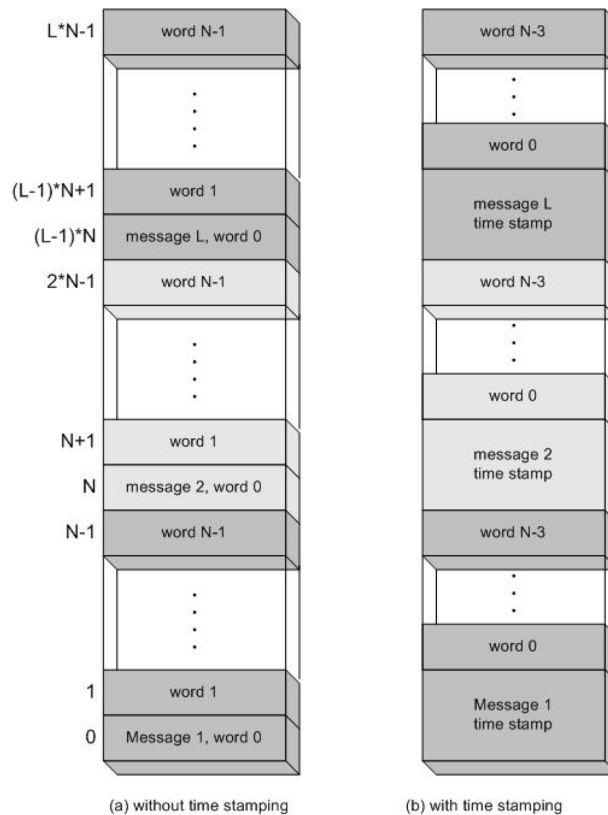

Figure 5: memory layout of sporadic ports

The queues of sporadic ports are realized as ring buffers, whereas the current read and write position in the queue is maintained. Messages are stored in the queue in FIFO style, and read accesses from the queue are consuming in order to model the exactly-once semantics of messages. A queue is used for any direction of a port, therefore outgoing and incoming ports have the same memory layout. Section 2.2.2.3 elaborates on the memory consistent synchronization of ports with sporadic messages in detail.

2.2.1.3 Conditional Ports

For conditional ports transmission of conditional messages is optional[2]. Although conditional messages convey state semantics, they are only transmitted when a new state message is available, which is a sharp contrast to periodic ports which always transmit the current value of a state variable.

For outgoing conditional ports the memory layout is the same as for outgoing periodic ports with explicit synchronization. Incoming conditional ports are equal to incoming periodic ports from the memory layout point of view. Nevertheless, a slight difference can be found in the port synchronization protocol (see section 2.2.2.4).

2.2.2 Memory-consistent access to Ports

As the host is able to arbitrarily access and modify the allocated data words of a port in the Port Memory, we identify the need of a mandatory synchronization protocol in order to assure memory consistency of messages. As long as TISS and host follow this synchronization protocol, we avoid malformed messages in the value domain. As the memory layout is different for each type and direction of port, there exist dedicated mechanisms of synchronization.

2.2.2.1 Port Synchronization Flags

The TISS contains a dedicated memory, the Port Synchronization Memory, which stores the state respectively progress of synchronization for each port. As we learn from section 2.2.4, the Port Synchronization Memory is accessible via the Control Interface in the address range of the basic communication service. Figure 6 shows the layout of a memory entry of the Port Synchronization Memory. Even though it is accessed through the 32-bit wide Control Interface, only 27 bits are occupied leaving 5 bits unused. In addition to this, only 24 of those 27 bits are really stored in memory, while the other 3 bits are only mapped into the data word of the Control Interface on each write access to the Port Synchronization Memory.

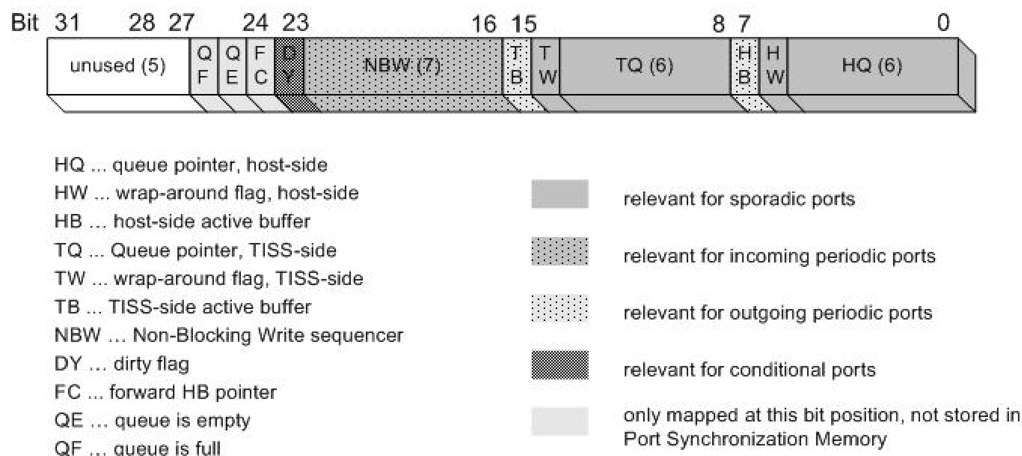

Figure 6: layout of a memory entry of the Port Synchronization Memory

As we can see in Figure 6, the given fields within a memory entry of the Port Synchronization Memory take the role of the synchronization flags for all types of synchronization. However, only one group associated with the current configuration of a given port is active at a given time.

[2] Conditional ports are supposed to realize APEX-style sampling ports.

As we will learn below, the TISS maintains the synchronization flags autonomously, while all write accesses to synchronization flags by the host are ignored as long as the associated port is unlocked in the Port Configuration Memory (by means of the field **PU = 1**). If the port is locked (**PU = 0**), a write operation (with an arbitrary value) to a given memory entry causes a clear of that entry, i.e., all bits are set to ‘0’. In this context, the host can not initialize a given memory entry in the Port Synchronization Memory with an arbitrary value, but just clear the whole entry. For example, this feature is used to reset the synchronization flags of a port when it is reconfigured from one type to another.

The role of each synchronization flag is described in the following sections where the synchronization protocol for each type of port is elaborated on.

2.2.2.2 Synchronization of periodic ports

Periodic ports are used for the periodic transmission of messages that usually incorporate state semantics. Therefore, we shall call them *state messages* from here on. State messages are instances of state variables, which contain the state of an observation of an object under consideration. A new version of the image totally substitutes the old version. Due to this “update-in-place” semantics, a periodic port holds exactly one valid image at a time. The challenge is to ensure that exactly one consistent image of that state variable is sent and received through the TISS, while the host might simultaneously update or inspect the image.

In section 2.2.1.1 the memory layout of periodic ports has been introduced. In the following section we describe the associated synchronization protocol.

2.2.2.2.1 Incoming state messages

For the synchronization of incoming state messages an incoming port employs the Non-Blocking Write (NBW) protocol. The purpose of NBW protocol is to detect a situation, when the TISS updates allocated data words of that incoming port in the Port Memory, while the host conducts a read access from the same port. The host might be forced to restart read operations, while the TISS never interrupts and also does not consider the presence of read operations. To sum up, the NBW protocol priories the producer (writer) of a message, which is the TISS in this case.

The NBW protocol is realized by means of a sequencer that is exclusively written by the TISS and read by the host. This NBW sequencer resides in the field **NBW** of a memory entry of the Port Synchronization Memory, as introduced in section 2.2.2.1. **NBW** is 7 bit wide and wraps around upon increment at value 0x7F, whereas increments of this sequencer can only be executed by the TISS.

At the beginning of the NBW protocol the sequencer is initialized with 0. When the TISS starts an update of the port according to the time-triggered schedule, it increments the sequencer. After completion of the update, i.e., the completion of the whole state message including the time stamp if activated for the given port, the TISS increments the sequencer again.

By definition of this synchronization protocol, the host has to combine a read operation from the port with a look-up from the NBW sequencer. If the value is odd, there is an update (write operation by the TISS) in progress, and the host has to retry access. It depends on the host whether it polls the NBW sequencer or it postpones the next look-up at an arbitrary time[3]. If the value is even, an update has been completed and the state message in the port can be regarded as consistent. Thus, the host is allowed to begin the read operation now.

At the end of the host’s read operation it has to check the value of the sequencer once more, whether the value has changed in the meantime. If so the TISS has just begun (if the new sequencer value is odd) or completed (if the new sequencer value is even again) the next update, while the

[3] This behaviour can be controlled by the „blocking mode“ parameter in the TISS driver functions.

host has been reading. Then, the host has to discard the previously fetched data and retry the read access in order to not retrieve inconsistent or corrupted data.

2.2.2.2.2 Outgoing state messages

As we have seen in section 2.2.1.1, outgoing periodic ports (i.e. outgoing state messages) reserve a double buffer in order to assure memory consistency of the transmitted state messages, if and only if explicit synchronization is used for that port.

The double buffers introduce a strategy of alternating active and shadow buffers. While one buffer holds a consistent, irrevocable state message, which can not be updated any more and is allowed to be sent by the TISS, the other buffer can be written by the host to place a state message (with a new version of the associated state variable) into the Port Memory. A given buffer (let's call them "lower buffer" and "upper buffer" with respect to the order of the start addresses of their memory locations) toggles its role in the synchronization protocol. Once that buffer holds the state message, which is to be sent by the TISS at the next time-triggered send operation of the corresponding port, it is regarded as the *active buffer*. After the send operation has been completed and the host has updated a new state message into the opposite buffer, it becomes the *shadow buffer* to pick up the next state message.

The information which buffer is the active and the shadow at a given time is derived from the synchronization flags **HB** and **TB** in that port's entry of the Port Synchronization Memory (see section 2.2.2.1). The flags **HB** and **TB** embody pointers to the buffer which is regarded as "currently active buffer" from the point of view of host respectively TISS. Table 1 summarizes the meaning of the values of these synchronization flags.

Value of HB | TB	Description
0	The lower buffer is the currently active one.
1	The upper buffer is the currently active one

Table 1: meaning of synchronization flags of outgoing periodic ports

Figure 7 shows a sequence of operation how these synchronization flags are used to control active and shadow buffers. At the beginning in Figure 7(a) the lower buffer is considered as active and has already been filled by the host with a state message. Therefore, **HB = 0** and **TB = 0**. At this time the host is allowed to create a new state message in the upper buffer that is shadow.

When the host has completed the new state message it forwards its pointer (**HB**) to point to the upper buffer (Figure 7(b)). Unless the TISS has not completed to transmit the previous state messages in the lower buffer, the host is not allowed to create the next state message in the lower buffer. Additionally, the "commit" of the new state message is **irrevocable**. In other words, the host is neither enabled to modify data in the new state message in the upper buffer, nor to set its pointer back to the lower buffer.

Figure 7(c) shows the pointers after the completion of transmission by the TISS. Upon next send instant, the TISS is going to fetch transmission data from the upper buffer with the new state message, which is regarded as active now. In this context, the lower buffer is shadow and the host is allowed to write to it.

Note that it is not necessary to create a new state message and therefore conduct a role switch of buffers after each send operation. If the host does not produce a new state message, both pointers remain unchanged and the TISS will continuously transmit the same state message (from the currently active buffer) until a new state message is available.

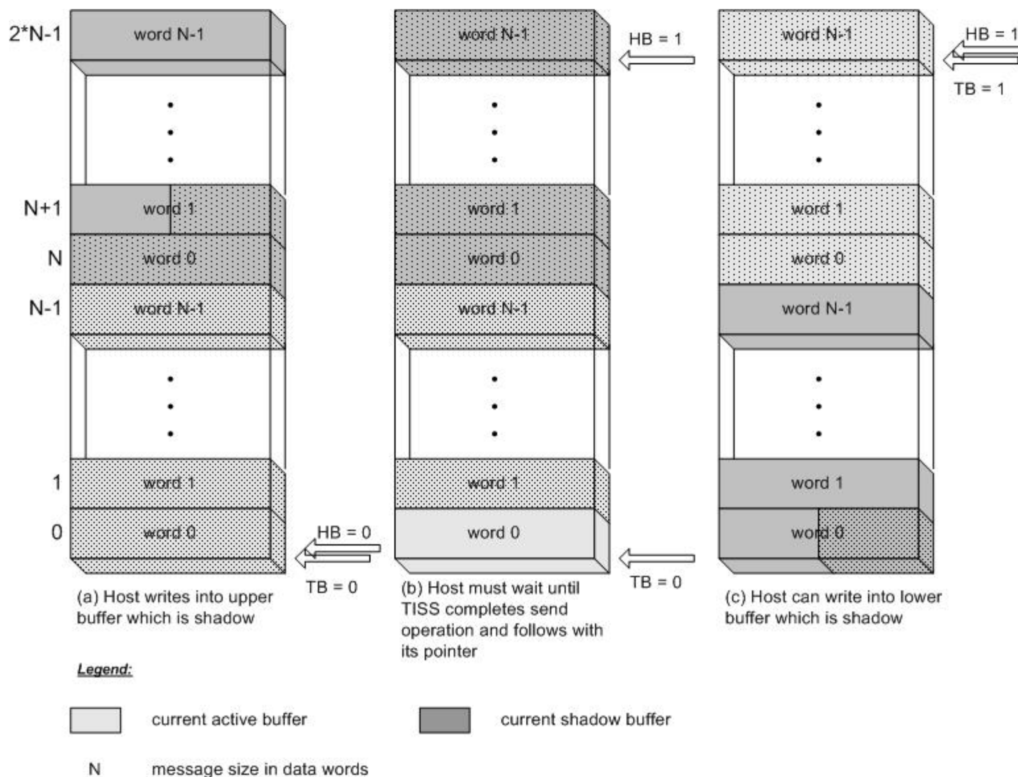

Figure 7: operation of synchronization flags of ports with outgoing periodic messages

The host forwards its pointer by writing a bit value of '1' at the bit field **FC** of the memory entry in the Port Synchronization Memory associated with the current port. Conversely, it is not allowed to write its pointer **HB** directly. The reason for this restriction is to avoid loss of synchronization in case of erroneous behaviour of the host when handling synchronization flags. Moreover, the TISS accepts this "forward command" if and only if there is no message transmission in progress. To be more precise, the "forward command" is refused if **HB** and **TB** are different like in Figure 7(b), but accepted in situations such as Figure 7(a) and Figure 7(c). If the TISS refuses a "forward command", the **HB** remains unchanged. It depends on the application software how to proceed with this situation, whether to poll the synchronization flags or to inspect them later at an arbitrary time and retry the "forward command"[4]. If the TISS accepts the "forward command", it toggles **HB** in the very next clock cycle after the arrival of that write command at the Control Interface.

Note that **FC** is not really stored in the Port Synchronization Memory, but just mapped to the corresponding bit position, whenever the host accesses the Port Synchronization Memory by a write operation at the Control Interface.

2.2.2.3 Synchronization of sporadic ports

As we have seen in section 2.2.1.2, sporadic ports incorporate a FIFO queue that is realized as a ring buffer in the Port Memory. As usual, this ring buffer entails a leading write position and a lagging read position. As such queues are used for incoming as well as outgoing sporadic ports, it depends on the direction whether the TISS drives the write or read position, while the host controls the respective counterpart.

For incoming ports, the TISS produces the messages due to the receive operation. Consequently, the TISS-side queue pointer which resides in the field **TQ** of the Port Synchronization Memory (see section 2.2.2.19) takes the role of the leading write position. On the contrary, the host-side queue

[4] This behaviour can be controlled by the „blocking mode" parameter in the TISS driver functions.

pointer in the field **HQ** of the Port Synchronization Memory embodies the lagging read position in this case. For outgoing ports, the roles of **TQ** and **HQ** are interchanged, because the host produces the messages and the TISS follows.

These queue pointers denote the numeric index of the current message in the queue, but not an offset from the port base address. On creation of a new event message the write position (and therefore the queue pointer that has this role) is incremented. On consumption of a message the read position the read pointer is incremented. As we can see in Figure 6, queue pointers are 6 bit wide. Consequently, a queue can embrace up to 64 event messages, whereas the numeric range of a queue pointer goes from 0 to 63. Because queue pointers can only be incremented, they will finally wrap around.

Without any additional information we would not be able to decide an empty or full queue, if the TISS-side and host-side queue pointers had the same value. As a consequence, the bit fields **TW** and **HW** in the memory entry of the Port Synchronization Memory are introduced to realize so-called "wrap-around flags" that are associated with TISS-side and host-side queue pointers accordingly. Whenever a queue pointer wraps around, the corresponding wrap-around flag is toggled. As a result, the fill state (empty or full) of the queue can be derived from the checks listed in Table 2. The conditions "queue empty" and "queue full" are mapped to the bit fields **QE** and **QF** in the Port Synchronization Memory, and reflect the checks described in Table 2. However, **QE** and **QF** are not really stored in the Port Synchronization Memory.

queue fill state	Check
empty	$\mathbf{QE} \leftarrow \mathbf{TW} = \mathbf{HW} \wedge \mathbf{TQ} = \mathbf{HQ}$
full	$\mathbf{QF} \leftarrow \mathbf{TW} \neq \mathbf{HW} \wedge \mathbf{TQ} = \mathbf{HQ}$

Table 2: checks to decide empty/full queues

Similar to the port synchronization of outgoing periodic ports, the host is unable to write its associated synchronization flags, i.e., the queue pointer **HQ** and the wrap-around flag **HW**. As a consequence, to increment its queue pointer, it must write a bit value of '1' to the bit field **FC** of the memory entry in the Port Synchronization Memory of the corresponding port. Provided that the TISS accepts this "forward command", the TISS increments **HQ**, simultaneously considers its wrap-around and if so also toggles **HW**. However, the TISS refuses the "forward command" in the following situations.

- for outgoing ports if the queue is full
- for incoming ports if the queue is empty

In such a situation, the TISS simply ignores the "forward command" and **HQ** and **HW** remain unchanged.

The TISS maintains its queue pointer and associated wrap-around flag autonomously. For incoming ports, it increments **TQ** on completed reception of an event message, i.e., in the very next clock cycle after it has written the very last data word of the event message (including time stamps if activated for the given port) into the Port Memory. For outgoing ports, it increments **TQ** on completed transmission of an event message, i.e., in the very next clock cycle after it has fetched the very last data word of the event message from the Port Memory. The wrap-around flag is maintained simultaneously.

In the following we describe the actions undertaken upon arrival of new messages in the queue fill states "empty" and "full". For outgoing ports, if a send operation is dispatched and the queue is empty, the TISS logically skips a transmission as there is no message to be sent. For the host an empty queue has no impairment, too, because it is able to deposit at least one message. However, in

case of a full queue, any attempt by the host to create a new message in the queue is rejected, and overwriting existing messages is not supported. This policy entails a mechanism of back-pressure, and it is up to its application software, how to respond to this situation. For instance, it might busy-wait (by means of polling the port synchronization flags) until the TISS has processed the first message in the queue so that there is space for a new one[5]. Another example is to report this incident to some other authority by means of a dedicated communication channel, e.g., to the diagnostic unit, or to request more bandwidth.

For incoming ports, an empty queue also means no critical condition. There is sufficient space for the TISS to deposit new arriving messages. As mentioned above, the host also has the possibility to busy-wait (by means of polling the port synchronization flags) until a new message has arrived. Similar to outgoing ports, full queues are also problematic for incoming ports. In this case, the TISS has no space to put an arriving message into the queue. Even worse, the TISS is forced to discard that message, because it must not busy-wait until the host has consumed at least one message. Otherwise it would have missed the dispatching of later communication activities, thus breaking the property of encapsulation and error containment of communication channels. The discarding of messages implies a loss of information, moreover a violation of the exactly-once semantics associated with sporadic ports. As a consequence, the TISS raises an error that is recorded by the error reporting core service (see section 2.7).

2.2.2.4 Synchronization of conditional ports

Conditional ports reuse the mechanisms of periodic ports. Outgoing periodic ports establish double buffering which is controlled by the fields **HB, TB** and entails the same synchronization protocol as described for outgoing periodic ports with explicit synchronization (see section 2.2.2.2.2).

In addition to this, the TISS maintains the *dirty flag* **DY** in a memory entry of the Port Synchronization Memory. The dirty flag indicates that the host has successfully submitted a new state message to the shadow buffer (which has become the active buffer with the "forward command" in the mean time). **DY = 1** at the very next clock cycle after the TISS accepts the "forward command" of the host (by writing a bit value of '1' to the field **FC**). If the TISS rejects the "forward command" under the same circumstances as for outgoing periodic ports, **DY** remains 0.

On the next send instant associated with that port, the TISS checks the memory entry of the Port Synchronization Memory, whether **DY = 1**. If so, it starts transmission. If **DY = 0** it skips the send operation, which is no erroneous condition in this case. At the very next clock cycle after complete transmission of the state message from the active buffer, the TISS sets **DY = 0** and the fields **HB**, **TB** change values according to the port synchronization of outgoing periodic ports.

For incoming conditional ports the same NBW sequencer is employed as for incoming periodic ports (see section 2.2.2.2.1). Additionally, the TISS sets **DY = 1** on complete arrival of an incoming conditional message. As a result, the host is able to notice the arrival of a new state message in the incoming conditional port by just examining **DY = 1**.

To indicate the processing of a new state message and to identify the arrival of the next new message, **DY** must be set 0. However, write access to the Port Synchronization Memory is not possible as long as a port is unlocked, so the host can not clear **DY** with reasonable effort. For this purpose, if **DY = 1** in the memory entry of the Port Synchronization Memory of an incoming conditional port, the read operation of that memory entry by the host automatically clears **DY = 0,** and the state message is regarded as processed. In other words, read operations are destructive for **DY**. For the read-out of the state message from the Port Memory the same rules apply as for incoming periodic ports, namely the host has to monitor the NBW sequencer for changes to reason about data consistency of the retrieved data.

[5] This behaviour can be controlled by the „blocking mode" parameter in the TISS driver functions.

If still **DY = 1** when a new state message arrives, **DY** remains at this state and the associated memory location in the Port Memory is overwritten as it is the case for incoming periodic ports. The incremented, odd NBW sequencer reflects the update in progress.

2.2.3 Port Configuration

As mentioned in section 2.2.1, a port is characterized by a set of parameters, whereas some are in the sphere of control of the TRM, i.e., direction and message size of a port. Others can be configured by the host, i.e., port type, port base address. Later sections will introduce additional features of ports that can be organized by the host, for example queue lengths of sporadic ports, explicit / implicit synchronization etc.

The place where the host stores all its configuration of ports is the Port Configuration Memory, which is a dedicated memory within the TISS like the Port Synchronization Memory. Figure 8 illustrates the layout of an entry in the Port Configuration Memory. In the following we explain each field of such an entry.

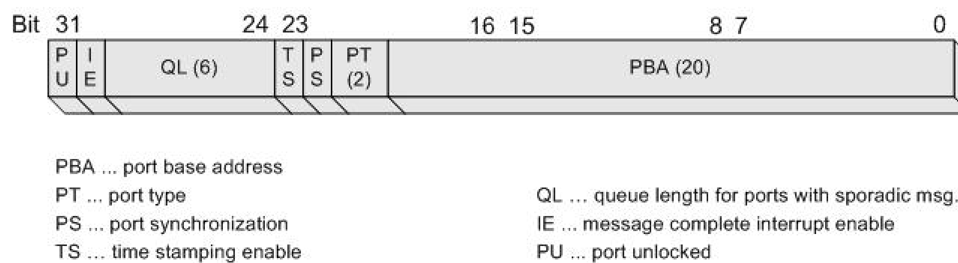

Figure 8: layout of a memory entry of the Port Configuration Memory

The port base address of a port is stored in the field **PBA**. Note that the port base address is given in words of the Port Memory, so this is a **word address**. As **PBA** is 20 bit wide, the TISS can address 2^{20} words. In case of a 32-bit wide Port Interface this yields a total address range of 4 MB. From the point of view of the memory-mapped master attached to a read/write port of the Port Memory the port base address denotes an **absolute byte address** within that memory-mapped master's address range. The TISS' wrapper that translates the native UNI to semantics of the target system interconnect, e.g. AVALON, converts the word address of **PBA** into a byte address. For matters of simplicity it is recommended that the Port Memory maps into the same address range at all memory-mapped masters to avoid error-prone address translations.

The host sets the type of a port, whether periodic, sporadic, or conditional, by means of the bit field **PT**. Table 3 lists the possible encoding of the port type in the field **PT**.

PT(1:0)	Description
00	invalid; port not configured properly (see section 2.7.1)
01	periodic
10	sporadic
11	conditional

Table 3: encoding of port types

The style of synchronization, i.e., explicit or implicit, which is only relevant for outgoing periodic ports, is expressed in the field **PS**. Thereby, a value of **PS = 1** means implicit synchronization, and **PS = 0** is explicit synchronization.

If the application software intends to use a given port with the time stamping feature, **TS** must be set to 1, 0 otherwise. For details on time stamping refer to section 2.5.5.

The field **QL** holds the queue length of sporadic ports as described in section 2.2.2.3. It denotes the maximum number of complete messages that fit into a sporadic port's queue. As this field is 6 bit wide, such queue supports up to 64 complete messages.

If the TISS shall raise an interrupt whenever a complete message has been sent or received for that port, the host sets **IE = 1**. That "message complete interrupt" is deactivated if **IE = 0**. For details on message complete interrupts refer to section 3.3.1.

Finally, the host is able to suppress communication activities by means of the bit field **PU**. On **PU = 1**, communication activities of the given port are unlocked, otherwise deferred on **PU = 0**. Note that the TISS might still dispatch communication activities if **PU = 0**, however it skips any transmission or receive operations. This implies that the port synchronization flags do not change, and no data is read from or written to the reserved memory location in the Port Memory of the given port. Disabling ports is necessary if the host wants to reset the port synchronization flags in the Port Synchronization Memory, as mentioned in section 2.2.2.1.

The parameters of a port, which can not be modified by the host, are collected in another dedicated memory of the TISS – the Port Parameters Memory. From the point of view of the host it is read-only. Consequently, the Port Parameter Memory is regarded as "protected memory" (see chapter 4).

2.2.4 Register File of the Basic Communication Services

The register file of the basic communication services blends in the memories mentioned in the above sections, i.e., the Port Configuration, Port Synchronization, and Port Parameter Memory. Figure 9 shows the address offsets where those memories are located within the address range of the register file of the basic communication service.

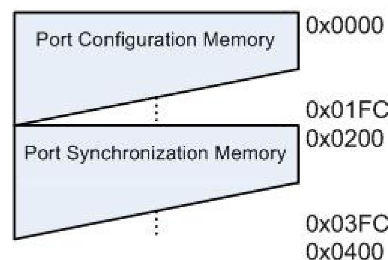

Figure 9: layout of the register file of the basic communication services

2.3 Component Control Service

This section describes the realization of the component control service (CORE_1_EXEC), which is used to control the operation of the host in a component. Component control allows starting, stopping, or resetting the execution of tasks or groups of tasks running in the host.

2.3.1 Component control channels

The commands of component control are distributed in messages of dedicated communication channels - we shall call these channels *component control channels* from here on. A message transmitted in a component control channel is called a *command message*.

Unlike "ordinary" communication channels of the basic communication services, each command message is preprocessed within the receiving TISS to derive the intended behaviour of component control. Additionally, the whole command message can be stored in the Port Memory (if the port is

unlocked via **PU = 1**) according to the configuration of the associated port (of the component control channel) in the Port Configuration Memory.

As component control involves **sporadic messages** as introduced by the basic communication services, the associated communication activities manifest in the time-triggered schedule. The route determines the source and sinks of component control channels. The source is the producer of the command messages in this case. From the source's point of view, the component control channel embodies an "ordinary[6]" communication channel accessible by an outgoing sporadic port, which is subscribed by the communication service. Consequently, at the source there is no preprocessing of the command messages involved.

When it comes to the configuration of the port at each sink of the component control channel, it must be configured as "ordinary" incoming sporadic port according to section 2.2.3. Also time stamping, message complete interrupts and port locking are supported. However, that port is subscribed by the component control service and not by a communication core service, which solely manifests in the Port Parameter Memory. For details about subscription of ports refer to section 4.1.1.

If that port is locked (**PU = 0** in the Port Configuration Memory), the command message is discarded, i.e., not stored in the Port Memory. However, it is still preprocessed so that the *reset* commands of component control can take effect. In other words, it is not possible for the host to evade component control (in particular the *reset* commands) by locking the port subscribed by the component control service. Certainly, for the application software it does not make any sense to throw away command messages, because it can no longer be notified of *start* and *stop* commands of task and task groups. The purpose of this behaviour is to prevent that a malicious host would avoid being reset by the according *reset* commands, if these command messages were not preprocessed due to a locked port.

2.3.2 Roles of Component Control

The producer of the command messages need not necessarily be a trusted component like the TRM. For each component control channel it is feasible to assign an arbitrary component as the source and therefore the producer of the command messages. As a consequence, that component is enabled to launch component control commands at the sink-side component. So, a given component prevails control of some other component without interaction of the TRM. An example of such a "dominant" component is the Diagnostic Unit, which might reset or disable a component after it has evaluated it as faulty and needs to be restored. Another example is the memory component that houses the boot master of the basic boot service, which tells all components to quit the boot loading mode and resume normal operation with application tasks after the boot process has been completed.

In this context, it is possible to have exactly one "component control master" component which is the source of all component control channels of all other components. By contrast, we could also built local groups of master and slave components, whereas the topology of the component control channels determines the role of an involved component. It is the decision of an application designer to assign master and slave roles to components with respect to component control.

2.3.3 Command Messages

In the following we introduce the structure of command messages. Command messages are wrapped in sporadic messages of the component control channel with a fixed size of 8 32-bit data words, whereas up to 6 data words (without time stamping) respectively 4 data words (with time

[6] However, section 2.3.3 points out that time stamping is not reasonable at the source-side of the component control channel.

stamping) can be used to convey application-specific parameters. Figure 10 shows the sequence of data words that make up a command message (without time stamp).

The very first data word contains a set of fields containing metadata that describes the content of the command message. Note that a time stamp is appended before the very first data word in the Port Memory of the sink of a component control channel so that the net payload of the command message, i.e., the application specific parameters, decreases by the size of the time stamp, as illustrated in Figure 11.

For command messages, **time stamping only seems to be reasonable at the sink side of the component control channel**, where the corresponding port is configured as incoming sporadic. If the default way of time stamping was also activated at the source side, the TISS would interpret the source's time stamp as very first payload data word containing the metadata, which might cause an undefined state of the component control service.

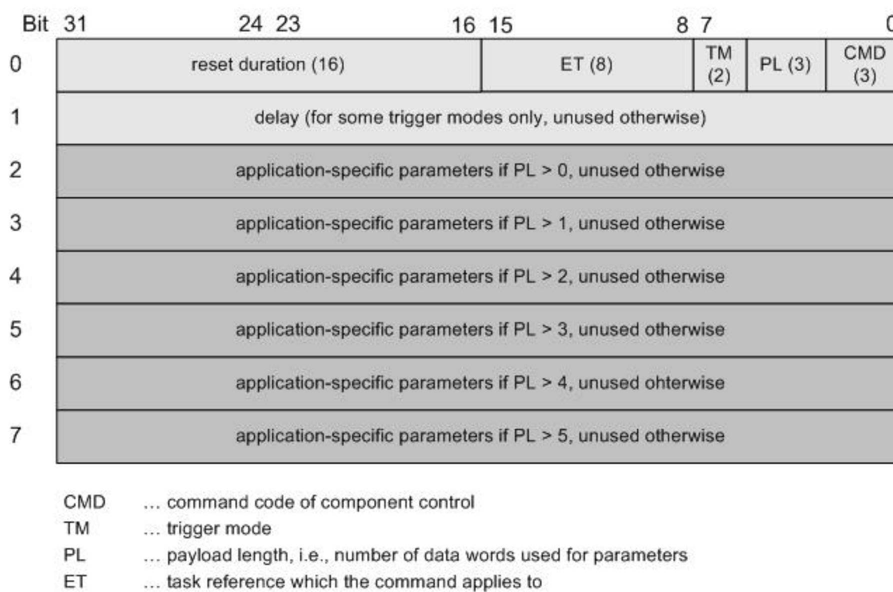

Figure 10: structure of a command message (without time stamp)

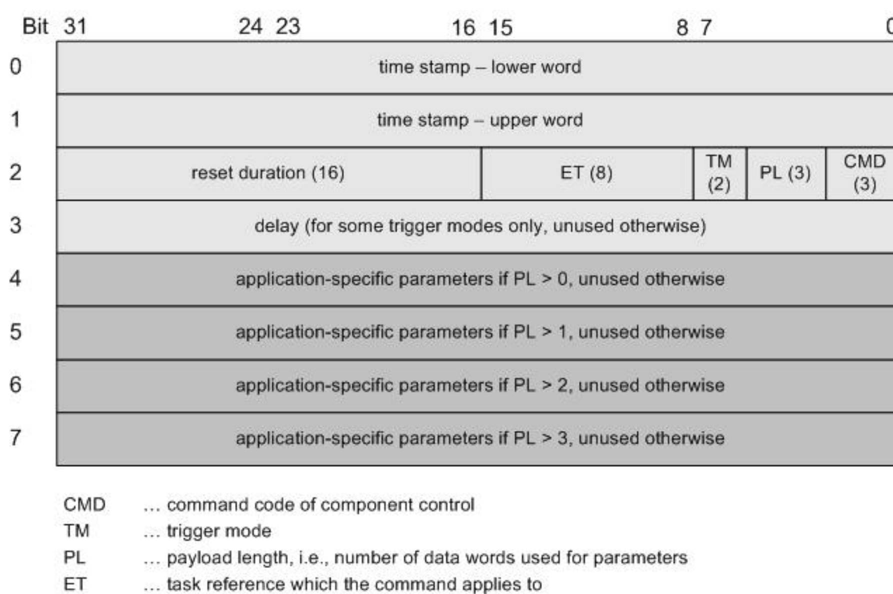

Figure 11: structure of a command message (with time stamp)

2.3.3.1 Command Code

The field **Cmd** contains the binary code of the actual command of component control. As it is 3 bit wide, we can express up to 8 different commands. Table 4 lists the defined commands.

Cmd(2:0)	Description
000	empty command: no command, but parameters in the message
001	*start* command: the task / task set given in **ET** is supposed to be started
010	*stop* command: the task / task set given in **ET** is supposed to be stopped
011	not used yet
100	*reset* command: the host is reset in the regular way
101	*hold reset* command: the reset wire is permanently kept in reset state
110	*release reset* command: the reset state of the reset wire is revoked
111	not used yet

Table 4: command codes of execution control

The *start* and *stop* commands are transparent to the TISS. The reason for this is that the TISS can not directly control the execution of the host, e.g., manipulating its program counter. It is only able to indicate the activation of such a command message by means of the *component control interrupt*. In fact, the host is in charge of controlling the execution of its own tasks. Usually, the interrupt service routine associated with the component control interrupt evaluates the first data word of the command message and uses the metadata as input for some task scheduler of the host's local operating system. From the point of view of the TISS, the effect of the *start* and *stop* commands is component-specific and beyond its scope, because it requires a local software part to realize the actual execution control of tasks.

The *reset* command is meant to force the execution of program code starting from the CPU's reset vector. For this purpose, the TISS drives a level-sensitive "reset wire" from the Control Interface to the host. The *reset* command causes the "reset wire" to be driven active for as many clock cycles (of the TISS' system operation frequency) as given by the 16-bit field **reset duration+1** of that command message. Note that the component control interrupt is not reasonable in case of *reset* commands (as well as *reset hold* and *reset release*) and therefore not realized. Namely, the host is not supposed to service a *reset* command in an interrupt service routine, but to restart program execution.

Contrary to the *reset* command, which holds the "reset wire" active for a specified time, the *reset hold* command permanently drives the "reset wire" active. As a consequence, the host can not recover from the reset state and is restrained from starting off with program execution at the reset vector. In other words, *reset hold* causes the host to be turned off by force - so to say; it is the "kill switch" of the host. Such a measure might be necessary if the host behaves maliciously or faulty and proper operation can not be restored any more.

The counterpart of *hold reset* is *release reset*, so that the TISS drives the "reset wire" in the inactive level. As a result, the host is able to recover from the reset state and resume execution of program code at the reset vector.

2.3.3.2 Payload Length

The field **PL** in the first data word of a command message denotes the number of data words that are used to convey application-specific parameters. Table 5 describes the meaning of all possible values of **PL**.

PL	Description
0	the command message uses no corresponding data words for parameters
1 - 6	as many data words as given by **PL** are occupied in a dense sequence
7	The maximum number of data words (=6) are used for parameters, additionally a subsequent command message delivers even more parameters which logically belong together.

Table 5: range of the payload length field in a command message

While the values [0,6] of **PL** are obvious, but **PL = 7** is reserved for the concatenation of parameters among several command messages. A usage scenario for this is a series of “empty” command messages (**CMD = 000**) which contain application-specific parameters, which is terminated by a command message with some *start* or *stop* command and the last set of parameter data words.

Note that the presence of application-specific parameters is transparent to the TISS. On retrieval of the command message from the Port Memory, the host has to process these parameters. The field **ET** is used to relate these parameters to a task or task set.

2.3.3.3 Trigger Modes

As we do not recommend the usage of polling of command messages for the component control service, there must be interrupts involved. A command message can provoke two interrupts:

1. As the ports associated with component control channels can use message complete interrupts, the host might notice such an interrupt on arrival (and storage in the Port Memory) of the command message.
2. If the message complete interrupt is not appropriate for a given application scenario, the dedicated *component control interrupt* indicates the activation of the command message that contains a *start* or *stop* command.

For the latter case the producer of a command message has the choice, when this component control interrupt should be raised after arrival of the command message. This choice is called the *trigger mode* of the command message and is encoded in the field **TM**. Table 6 lists the encoding of trigger modes.

TM(1:0)	Description
00	immediate effect
01	delay by period and phase
10	delay by number of macro ticks
11	delay by number of clock cycles of the system operation frequency

Table 6: Trigger modes in the header of a command message

The trigger mode defines the temporal relation between arrival of a command message and indication of the component control interrupt. A trigger is applied for all types of commands. For *start* and *stop* commands the trigger defines a immediate response or delay, when the component control interrupt is raised so that the host reacts on the request for starting or stopping the task or task set given in the field **ET**. For all reset commands (*reset*, *reset hold*, *reset release*) the trigger mode specifies the immediate or postponed driving of the “reset wire”.

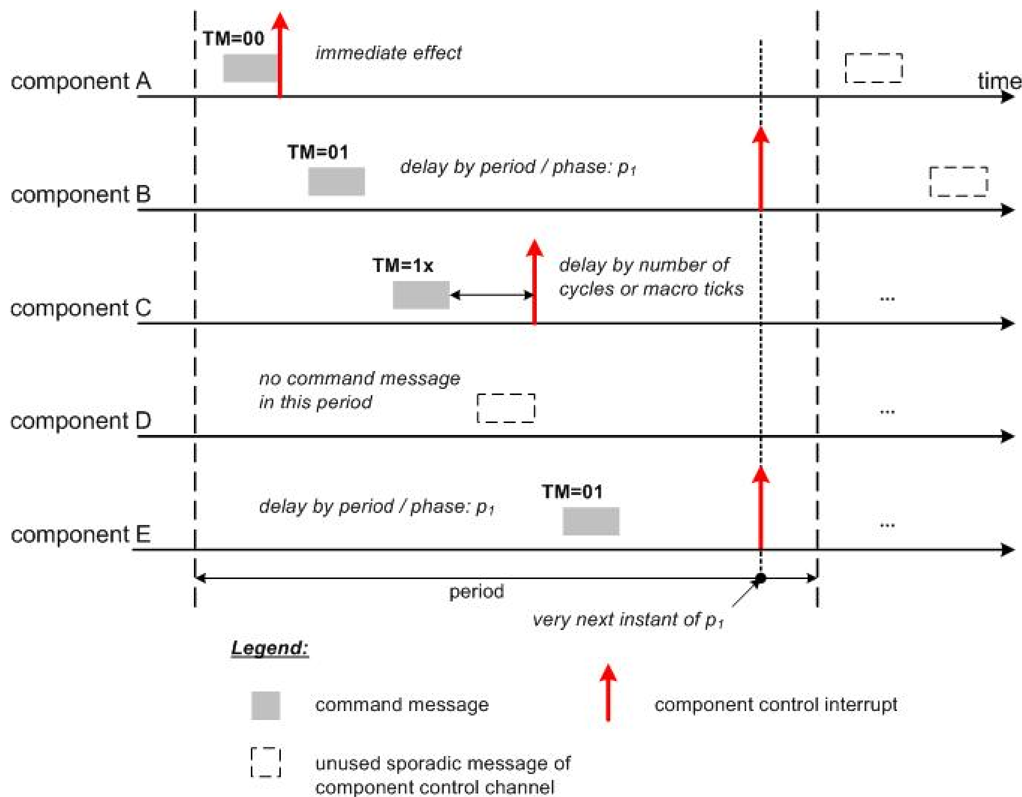

Figure 12: trigger modes of component control

As illustrated in Figure 12, the trigger mode **TM = 00** causes the generation of the component control interrupt respectively the driving of the “reset wire” immediately after the complete reception of the command message. In this case, if the message complete interrupt was used, it would coincide with the component control interrupt.

The trigger modes **TM = 10** and **TM = 11** incorporate a delay to the action, which is specified in the least significant 24 bit of the **delay** data word. The encoding of **TM** defines the quantifier, whether **delay** denotes number of clock cycles of the system operation frequency or macro ticks. After elapse of **delay+1** clock cycles respectively macro ticks, the TISS generates the component control interrupt for *start* and *stop* commands or drives the reset wire in case of some reset command.

Similar to the above trigger modes, **TM = 01** refers to a delay that is expressed in the notion of period and phase like a time stamp (see section 2.5.5). In this case, the action is performed when the overflow counter of the period referenced in the **delay** data word matches the phase in **delay** at the very next instant after arrival of the command message. This trigger mode can also be used to establish a common instant across several components to perform a given action synchronously – the so-called *component control instant*. For this purpose, command messages arriving at different components have to contain the identical values for period and phase, as illustrated in Figure 12. Note that the period referenced in **delay** need not necessarily be the same period the command message has been sent with.

2.3.3.4 Task Reference

The field **ET** denotes the so-called *task reference* of a given command message. This is an 8-bit wide numeric descriptor of either a plane numeric task identifier or a numeric descriptor of an application-specific data structure that describes sets of tasks. For instance, we define the value **ET = 255** as the task set that spans all tasks of a given component. As a consequence, a command message with **Cmd = 001** and **ET = 255** literally means, "start all tasks on that component". An example usage for such a command message is the mode switch from boot mode to normal mode at the end of the boot process.

2.3.4 Register File of Component Control Service

The component control service has no explicit register file.

2.4 Task Trigger Service

This section describes the realization of the task trigger service (CORE_2_EXEC).

If we assume the basic communication services and their ability to raise the message complete interrupt, it is obvious that an application designer might place application code in the callback mechanism respectively interrupt service routines of that interrupt source. Consequently, we achieve some kind of periodic activation of that application code, whereas the activation directly correlates to some communication activity. However, it might be desirable for the application designer to have application tasks triggered that do not necessarily correspond to any communication activity, for instance a task performing diagnosis and statistics or a maintenance task of an operating system. For this purpose, the task trigger service facilitates the triggering of periodic tasks.

2.4.1 Realization of Task Triggers

The triggering of periodic tasks uses the same dispatching circuitry in the TISS as the basic communication service. In this context the periodic instant of triggering manifests in an entry of the time-triggered schedule. In Figure 21 we see the fields in a memory entry of the time-triggered schedule that are shared among triggering of periodic tasks and dispatching of ordinary communication activities. The bit **T** decides whether a given memory entry embodies the dispatching of a communication activity or a task trigger. If **T = 1**, that memory entry is associated with the triggering of a task. In such a case, the field **Dispatching ID** is interpreted is the *numeric identifier of the task to be executed*. As **Dispatching ID** is 8 bit wide, we can distinguish 256 different tasks by number.

From the implementation point of view, dispatching of an entry in the time-triggered schedule associated with a periodic task causes the TISS to raise the "*task trigger interrupt*". Hence, the periodic task trigger incorporates the interrupt mechanism of the TISS to provoke the execution of application code in the host. In case of a task trigger interrupt, the currently activated task is kept in the dedicated bit field **CT** in the register file of the task trigger service in order to inform the host which task has to be executed. As illustrated in Figure 13, the bit field **CT** holds the value of the field **Dispatching ID** of he corresponding memory entry in the time-triggered schedule of the current task. Usually, the host evaluates **CT** in the interrupt service routine of the task trigger interrupt. For example, it forwards this parameter to the scheduler of its local operating system or directly executes application code in the interrupt service routine.

Like communication activities in the time-triggered schedule, the periodic activation of tasks is defined a-priori and conducted autonomously without set-up by the host. Furthermore, as the host

is not able to modify the time-triggered schedule, it is not able to add or remove task activations without interaction of the TRM. The host only has the possibility to suppress the raising of the "task trigger interrupt" by masking this interrupt source. Details about the interrupt and callback mechanism can be found in section 3.4.

2.4.2 Register File of the Task Trigger Services

Figure 13 depicts the layout of the register file of the task trigger service.

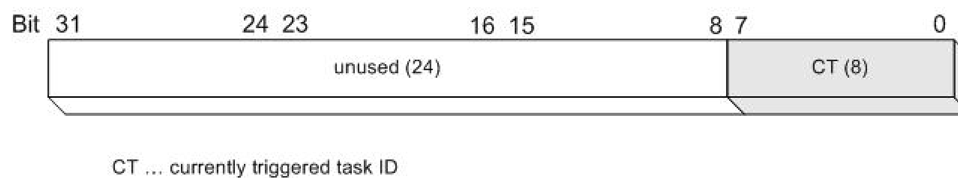

Figure 13: layout of the register file of the task trigger service

2.5 Common Time Service

This section deals with the implementation of the common time service (CORE_1_TIME) from the point of view of a host and supplements the functional specification[7]. Therefore, it describes the interface of the common time service as it is accessible by a host. Additional aspects of that core service mentioned in its functional specification, which are not visible at the local interface such as the external clock synchronization, are covered in dedicated sections.

2.5.1 Realization of Periods

An ACROSS MPSoC supports up to 8 distinct periods which do not possess any numeric relation to each other except the macro tick as a common denominator. Each period can be expressed as an integer multiple of the macro tick.

A TISS maintains an overflow counter for each period that counts the number of macro ticks. Consequently, a given TISS is attached to a chip-wide global clock distribution network which drives the macro tick. On each rising edge of the macro tick, the overflow counters of each period in a TISS are incremented simultaneously. Furthermore, the overflow counters of each period of each TISS execute this increment simultaneously.

To measure the elapsing of a period, a top value for such overflow counter must be defined. On compare match between the overflow counter and the top value, the intended duration associated with the period has passed by. Additionally, the counter is wrapped around and starts over from value 0.

From the point of view of the host, the whole process of realization of periods runs autonomously. In fact, it has no means of influence, but it has the possibility to observe the progress of overflow counters including the top values. For this purpose, the register file of the common time service provides images of the current value of the overflow counter and the top value of each supported period, as illustrated in Figure 14.

While the increment of overflow counters is implemented within the TISS and runs autonomously, the top values must be supplied to achieve the above behaviour. However, as the common time service is not in the sphere of control of a host, it does not have write access to the registers holding the top values. As a consequence, the TRM has to assign values to the registers corresponding to an

[7] refer to D1.2 for further information

overflow counter's top value. For this purpose, the TRM leverages the reconfiguration protocol defined by the inter-component channel configuration.

2.5.2 Duration of Periods

As the top value associated with an overflow counter combined with the granularity of the macro tick specify the duration of a period, the higher the top value the longer the period. As a consequence, the longest possible period depends on the maximum value that can be stored in the top value register and is therefore a function of the width of that register.

In Figure 14 we can see that the registers of the top values span 24 bit so that an overflow counter of a period can count up to 2^{24} macro ticks. If we assume a (default) macro tick granularity of 100 ns, the longest possible period is 1.678 seconds. Some top value register CNTTOPn has a valid integer range of [0, 2^{24}-1] so that the compare match occurs after CNTTOPn+1 macro ticks.

The granularity of the macro tick can be retrieved from a dedicated field in the register file of the common time service. Similar to the top values, this value is controlled by the TRM in order to inform the host about this parameter.

As the host accesses the register file via the 32-bit wide Control Interface, additional information is necessary to correctly interpret how many bits of such a data word are really occupied by top value registers. For this purpose, the register file of the common time service contains the field **VW** which is 5 bit wide and can theoretically span a whole 32-bit wide data word of the Control Interface. The value **VW+1** denotes the number of bits occupied in some register CNTTOPn as well as CNTVALn. In case of Figure 14 **VW = 23**.

2.5.3 Active Periods

From the point of view of the host, the common time service has no direct utility. However, the common time service is the basis for other core services, in particular the basic communication services. As communication activities are defined in the time-triggered schedule, the notion of periods and the phases within periods manifest there. The configuration of the time-triggered schedule is application-specific, and there might be applications, where not all periods provided by the common time service are used.

For this purpose, the TRM has the ability to selectively activate and deactivate periods. If a period is deactivated, the overflow counters do no longer increment on rising edges of the macro tick. In this context, the TRM maintains a bit vector in the register file of the common time service (in each TISS), which periods are currently active. The host can find this information in the field **VAP**. The least significant bit of **VAP** maps to period 0, while the most significant bit embodies period 7. If a given bit in **VAP** is 1, the corresponding period is active.

Additionally to **VAP**, the register file contains the number of all currently active periods in the field **AP**, which is 3 bit wide. **AP** has a valid range of [0, 7], whereas **AP+1** periods are declared as active. This field is a shortcut to avoid counting each set bit in **VAP**.

Note that **at least one period must be active in the time-triggered schedule at any time**, even if **AP=0** (according to the formula **AP+1**). The reason for this is that each configuration of the time-triggered schedule must contain critical communication channels that uphold the operation of core service, e.g., the communication channels of inter-channel component configuration, component control, and basic boot service. As the communication activities of these special communication channels also manifest in the time-triggered schedule, at least one period must be active so that these communication activities can be dispatched.

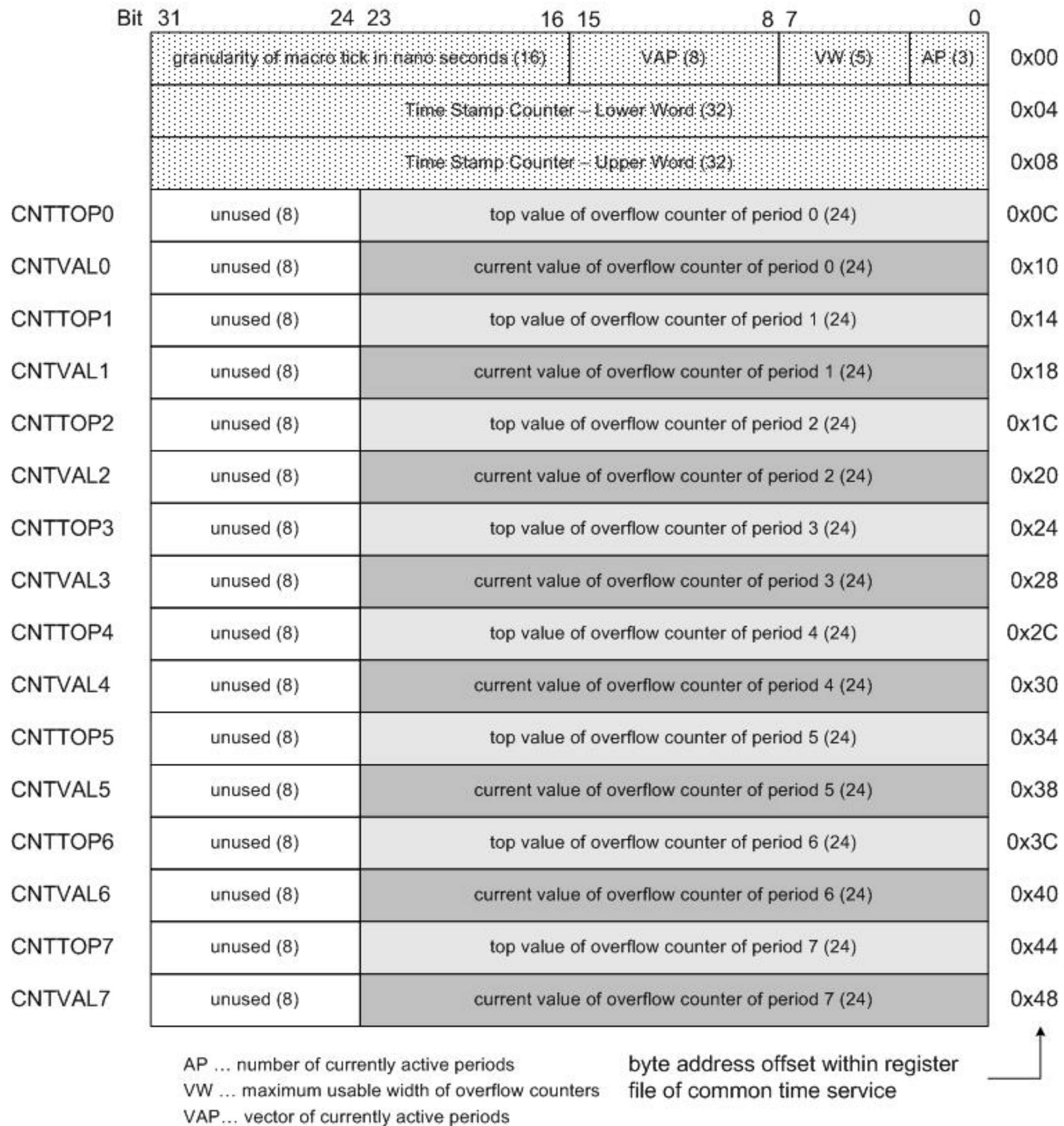

Figure 14: layout of the register file of the common time service

2.5.4 Register File of the Common Time Service

In fact, the host has no control of the operation of the common time service. It can only retrieve information as mentioned in the above sections. As a consequence, all registers illustrated in Figure 14 are read-only. Write operations by the host are ignored.

2.5.5 Time stamps

Some applications require information about the instant when a message has been produced respectively received. The time stamp records two incidents of message transmission:

- for incoming messages the time stamp coincides with the storage of the very last data word of a complete message in the Port Memory
- for outgoing messages the time stamp denotes the instant of submission of a complete message into the Port Memory, i.e., the message is declared as completed and ready for transmission

Depending on the direction of the port and therefore the message, the producer of the time stamp is either the TISS (incoming) or the host (outgoing). The time stamp is the copy of the current value of the time stamp counter at the time of creation. This dedicated time stamp counter is 64-bit wide

and incremented on each rising edge of the macro tick. At power-up it initializes with all bits ‘0’. It can be read from the register file of the common time service as illustrated in Figure 14. A read access to the time stamp counter must always be conducted in the sequence lower word, then upper word, as on access to the lower word the upper word is buffered transparently to assure a consistently read value.

The producer of the time stamp appends the time stamp in front of the very first payload data word of the message split into two 32-bit words as shown in Figure 15. As a consequence, the time stamp decreases the space for net payload of a message by two data words. Note that the total size of a message is defined by the **Size** field in the Port Parameter Memory and can not be extended dynamically by activation of time stamping for a port. Consequently, the presence of a time stamp should be considered in advance during application design so that it is taken into account in the **Size** field of the Port Parameter Memory.

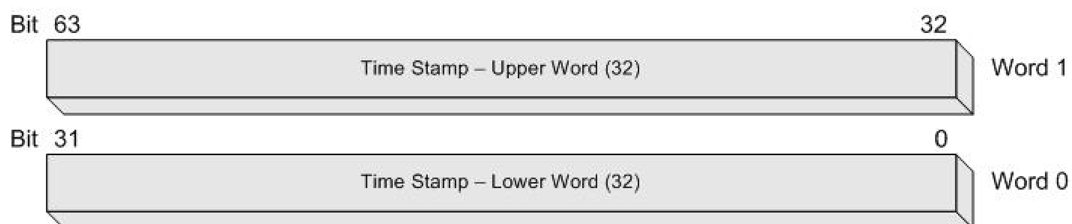

Figure 15: structure of a time stamp

As mention in section 2.2.3, the host has to set the bit field **TS = 1** in the Port Configuration Memory of the corresponding port to enable time stamping in any direction. Apart from that the host is not directly involved in generation of time stamps, because the TISS works autonomously for incoming messages and the TISS driver handles the generation of time stamps for outgoing messages (refer to section 3.3.5) so that this procedure is not visible in the host’s application code.

2.6 Timer Interrupt Service

The timer interrupt service (CORE_2_TIME) is an additional basic time service that provides immanent access to the common time service. In general, the common time service is incorporated in the autonomous dispatching of activities of the basic communication services and task trigger service. Both are not in the sphere of control of the host.

With the timer interrupt service the host obtains an additional degree of freedom and can harness the circuitry of the common time service for the implementation of application functions, e.g., the periodic dispatching of an operating system’s scheduler. Contrary to communication activities and task triggers, the timer interrupt service does not entail entries in the time-triggered schedule. Furthermore, the generated interrupts are only locally visible and can not be correlated at other components so that events can not be used for the coordination of distributed activities among several components.

2.6.1 Realization of the Timer Interrupt Service

Like the common time service the timer interrupt service involves a wrap-around overflow counter, which derives the associated interrupt on compare match with a top value. The registers to control this overflow counter are mapped into the register file of the timer interrupt service. A register TIMERTOP contains the top value, the register TIMERVAL presents the current value of the overflow counter. As the timer interrupt service is in the sphere of control of the host, the TIMERTOP register

is writeable so that the host is able to define the counting cycle until compare match. As expected, TIMERVAL is read-only.

Note that updates of the TIMERTOP value are only accepted, if and only if the timer interrupt service is disabled.

2.6.2 Running Modes

On compare match, the future behaviour of the timer interrupt service can be controlled by the running mode. Two options are supported:

- On compare match the timer interrupt service disables itself automatically – the so-called *single-shot mode*
- On compare match the timer interrupt service remains enabled, the overflow counter wraps around and continues counting so that a new interrupt is generated upon next compare match – the so-called *free-running mode*. In order to prevent further interrupt generations, the timer interrupt service must be explicitly deactivated by the host.

The running mode can be configured by means of the bit field **RM** in the register file of the timer interrupt service. Table 7 lists the mapping of running modes to the bit field **RM**.

Value of RM	Description
0	single-shot mode
1	free-running mode

Table 7: setting the running mode of the timer interrupt service

2.6.3 Activation

The host controls the activation via the register file of the timer interrupt service by means of the bit field **TE**. Table 8 shows the encoding of the bit field **TE**.

Value of TE	Description
0	timer interrupt service disabled
1	timer interrupt service enabled

Table 8: controlling activation of the timer interrupt service

On activation of the timer interrupt service by means of a write operation to **TE = 1**, the overflow counter starts from value 0 at the rising edge of the very next macro tick and increments on each macro tick. On compare match with the value of TIMERTOP the timer interrupt is generated. Depending on the running mode, the timer interrupt service disables automatically (i.e., single-shot mode) so that **TE** goes back to 0 on the very next clock cycle after the compare match. Otherwise (i.e. free-running mode) the timer interrupt service has to be deactivated by a dedicated write operation to **TE = 0**.

2.6.4 Register File of the Timer Interrupt Service

Figure 16 depicts the layout of the register file of the timer interrupt service.

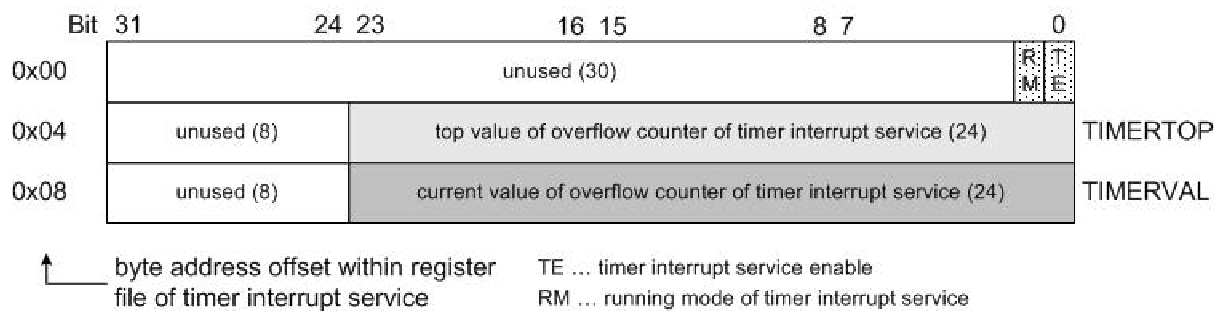

Figure 16: layout of the register file of the timer interrupt service

2.7 Error Detection Service

The error detection service (CORE_1_DIAG) is one of two basic diagnostic services. It records status information and error conditions that occur during operation of other core services (in particular the basic communication services). The purpose of this information is to reason about the health state of core services and the overall system to undertake countermeasures in case of deviation to specified behaviour. Additionally, it contributes to reveal software design failures during application software development such as timing violations or misconfigurations.

2.7.1 Error Sources and Error Flags

The error detection service defines 4 incidents – the so-called *error sources* – that are recorded in *error flags* and accessible by the host via its register file. We describe these error sources in the following.

- *watchdog miss* **(WM)**: If the host fails to update the watchdog life-sign register of the watchdog module of the error detection service (see section 2.7.3), the watchdog miss is indicated and recorded.
- *periodic message miss* **(PM)**: The TISS dispatches a burst that refers to an incoming periodic message, but the message does not arrive at the TISS' NoC interface within the valid receive window. Actually, the receiving component can not be blamed for the loss of that message, but recording of this incident allows to reason about a misconfiguration of the sending component or faulty routing in the TTNoC. The error detection service extracts the port identifier of the most recent port with a missed periodic message into the dedicated read-only field **LMP** of the register file of the error detection service, as depicted in Figure 17.
  Note that the omission of an incoming conditional messages is no error on this behalf.
- *sporadic queue overflow* **(QO)**: This error source indicates the overflow of the message queue of some sporadic port. Additionally, the error detection service also records the port identifier of the most recent port with an overflow. This information can be retrieved from the read-only bit field **LOS** in the register file of the error detection service, as shown in Figure 17.
  For incoming sporadic ports this incident means that the host in the component is too slow to consume the incoming sporadic message (for details refer to section 2.2.2.3). In this case, the TISS is forced to drop an incoming sporadic message, which is a severe error condition.
  For outgoing sporadic ports it pinpoints a too fast host that produces outgoing sporadic message in a higher frequency than the TISS sends them.
- *port lock notification* **(PL):** The TISS dispatches communication activities associated with a given port, but the host has locked or not yet unlocked that port in the Port Configuration Memory.

- *port configuration error* **(PC):** The TISS dispatches communication activities associated with a given port that is unlocked, though, however the port type **PT = 00** (“invalid”) in the memory word of the Port Configuration Memory.

These error flags are realized as single bit fields and mapped into the register file of the error detection service, as illustrated in Figure 17. A value of ‘1’ denotes a currently active error, and a value of ‘0’ indicates that a given error source has not been indicated since the last clearing of that error flag. A repeated occurrence of an error while the corresponding error flag has already been set since the last clearing of that flag can not be captured.

The error detection service incorporates an interrupt source, namely the *error interrupt*, which can be triggered if at least one error flag becomes active. For details refer to section 5.1.

2.7.2 Clearing Error Flags

From the host’s point of view, the error flags can either be read-only or read-writeable so that the host is able to clear the error flags itself. The bit field **CL** in the register file of the error detection service informs the host, which permission it possesses.

- If **CL = 1** the host is allowed to clear active error flags by writing a binary ‘1’ to the corresponding bit field. If that error flag is inactive at the time of writing a ‘1’, the state of the bit field is unaffected. Moreover, a write operation with a binary ‘0’ is ignored.
  Note that this setting excludes the usage of the health reporting service.
- If **CL = 0** each write operation to the bit fields of error flags is ignored so that the host is not able to clear error flags. Instead of this, error flags are automatically cleared (by the TISS) on dissemination by the health reporting service (see section 2.8). Logically, this setting implies that the health reporting service is present.

The bit field **CL** is read-only for the host. Actually, it is in the sphere of control of the TRM. Consequently, the TRM decides whether the host is allowed to maintain error flags on its own, e.g., if the application software features some software module that processes error flags and handles its dissemination in the application-level context such as a Local Diagnostic Unit. For this purpose, the TRM uses the inter-component channel configuration to access the bit field **CL**.

Note that the read-only bit fields **LOS** and **LMP** are not meant to be cleared. Anyway, their values are only reasonable if and only if the corresponding error flag indicates an active error condition.

2.7.3 Watchdog

The error detection service entails a watchdog feature to assure the liveliness of the host. The host is forced to update the watchdog life sign register of the register file of the error detection service within a given period – the *watchdog period* - to avoid a *watchdog miss* error.

The watchdog is active, if the bit field **WA = 1** of the register file of the error detection service. Similar to the bit field **CL**, **WA** is in the sphere of control of the TRM so that the host is not able to activate or deactivate the watchdog on its own. Additionally, the bit field **WB** that is also configured by the TRM via the inter-component channel configuration informs about the behaviour of the TISS in case of a watchdog miss. If **WB = 1**, on a watchdog miss the TISS resets the host by driving the “reset wire” like the *reset* command of the component control service and records the incident in the corresponding error flag. Otherwise **(WB = 0)** the watchdog miss is just recorded, but the TISS does not trigger a reset of the host. This feature might be useful for debugging of application software to inform the application software of failed timings by means of an error interrupt and not get confused by an unexpected reset.

The update of the 32-bit wide watchdog life sign register WDLIFE involves a write operation by the host with a pre-defined bit pattern. A read operation from WDLIFE reveals that bit pattern, e.g.,

0x55555555. We recommend retrieving the watchdog life sign by the host at start-up of the application software.

The host has to perform such a write operation at least once per *watchdog period*. The host learns the watchdog period from the read-only bit field **WDP** of the register file of the error detection service, which holds a numeric index of some period of the common time service used to realize the watchdog period. Same as **WA** and **WB**, the watchdog period **WDP** is set by the TRM. Consequently, the host can be certain that the settings of **WA**, **WB**, and **WDP** and therefore the configuration of the watchdog are consistent and trusted.

2.7.4 Register File of Error Detection Service

Figure 17 depicts the layout of the register file of the error detection service.

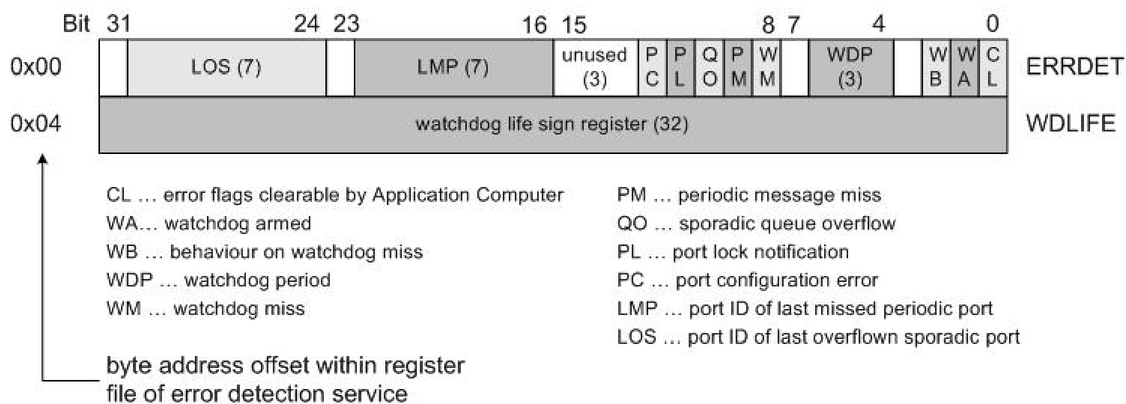

Figure 17: layout of the register file of the error detection service

2.8 Health Reporting Service

The health reporting service (CORE_2_DIAG) is the second basic diagnostic service. Its purpose is to disseminate the status information and error conditions that have been recorded by the error detection service.

As mentioned in section 2.7.2, the health reporting service is mutually exclusive with write access to the error flags in the register file of the error detection service. Thus, if a host wants to check, whether the health reporting service is currently present or not, it simply checks the bit field **CL = 1** in the register file of the error detection service.

2.8.1 Health Reporting Channels

Similar to the component control service, the health reporting service uses dedicated communication channels – the *health reporting channels* - to convey the status information and error flags collected by the error detection service. Certainly, the communication activities associated with health reporting channels manifest in the time-triggered schedule. The messages transported in the health reporting channels are called *health state messages*.

In general the TISS compiles the information of the error detection service and transmits them autonomously. However, the host is allowed to insert application-specific data into health state messages in order to use the set of available ports at a component more efficiently.

The route determines the source and sinks of the health reporting channel. The sink is some component, i.e., the *Diagnostic Unit*, within the MPSoC (or at another MPSoC reachable via a gateway) that receives health state messages of all other components in order to reason about the

overall health state of that MPSoC. The source of each health reporting channel is a distinct component respectively its TISS. In general, all TISSs disseminate health state messages via distinct health reporting channels, which have a common sink.

As its name suggests, health state messages are realized as **periodic messages**. From the point of view of the sink, i.e., the Diagnostic Unit, the health reporting channel is accessible via an “ordinary” incoming periodic port that is subscribed by the basic communication service. On the other side, the health reporting channel at the source, i.e., at each component, is modelled as “ordinary” outgoing periodic port that is subscribed by the health reporting service. For details about subscription of ports refer to section 4.1.1.

Nevertheless, to include the information of the error detection service a special treatment, which actually is transparent to the source-side host, must be executed for the outgoing periodic messages implementing health state messages.

2.8.2 Structure of Health State Messages

First of all, the purpose of a health state message is to distribute the findings of the error detection service. So, a copy of the register ERRDET of the register file of the error detection service is mandatory.

Secondly, a host might contain a software module that executes some local analysis concerning the health state of the component, i.e., the Local Diagnostic Unit mentioned earlier. This software module produces information that is relevant for the (global) Diagnostic Unit so that it has to disseminate this information, too. On the one hand, it could occupy a dedicated communication channel for this diagnostic information. On the other hand, the health reporting service offers the possibility to incorporate this diagnostic information into health state messages.

Figure 18 shows how a health state message is compiled at the source of the health reporting channel and stored in the Port Memory of the sink. Note that the copy of the ERRDET register, which succeeds the application-specific data, is present in the Port Memory of the sink, but not in the source. From the source’s point of view, the application-specific data is generated by the host like an “ordinary” outgoing periodic message that supports time stamping, message complete interrupts, explicit and implicit synchronization etc. However, the ERRDET register resides within the TISS and is not explicitly copied into Port Memory before transmission of the overall health state message. Instead of this, the TISS applies a special treatment by inserting the copy of the ERRDET register into the data stream after the raw application-specific part. This process is transparent to the host at the source. If the port associated with the health reporting channel has the message complete interrupt enabled in its entry of the Port Configuration Memory, the TISS triggers that interrupt after the transmission of the copy of the ERRDAT register.

The sink-side obtains a consistent message, whereas the very last data word embodies the copy of the ERRDAT register of the source. Note that the size of the message in the Port Memory of the sink is **greater by one data word** than in the Port Memory of the source, because the copy of the ERRDAT register consumes one additional data word at the sink-side. Consequently, the extension of the health state message at the side has to manifest in the **Size** field of the Port Parameter Memory for each incoming periodic port of health reporting channels at the sink.

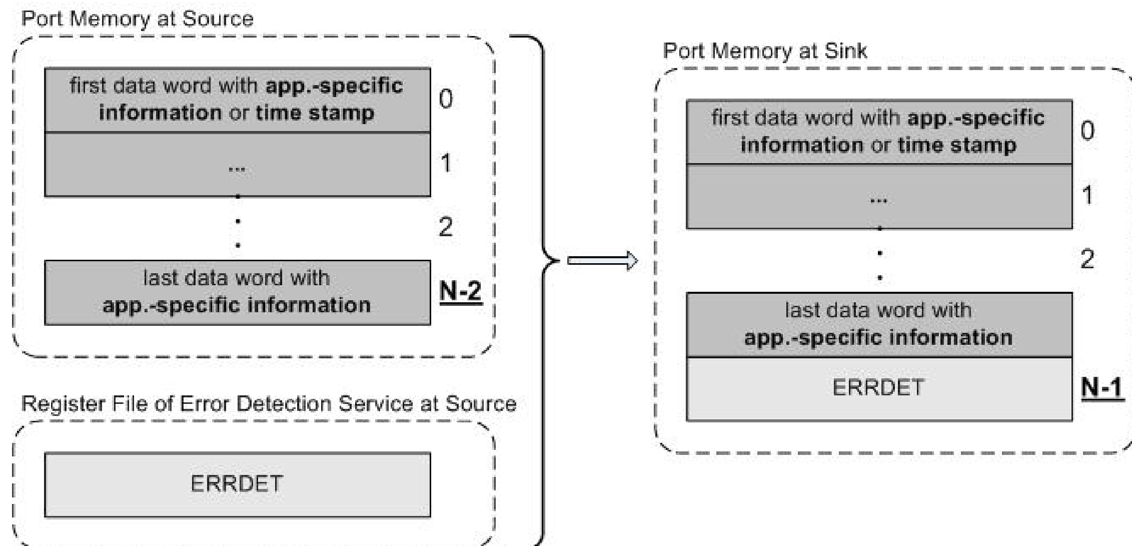

Figure 18: structure of a health state message

2.8.3 Omitting Application-Specific Data

In some application a Local Diagnostic Unit might not be present or is deactivated at run-time so that there is no need to include application-specific data. There are two possibilities how to omit application-specific data in health state messages.

1. At design time the need for application-specific data in health state message is precluded a priori. Then, the size of health state message is fixed at one single data word that just contains the copy of the ERRDAT register at the source as well as the sink of the corresponding health reporting channel. Even though the host at a source of some health reporting channel configures the port subscribed by the health reporting service and submits messages into its Port Memory, the TISS does not pay attention to this data.
2. During run-time the host at the source-side of a health reporting channel locks the port subscribed by the health reporting service by setting the field **PU = 0** in the Port Configuration Memory. As a consequence, the TISS at the source-side only uses the copy of the ERRDAT register to compile the health state message and fills up the missing application-specific data with dummy data words consisting of all 0 to achieve a complete health state message as given by the **Size** field in the Port Parameter Memory.
   In such a case the host at the sink-side finds the copy of the ERRDAT register at the very last data word of the health state message as usual. Unfortunately, it has to discard the dummy application-specific part of the health state message, which can easily be recognized by a value of all 0.

The latter case is necessary to continue the transmission of health state messages in order to prevent a faulty or malicious host from suppressing the dissemination of status information and error conditions that have been recorded by the error detection service.

2.8.4 Clearing Error Flags of the Error Detection Service

A host has no write access to the error flags of the register file of the error detection service, if the health reporting service is present. Consequently, it has no means to clear the error flags itself.

The TISS clears these error flags autonomously on transmission of the health state message. To be more precise, the error flags are set to inactive at the very next clock cycle of the system operation frequency after the copy of the ERRDAT register has been sent.

2.8.5 Register File of the Health Reporting Service

The health reporting service has no explicit register file.

2.9 Identification Service

The identification service is solely realized as the registers *interface revision number* and *component ID* as introduced in Figure 2 in section 2.1.2.1.

2.10 Inter-Component Channel Configuration

The purpose of the inter-component channel configuration core service is to grant the TRM access to register file and protected memories, which are not writable from the host's point of view. Consequently, the inter-component channel configuration is the one and only core service, which operation can not be observed from the host's UNI. In other words, the process of on-the-fly reconfiguration during run-time is transparent to the host. The host is only able to get informed about completion of reconfiguration by means of the reconfiguration interrupt mentioned in section 5.1. Moreover, the host might notice a port subscribed by this core services, so that that port is not available for its application software.

While the functional behaviour of inter-component channel configuration has first been specified in D1.2, we do not reveal the in-depth design in this document D1.3, because this core service does not posses an interface to the host at all. The details of implementation of the inter-component channel configuration can be found in D1.6.

3 Application Programming Interface

This chapter describes the application programming interface (API) how to use the ACROSS core services in the application software executed on a host. Generally, the core services are accessible via a driver software module, the so-called *TISS driver*.

3.1 The TISS Driver

The TISS driver handles the Uniform Network Interface (UNI) of the TISS. It provides a set of functions which abstract from implementation details of core services, e.g., memory consistent access to the Port Memory. These functions can be organized as follows:

- *register access functions*: These hide the address translations when accessing registers of register files of the core services.
- *send & receive functions*: As the basic communication services are the most frequently used, they exhibit dedicated functions that encapsulate the memory consistent access to the Port Memory ("port synchronization") and transfer messages between Port Memory and the host's working memory.
- *interrupt callback mechanism*: As the core services entail a set of interrupt sources, the TISS driver establishes a software mechanism of callback routines that realize the interrupt service routines.

The code of the TISS driver can be classified in a platform-independent and platform-dependent part. The platform-independent part is written in such a coding style that this source code can trivially be ported to other hardware architectures simply by recompilation with a cross-compilation tool chain. Actually, the send & receive functions, high-level parts of the interrupt callback mechanism as well as the address translation of register access functions are platform-independent.

The platform-dependent part is bound to a given hardware architecture, for example Altera NiosII CPUs. For example, this part contains the low-level I/O primitives in order to access memory-mapped registers and platform-specific data types. Moreover, the platform-dependent part incorporates the glue code of the target architecture's software framework. In case of Altera NiosII CPUs this is the code to register interrupt service routines in the Altera HAL and to conduct platform-specific initializations.

The following sections elaborate on each category of driver functions. For details about the type definitions used by the TISS driver refer to chapter 6.

3.2 Register Access Functions

We have learned in section 2.1 that a host can retrieve status and configuration information, synchronization flags, and control registers of core services, protected memories, and interrupt control. This information is available in registers of a set of register files that is accessible via the memory-mapped Control Interface. Then, Figure 2 has introduced a look-up table of pointers which refer to start offsets of these register files within the address range of the Control Interface.

Whenever a host intends to access a given entity of a register file, it has to determine the absolute address from the point of view of its memory-mapped (AVALON) master, e.g., the data master of a NiosII CPU that is attached to the Control Interface. The process of address translation requires a calculation in four steps.

1. read the pointer to the register file of the given core service
2. add that pointer to the static base address of the Control Interface in the memory-mapped (AVALON) master's address range
3. add the relative offset of the indented entity within the register file to the sum of the above
4. convert the result into a byte address and return it to the application software's context

These four steps of address translation are implemented in the driver function `across_getRegPtr()`.

Note that it is possible to use hard-coded addresses of registers, of course. However, this is not recommended, as the layout of the Control Interface can be changed due to extension or redesign of core services. Furthermore, the usage of address translation facilitates the change of the Control Interface's layout without recompilation of the application software.

3.2.1 Function: across_getRegPtr()

This function determines the byte address of a data word in the address range of the Control Interface. The parameters of the function specify the register file and distinct register within that register file, which is supposed to be read or written in the application software by means of a pointer dereference.

3.2.1.1 Signature

```
tiss_register *across_getRegPtr(int regfile, unsigned int offset);
```

3.2.1.2 Parameters

The parameter `regfile` is an integer identifier of the core service, which the intended register belongs to. This parameter accepts all constants of the enumeration type `registerfiles_t`.

The parameter `offset` embodies the relative offset of the intended entity within the register file of the core service given by `regfile`. Note that the TISS driver does not define constants for `offset` of frequently used registers.

3.2.1.3 Return Value

The return value of `across_getRegPtr()` is a pointer with the absolute byte address of the 32-bit data word within the Control Interface that contains the register specified by the two parameters. Plain 32-bit data words of the Control Interface are modeled by the data type `tiss_register`.

3.2.1.4 Behaviour in case of error

In case of an error the function returns a null pointer. An error occurs in the following situations:

- The enumeration type `registerfiles_t` does not include the constant value given by `regfile`.
- The `offset` exceeds the maximum number of data words in the register file of the core service identified by `regfile`.
- The register file given by `regfile` does not contain registers at all.

3.2.1.5 Example Usage

```
tiss_register *regptr;

/* get the pointer to the single data word of the register file of the
execution control service */
regptr = across_getRegPtr(regfile_execctrl, 0);
if (!regptr)
{ /* error */ }
*regptr = 1 << 15;      /* clears the field RC */
```

3.2.2 Function: across_getIRV()

The purpose of the function `across_getIRV()` is to retrieve the interface revision number, as defined in section 2.1.2.2. This function could be used at start-up of the application, i.e., at the very beginning of `main()`, to check whether the application software has been designed for the proper revision[8] of the core services, their register files, and the features of the TISS driver.

3.2.2.1 Signature

```
tiss_register across_getIRV(void);
```

3.2.2.2 Return Value

The function provides a full 32-bit wide data word (copy-by-value) of the Control Interface. Its value is a plain hexadecimal string that contains the interface revision number interpreted as a date in the format YYYY/MM/DD, which is 0xYYYYMMDD in a hexadecimal notation.

3.2.2.3 Example Usage

```
tiss_register irv;

irv = across_getIRV();
if (irv != 0x20110105) /* interface revision of January 5th, 2011 */
{ printf("Dazed and confused!"); /* handle incompatible IRV */ }
```

3.2.3 Function: across_getComponentID()

Similar to `across_getIRV()`, the function `across_getComponentID()` retrieves component ID and serial number, as defined in section 2.1.2.2.

3.2.3.1 Signature

```
tiss_register across_getComponentID(void);
```

3.2.3.2 Return Value

The function provides a full 32-bit wide data word (copy-by-value) of the Control Interface. Its value contains a plain hexadecimal string that contains the component ID and serial number.

[8] The interface revision number will be defined when this design document has reached a “stable” maturity stage.

3.3 Send & Receive Functions

The send & receive functions are the most frequently used functions of the TISS driver, because they cover the fundamental procedures of communication in the ACROSS architecture. They consist of two pairs of functions:

1. *back-end functions*: These functions encapsulate the memory consistent access to the Port Memory and follow the port synchronization protocols defined throughout section 2.2.2. Additionally, they facilitate direct operation on the Port Memory, i.e., the application software is able to generate or retrieve messages word-by-word without memory copy operations between Port Memory and working memory like the front-end functions.
2. *front-end functions*: These functions are usually invoked by application software to submit whole messages to the Port Memory (send) or fetch whole messages from the Port Memory (receive). They invoke the back-end functions in order to have the port synchronization handled. Unlike the back-end functions they are “copying” functions, i.e., they copy whole messages between Port Memory and working memory of the host. Finally, the front-end send function is in charge of producing the time stamp for outgoing messages if time stamping is activated for the corresponding port.

Usually, the application software uses the front-end functions in its source code, as they seem to be sufficient in most situations and are more comfortable. The direct invocation of the back-end functions in the source code of the application software is for advanced usage only.

3.3.1 Blocking Mode of Send & Receive Functions

In general communication in ACROSS is deterministic and synchronous among ACROSS components. However, this property only holds below the temporal firewall which conceptually can be identified as the Uniform Network Interface (UNI) of the TISS. A component’s behaviour beyond the UNI can not be assumed as that deterministic. Consequently, it might be desirable for the application software executing in hosts to explicitly synchronize on events generated by the TISS that reflect the deterministic character of ACROSS’ communication.

In the context of the communication service such “synchronization points” are the completed transmission of messages of ports. For outgoing ports this *message complete* event correlates to the sending of the very last data word of an outgoing message. For incoming ports it is the reception of the very last data word of an incoming message, probably prolonged by the time it takes the TISS to produce the time stamp if activated for that port.

These “synchronization points” can be identified by the host by two means:

- notification of a completed message by an interrupt, the so-called *message complete interrupt*
- polling of port synchronization flags until they indicate the arrival of an incoming or the complete transmission of an outgoing message

The first incorporates the interrupt callback mechanism presented in later section 3.4. The second way leads to the *blocking mode* mentioned throughout section 2.2.2 for outgoing periodic as well as sporadic ports.

As the send & receive functions of the TISS driver process the port synchronization flags and realize the memory consistent access to ports, they are the point where the blocking mode has to be considered.

Usually, blocking send & receive functions exclude the usage of the interrupt callback mechanism for message complete interrupts. On the contrary, if that interrupt callback mechanism is integrated in application software, send & receive functions are reasonably used non-blocking only. The first might be the case for simple, single-threaded application software, where the installation of the interrupt callback mechanism for message complete interrupts is too expensive. The second is the

method of choice for concurrent, multi-threaded application software that demands for efficient CPU usage.

3.3.2 Back-end function: across_getMsgPtr()

The purpose of `across_getMsgPtr()` is to calculate the start address of a message with respect to the memory layout of ports (see section 2.2.1). That start address embodies an absolute byte address in the Port Memory that is mapped in the memory-mapped master's address space, e.g., the data master of a NiosII CPU.

For outgoing ports the function determines the buffer, where the application software can place the message it wants to send. For incoming ports the function provides the buffer where the TISS has stored the prior received message. For this purpose the function distinguishes between direction, port type, and synchronization mode of a given port, which have impact on the memory layout of the port. Furthermore, it applies the according port synchronization protocol.

3.3.2.1 Signature

```
across_errcode across_getMsgPtr(across_portid portid, void **msgbuffer,
                                char blocking);
```

3.3.2.2 Parameters

The first parameter `portid` is the numeric identifier of a port, which the operation is applied to. Moreover, it is the offset to refer to the corresponding memory entry in Port Configuration, Port Synchronization and Port Parameter Memory, which is necessary to get all information required for the operation of that function.

The second parameter `msgbuffer` is a pointer of a pointer, whereas the first tier pointer refers to the absolute start address of a buffer in the Port Memory. The parameter `msgbuffer` contains the address of that first tier pointer, so that the function `across_getMsgPtr()` is able to pass through a new value.

The third parameter `blocking` is supposed to be a Boolean value. A value unequal 0 (preferably 1) activates the blocking mode. However, `across_getMsgPtr()` does not realize blocking of incoming periodic messages. In this special case, the blocking mode has to be directly considered by the front-end receive function `across_recvMsg()` or the by application software if it applies back-end functions for advanced uses.

3.3.2.3 Return Values

The function returns error codes that inform about the success of the operation. On success `across_getMsgPtr()` delivers the constant `errcode_success`, which is a constant in the enumeration type `errcodes_t`.

3.3.2.4 Behaviour in case of errors

There are four situations when the function aborts with an error code.

- The given port identifier in `portid` does not exist (`errcode_invalid_portid`).
- The message queue of an outgoing sporadic port is full and blocking mode is deactivated (`errcode_queue_full`).
- The message queue of an incoming sporadic port is empty and blocking mode is deactivated (`errcode_queue_empty`).

- There is a message transmission ongoing for outgoing periodic ports (with explicit synchronization), and blocking mode is deactivated (`errcode_busy`).

In addition to the indication of an error by the return value, for all situations except `errcode_invalid_portid` the function also sets `msgbuffer` to 0 so that the first tier pointer becomes a null pointer. For `errcode_invalid_portid` itself `msgbuffer` is not invalidated, yet not even modified.

3.3.3 Back-end function: across_commitMsg()

As it name suggests, this function declares a message of a port as completely processed. For this purpose, it iterates the port synchronization flags according to port type, synchronization mode and direction, as specified throughout section 2.2.2.

Logically, `across_commitMsg()` must be invoked after `across_getMsgPtr()`, which determines or “opens” the current buffer, while `across_commitMsg()` commits or “closes” the buffer.

After the execution of `across_commitMsg()`, an outgoing message is ready to be transmitted at the corresponding send instant in the time-triggered schedule of the given port. Incoming periodic messages are considered as processed, while incoming sporadic message are regarded as consumed.

3.3.3.1 Signature

```
across_errcode across_commitMsg(across_portid portid);
```

3.3.3.2 Parameters

The single parameter `portid` is the numeric identifier of a port, which the operation is applied to. Moreover, it is the offset to refer to the corresponding memory entry in the Port Synchronization Memory in order to access the port synchronization flags.

3.3.3.3 Return Values

The function returns error codes that inform about the success of the operation. On success `across_commitMsg()` delivers the constant `errcode_success`, which is a constant in the enumeration type `errcodes_t`.

3.3.3.4 Behaviour in case of errors

There is only one situation when an error is raised, namely if the given port identifier `portid` does not exist. In this case the return value is `errcode_invalid_portid`.

3.3.4 Front-end function: across_recvMsg()

This function is supposed to be used by the application software. Literally, it “receives” a message. That is, it copies a whole message word-by-word (including time stamp if enabled for that port) from the Port Memory into some local buffer of working memory in the scope of the application software.

The function `across_recvMsg()` leverages the back-end functions `across_getMsgPtr()` and `across_commitMsg()` in order to consider memory layout and memory consistent access of ports. So, this function wraps the back-end functions, adds a word-wise copy, and considers blocking of incoming periodic message.

3.3.4.1 Signature

```
across_errcode across_recvMsg(across_portid portid, void *appbuffer,
                              char blocking);
```

3.3.4.2 Parameters

The first parameter `portid` is the numeric identifier of a port to retrieve the message from. Additionally, the `portid` is a required parameter by the back-end functions invoked by `across_recvMsg()`, so that it forwards this value to these functions.

The second parameter `appbuffer` is a pointer that refers to the start address of a message buffer in some working memory of the host, where the complete message in the Port Memory is copied to. Logically, that message buffer has to be maintained by the application software.

The third parameter `blocking` is supposed to be a Boolean value. A value unequal 0 (preferably 1) activates the blocking mode. The function `across_recvMsg()` only realizes blocking for incoming periodic messages. As the parameter `blocking` is forwarded to the back-end function `across_getMsgPtr()`, blocking for all other port types, synchronization modes, and directions is realized in there.

3.3.4.3 Return Values

The function returns error codes that inform about the success of the operation. On success `across_recvMsg()` delivers the constant `errcode_success`, which is a constant in the enumeration type `errcodes_t`.

3.3.4.4 Behaviour in case of errors

As `across_recvMsg()` calls the back-end functions, it passes through their error codes via its return value. Additionally, it raises an error in one situation, namely if blocking mode is deactivated and the TISS updates the incoming periodic message during the copy operation according to the port synchronization described in section 2.2.2.2.1. In this case the return value is `errcode_busy`.

3.3.5 Front-end function: across_sendMsg()

This function is supposed to be used by the application software. Literally, it "sends" a message. That is, it copies a whole message word-by-word from some local buffer of working memory into the Port Memory.

The function `across_sendMsg()` leverages the back-end functions `across_getMsgPtr()` and `across_commitMsg()` in order to consider memory layout and memory consistent access of ports. So, this function wraps the back-end functions and adds a word-wise copy. If time stamping is activated for the given outgoing port, the function also generates the time stamp after the copy of the very last data word of the message and before iteration of port synchronization via `across_commitMsg()`.

3.3.5.1 Signature

```
across_errcode across_sendMsg(across_portid portid, void *appbuffer,
                              char blocking);
```

3.3.5.2 Parameters

The first parameter `portid` is the numeric identifier of a port via the message is to be sent. Additionally, the `portid` is a required parameter by the back-end functions invoked by `across_recvMsg()`, so that it forwards this value to these functions.

The second parameter `appbuffer` is a pointer that refers to the start address of the buffer in some working memory of the host, where the complete message to be sent has been prepared by the application software. Logically, that buffer has to be maintained by the application software.

The third parameter `blocking` is supposed to be a Boolean value. A value unequal 0 (preferably 1) activates the blocking mode, which has been mentioned throughout section 2.2.2. Note that the function `across_sendMsg()` does not use this parameter itself. However, the parameter `blocking` is required by the back-end function `across_getMsgPtr()` and therefore forwarded to its function call.

3.3.5.3 Return Values

The function returns error codes that inform about the success of the operation. On success `across_sendMsg()` delivers the constant `errcode_success`, which is a constant in the enumeration type `errcodes_t`.

3.3.5.4 Behaviour in case of errors

As `across_sendMsg()` calls the back-end functions, it passes through their error codes via its return value. It does not raise any errors itself.

3.4 Interrupt Callback Mechanism

The core services produce events during their operation, which are reported by means of interrupts to the host, as it is described in chapter 5. The host receives these interrupts through the ACROSS component's system interconnect and might intend to react to the interrupts in its application software. On the one hand, it has to handle the interrupt status flags in order to acknowledge the interrupts. This handling is completely encapsulated by the TISS driver functions that establish the interrupt callback mechanism. Thus, the application software is not concerned with the semantics of interrupt handling. On the other hand, the application software has to provide application-specific source code that is to be executed on each interrupt; so to say the *interrupt service routines of callback*.

This section describes the functions of the TISS driver that build this interrupt callback mechanism and how the application software integrates its interrupt service routines in source code.

3.4.1 Dispatching of interrupts

The interrupt callback mechanism starts at the interrupt vector table, which contains a function pointer for each interrupt line that terminates at the host. According to chapter 5, the TISS entails three interrupt lines. Consequently, the TISS driver has to attach to those entries in the interrupt vector table.

We see in Figure 19 that this first tier of the interrupt callback mechanism is occupied by so called *dispatcher functions*. Upon initialization of the TISS driver (at start-up), it registers these dispatcher functions in the interrupt vector table. For example, in the Altera HAL this is done by invoking the function `alt_irq_register()`.

The dispatcher functions are the second tier of the interrupt callback mechanism. They consult the *hooks* in order to dereference the actual user-specified interrupt service routines. The TISS driver provides the following dispatcher functions.

- The *message complete dispatcher* is associated with the communication interrupt and serves to further dispatch the user-specified interrupt service routines on *message complete*. The *message complete hooks* embody a directory of interrupt service routines, whereas each entry

corresponds to a port identifier. The message complete hooks are implemented as an array of function pointers.

- The *task trigger dispatcher* is processed upon the task trigger interrupt. Similar to the message complete dispatcher, it processes the single task activations. For each task there is one interrupt service routine implemented in application software. The task trigger dispatcher uses the *task trigger hooks* to enumerate these interrupt service routines, which is also implemented as array of function pointers.
- The remaining interrupt sources are collected by one interrupt line. The interrupt service routines associated with these interrupt sources are specified in the *generic hooks*, which is used by the *generic dispatcher* to call the proper interrupt service routine.

Dispatcher functions handle the interrupt control registers.

The generic dispatcher evaluates the interrupt status flag register (INTSTAT) to determine the interrupt sources that caused the interrupt to rise. Afterwards it calls the according interrupt service routine that is registered in the generic hooks.

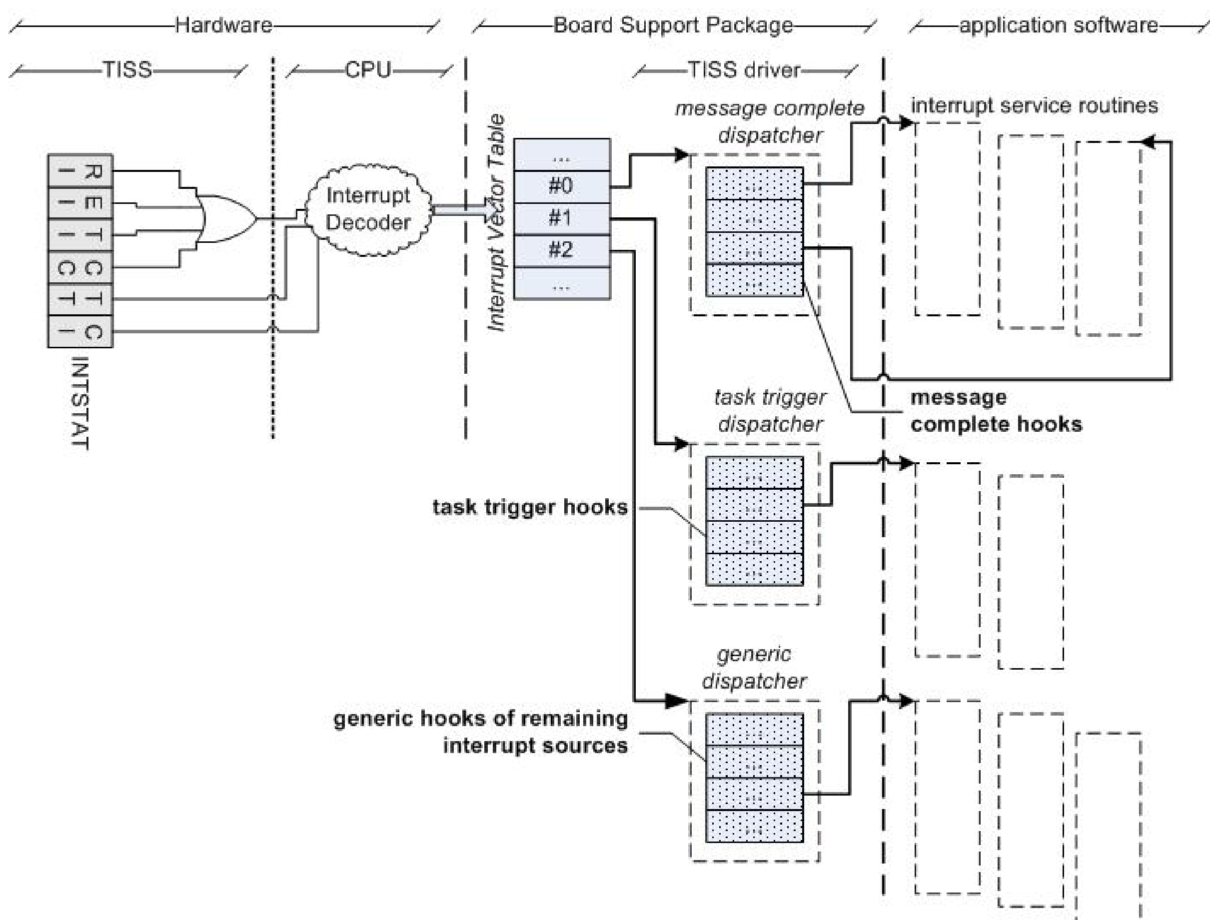

Figure 19: the interrupt callback mechanism

The message complete dispatcher looks through the interrupt status flag registers of message complete interrupts (MSGSTATn) to identify the ports with completed message and to invoke the proper interrupt service routines from the message complete dispatcher.

Similarly, the task trigger dispatcher consults the register file of the execution control register to learn the currently activated task and to resolve the proper entry in the task trigger hooks. Finally, all dispatcher functions clear the interrupt status flags after termination of the invoked interrupt

service routine. For this purpose, they apply write operations to the corresponding interrupt acknowledge registers (INTACK respectively MSGACKn).

If a hook is unset, i.e., the entry in the array of function pointers contains the null pointer, but an interrupt service routine should be dispatched, no callback is executed. However, the dispatching function clears the corresponding interrupt status flag. In other words, interrupts are acknowledged if there exists no interrupt service routine for them anyway. The reason for this is to make the application software more robust. If the application software omits interrupt service routines (although it has enabled the corresponding interrupt source by interrupt masking), an orphaned interrupt status flag will cause permanent interrupts and eventually live-lock the host.

Note that application software need not care about handling of the interrupt control registers, because this is already done by the dispatcher functions. Also note that dispatcher functions are not visible to the application software. It must only provide the user-specified interrupt service routines and "hook them in", i.e., configure the function pointers within hooks to point to these interrupt service routines.

3.4.2 Set-up of hooks: across_setHook()

As hooks relate to user-specified interrupt service routines, their array of function pointers must explicitly be set-up in the application software's source code. For this purpose the TISS driver offers the function `across_setHook()`. This is the only TISS driver function that is visible to the application software when dealing with the interrupt service routines.

3.4.2.1 Signature

```
across_errcode across_setHook(int hooks, int item, (void)(*fptr)(void));
```

3.4.2.2 Parameters

The first parameter `hooks` identifies the array of function pointers, i.e., "hooks", which the operation is applied to. It accepts the constants of the enumeration type `hooks_t`.

The second parameter `item` specifies the item within the given hooks, i.e., it's the index for the array of function pointers that is used to deference the according function pointer.
For the generic hooks (parameter `hooks` is `hooks_generic`) this parameter is a constant of the enumeration type `interrupt_sources_t`. Note that `interrupt_comm` and `interrupt_tasktrigger` do not belong to `hooks_generic` as specified in section 3.4.1. For message complete hooks and task trigger hooks (parameter `hooks` is `hooks_comm` respectively `hook_tasktrigger`) the parameter `item` holds the numeric identifier of the port respectively task for which the interrupt service routine is assigned.

The third parameter is a function pointer that contains the address of the interrupt service routine that is assigned to the item in some hook given be the other parameters.

3.4.2.3 Return Values

The function returns error codes that inform about the success of the operation. On success `across_setHook()` delivers the constant `errcode_success`, which is a constant in the enumeration type `errcodes_t`.

3.4.2.4 Behaviour in case of errors

As `across_setHook()` performs range checks of the first two supplied parameters, it at least one parameter exceeds its allowed range this is expressed by the following return values.

- If the parameter `hooks` contains a value not covered by the enumeration type `hooks_t`, the return value is `errcode_unknown_hooks`.
- If the parameter `item` exceeds it valid range, the error code is `errcode_invalid_item`. For the generic hooks this is the case when the value of item is not in the enumeration type `interrupt_sources_t`. For message complete hooks the parameter is not allowed to exceed the number of supported ports, i.e., range 0 - 127. The same applies to the task trigger hooks, whereas the parameter must be lower than the number of supported tasks, i.e., range 0 – 255.

3.4.3 Set-up of interrupt masks

The TISS driver does not provide an explicit function to deal with interrupt masks. Instead of this, register access functions must be used in order to determine the absolute address of interrupt mask and unmask registers in the Control Interface's address range. Then, the application software uses pointer dereferences to realize read or write operations. Finally, the application software has to follow the semantics of interrupt masking as defined in section 5.3.

3.4.4 User-specified dispatcher functions

In some situations the message complete dispatcher or the task trigger dispatcher might not satisfy the needs of the application software. For example an operating system entails its own task dispatcher and therefore has no utility for the task trigger dispatcher of the TISS driver.

For such advanced requirement it is possible for the application software to hook an own interrupt service routine directly into the interrupt vector table so that the dispatcher functions of the TISS driver are bypassed. However, this procedure is platform-specific and involves the registration functions for interrupt service routines of the underlying framework, e.g., `alt_irq_register()` of the Altera HAL. Then, the interrupt vector table for a given interrupt is overwritten by a user-specified interrupt service routine for instance at start-up of the application software and after the initialization of the TISS driver.

Also consider that the user-specified interrupt service routines replacing the dispatcher functions must take over the handling of the interrupt control registers.

3.4.5 Limiting size of hooks

The array of function pointers used to realize hooks in the TISS driver are dimensioned to span all possible interrupt sources. While this is a comparably low number of the generic hooks (just 7 elements of the function pointer array at the moment), that array is quite large for the message complete hooks and task trigger hooks, because the array have the size of the number of supported ports respectively supported task triggers. However, a given application might not use all of these function pointers leaving a considerable amount of precious embedded memory occupied without any usefulness.

In order to circumvent such an unpleasant situation the TISS driver can be configured by means of constant definitions that denote the number of actually used ports and task triggers by the application software. As a result, the function pointer arrays can be tailored to the needs of the application software.

4 Protected Memories

This chapter deals with all memories that physically reside in the TISS and are accessible via the Control Interface, but the host only has read-access. The reason for this restriction is that these memories contain critical information concerning the timer-triggered schedule, dispatching according to the time-triggered schedule, and routing in the TTNoC. In general, a host is not allowed to modify this information in its TISS' memories. However, it is necessary for some operations of the host, e.g., port synchronization, to obtain this information. Consequently, the TISS allows read access to these protected memories.

We define four memories within the TISS to be protected:

1. the Port Parameter Memory
2. the memory of the time-triggered schedule
3. the Burst Configuration Memory
4. the Routing Information Memory

In the following section we describe each memory in detail.

4.1 Port Parameter Memory

As mentioned in section 2.2.1, the port and its memory layout is characterized by four parameters. Two of them can be configured by the host, whereas the remaining two have direct impact on bandwidth and routing in the TTNoC and are therefore in the sphere of control of the TRM. While the first are physically located in the Port Configuration Memory, the Port Parameter Memory is the entity where the latter protected parameters reside.

The information contained in the Port Parameter Memory is required during normal operation of the host, e.g., to generate time stamps for outgoing messages or to organize ports in the Port Memory. In addition to this, the TISS also relies on this information for port synchronization and to determine the direction of a communication channel.

Figure 20 shows the layout of a memory entry of the Port Parameter Memory.

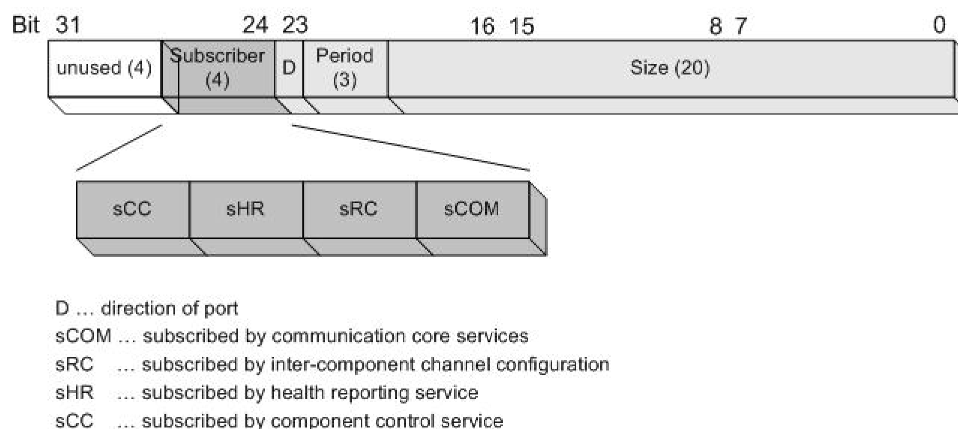

Figure 20: layout of a memory entry of the Port Parameter Memory

The first field is named **Size** which holds the size of a complete message associated with a given port. The value of **Size** denotes the number of data words of the Port Memory. Its 20-bit width matches

the port base address (field **PBA** in the Port Configuration Memory) so that the maximum size of a single message can span the whole addressable range of the Port Interface. This information is a required input when the host has to organize the memory allocation for ports in the Port Memory.

The field **Period** is the numeric index of a period (i.e., overflow counter) of the common time service, in which the communication activities associated with a given port are dispatched. In other words, this field informs the host with which period the message of a given port are transmitted. Note that application software in the host should not be dependent on this information. It just serves informational purposes.

The bit field **D** is the direction of the corresponding port. A value of **D = 0** identifies an incoming port. Contrary, **D = 1** refers to an outgoing port.

4.1.1 Subscription of Ports

The last entity in an entry of the Port Parameter Memory is the **Subscriber** field. It is a 4-bit wide field, whereas each bit refers to a distinct core service.

Each port can be used by **at most one** core service for a given application scenario. We call such assignment a *subscription* and the corresponding core service it *subscriber*. A bit value of '1' denotes a subscription by the corresponding core service, whereas each port can have at most one bit set to '1'. If a port has no bit in the **Subscriber** field set to '1', it is not used in the current application scenario, but could be revived during the next on-the-fly reconfiguration.

A subscription instructs the TISS, whether special processing is required for communication activities of a given port. We define four core services that are able to subscribe to ports for their operation.

- In the *normal case* a port is subscribed by the basic communication services (**sCOM**). The processing incorporates the elementary steps of the TISS to establish communication channels. These are:
  - for incoming messages the establishment of a receive window, the skipping of surplus routing information, the storage of data flits, time stamping
  - for outgoing messages the injection of routing information before data flits, and the retrieval of data flits
  - for both directions the look-up of the configuration from the Port Configuration Memory (e.g., port base address), the application of the port synchronization protocol, the generation of the message complete interrupt
- The inter-component channel configuration (**sRC**) is one case demanding for special treatment. Besides the elemental tasks of the normal case, the TISS parses all incoming data according to the reconfiguration protocol and redirects it into the protected memories. Consequently, no data passes the Port Interface. Additionally, it maintains the reconfiguration interrupt.
  The port subscribed by this core services is occupied so that it can not be used by application-software. Thus, any configuration in the Port Configuration Memory by the host is ignored for that port.
- The health reporting service (**sHR**) is nearly identical to the normal case. The only deviation is the appending of the status information and error conditions of the error detection service at the end of a message, which justifies the special processing by the TISS.
- In addition to the normal case the component control service (**sCC**) the TISS parses the metadata of the very first data word of a command message. Besides this, it maintains the component control interrupt.

As the **Subscriber** field is present in each entry of the Port Parameter Memory, it is possible for a given application scenario to assign more than one ports to a core service demanding for special processing in the TISS. Additionally, it is possible to reassign subscriptions during run-time, i.e.,

during on-the-fly reconfiguration. In other words, the assignment of ports to core services for special processing is not hard-coded.

If the application software running in the host wants to find out, which ports are subscribed by which core service at the moment, it has to investigate the **Subscriber** field of each port in the Port Parameter Memory. However, in most cases an application designer might stick to the same port associated for inter-component channel configuration, health reporting service, and component control service even after on-the-fly reconfiguration. In this case the application software can assume fixed assignments.

4.2 Time-Triggered Schedule

The time-triggered schedule plays a vital role in the TISS. It contains the information, when communication activities and task triggers are initiated. To be more precise, the time-triggered schedule describes the instants with respect to the global time base of the common time service, when a task trigger of a burst of some fragment a message is made of is to be launched. As a result, all operations and communication activities are triggered according to the information contained in the time-triggered schedule.

4.2.1 Entries of the Time-Triggered Schedule

Conceptually, the time-triggered schedule is made up of circular linked lists. Each list is associated with a period of the common time service. Each entry belongs to exactly one list and is unique among all other entries.

The information included in an entry refers to either the burst of exactly one fragment of a message that is conveyed in a communication channel or the periodic task activation with respect to the task trigger service. Figure 21 illustrates the layout of an entry of the time-triggered schedule.

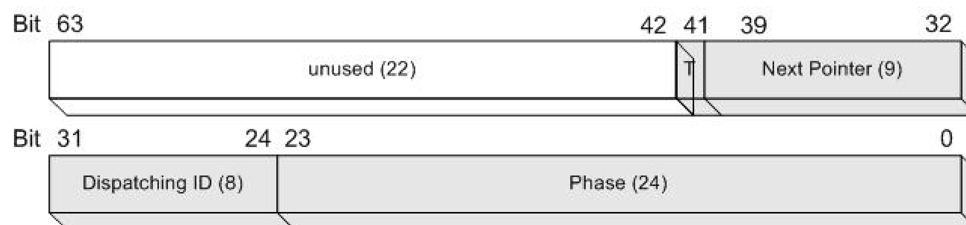

Figure 21: layout of an entry of the time-triggered schedule

In the following we describe each field in an entry of the time-triggered schedule.

The single bit field **T** differentiates between a burst and task trigger that is described in the current entry. If **T = 1** we deal with a task trigger, otherwise it is a burst.

The field **Next Pointer** is, as its name suggests, the pointer to the semantic next entry of the circular linked list. It holds the numeric address of some other entry in the memory of the time-triggered schedule. In case of a single entry in the whole list, it points to itself. If the entry is the last in the list, it points to the very first entry so that it closes the circle. Obviously, its width of 9 bit indicates 512 entries in the memory of the time-triggered schedule.

The interpretation of the field **Dispatcher ID** depends on the setting of **T**. If **T = 1** it denotes the numeric identifier of a task to be triggered in conjunction with the current entry. In the end, the

value of **Dispatcher ID** will show up in the field **CT** of the register file of the task trigger service. If **T = 0** this field contains the numeric address of an entry in the Burst Configuration Memory, which gives further information of the burst to be processed in conjunction with the current entry. As this field is 8-bit wide, it can either describe 256 task references or address 256 entries of the Burst Configuration Memory.

The field **Phase** embodies the temporal alignment relative to the beginning of that period of the common time service, which the circular linked list is associated with. The compare match of that overflow counter and the value given by **Phase** indicates the instant, when a task trigger or a burst has to be dispatched.

4.2.2 Layout of the Time-Triggered Schedule

While the previous section has introduced the layout of entries of the time-triggered schedule, this section explains how these lists physically reside in the memory of the time-triggered schedule.

Each entry of a circular linked list resides in exactly one data word of the physical memory of the time-triggered schedule. Figure 22 depicts the memory map of the time-triggered schedule for a fictitious application scenario. Note that the grouping into rows and columns of the memory is arbitrary. Generally, the memory is sequential, and that grouping contributes to better illustration.

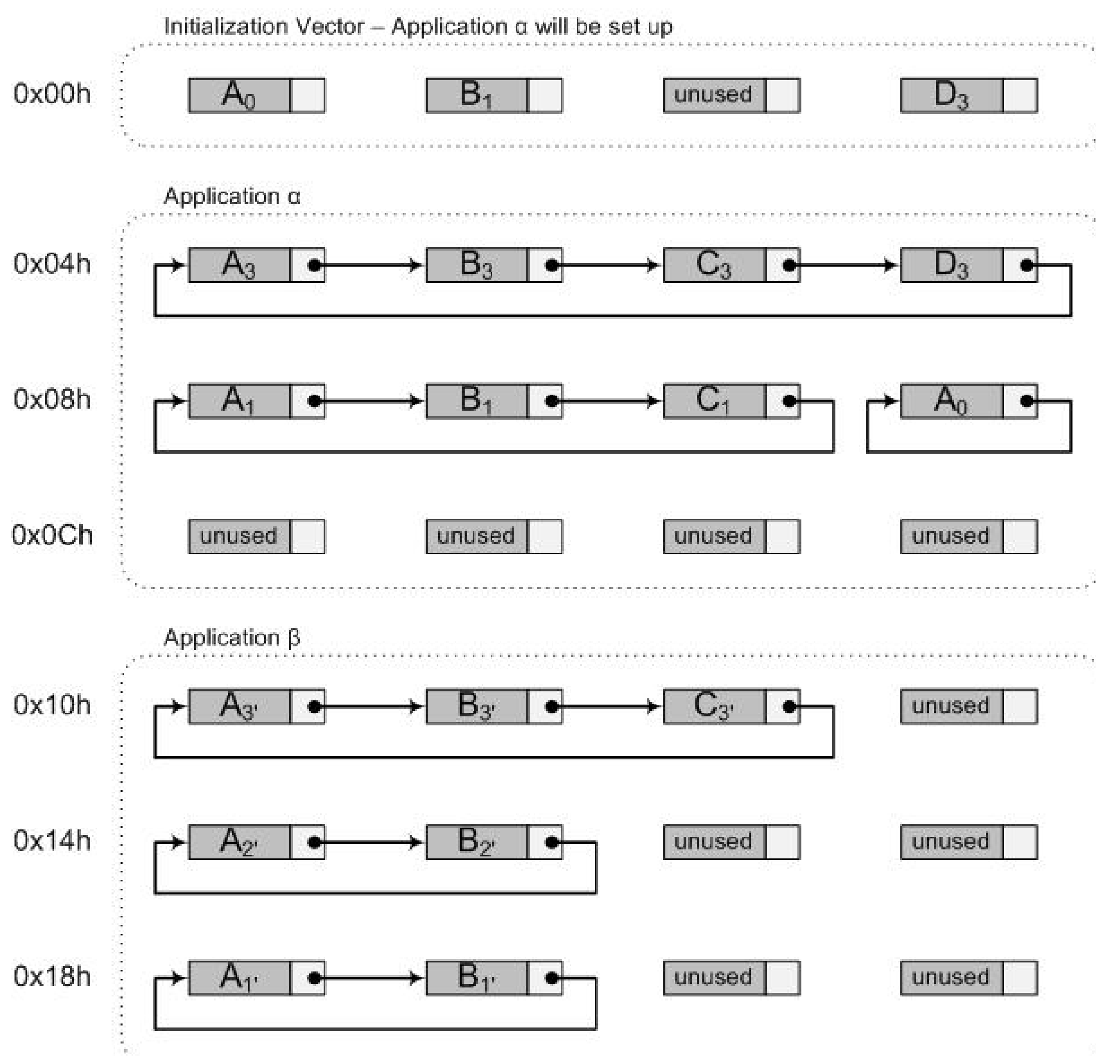

Figure 22: layout of the time-triggered schedule

The time-triggered schedule in Figure 22 comprises 3 sections.

1. the *initialization vector*
2. application α
3. application β

4.2.2.1 Application Sections

The time-triggered schedule is allowed to embed different configurations in *application sections*, whereas each application section corresponds to communication activities and task triggers of a particular application scenario. For a given instant of time, exactly one application section is active, that is, the entries in its lists are traversed and the corresponding communication activities and task triggers are operated.

The existence of several application sections supports the on-the-fly reconfiguration of the inter-component channel configuration. The TISS is able to continue the processing of communication activities and task triggers of the current application section, while the on-the-fly reconfiguration installs a new application section. As a result, no interruption of the basic communication services and task trigger service is required during on-the-fly reconfiguration.

The number of data words occupied by an application section as well as their location in the memory of the time-triggered schedule is not defined. However, the location of the initialization vector is reserved and therefore can not be overlapped by application sections.

Each application section contains at most as many circular linked lists as the number of supported periods of the common time service. Conversely, for a given configuration within an application section some circular linked lists associated with a specific period can be omitted. For instance, in Figure 22 application α uses the periods {3, 1 ,0}, but leaves out period 2. Similarly, application β omits period 0, but includes circular linked lists for all other periods {3, 2, 1}.

We also learn from Figure 22 that within an application section the arrangement of entries need not necessarily be dense. While application α organizes the entries in sequential order, application β skips entries between lists. This arrangement has no impact on the operation of the TISS, although it might lead to a fragmentation of the memory of the time-triggered schedule. The organization of the time-triggered schedule is beyond the scope of the TISS, but in the sphere of control of the TRM during on-the-fly reconfiguration.

Each linked list is circular and closed within an application section. No entry's **Next Pointer** field point to entries of other lists (i.e., periods), nor to entries outside its own application section. Entries within a given list are ascendingly ordered by the value of the **Phase** field, which denotes the phase of the associated communication activity or task trigger. So, the numeric ordering equals temporal ordering. In Figure 22 we express the ordering by alphabetic letters. For instance, in application α the list of period 3 begins at address 0x04 with entry A_3. As $A_x < B_x < C_x$… for any period x, the entry B_3 semantically is the next entry of the circular linked list associated with period 3.

The numeric as well as temporal ordering manifests in the linkage of **Next Pointer** fields. In Figure 22 the numeric ordering of entries matches the physical ordering in the memory. Generally, it is not necessary to have entries physically ordered in the memory, as the linking by the **Next Pointer** always establishes the proper numeric as well as temporal ordering.

4.2.2.2 Initialization Vector

The size and location of the initialization vector is defined in the time-triggered schedule. This entity is relevant for initialization of the dispatching circuitry of the TISS at start-up and reconfiguration. In case of initialization, the initialization vector presents the entry point of traversal for each period. The lists associated with these periods must belong to the same application section.

The initialization vector contains a copy of the first entry to be processed for each period. For some period **x** this copy resides at address **x** of the memory of the time-triggered schedule. If a period is not present in that application section, the corresponding data word of the memory for the initialization vector is also unused. For instance, in Figure 22 the initialization vector pinpoints application α, which omits period 2. Consequently, the data word at address 0x02 is unused.

The initialization vector always consumes as many data words as the number of supported period in the common time service. For example, in Figure 22 we support 4 periods, hence an application section can possess up to 4 circular linked lists. Furthermore, the initialization vector takes the very first 4 data words in the memory of the time-triggered schedule, even though there might be unused entries in the initialization vector due to omitted periods in any application section.

An entry of the initialization vector marks the entry point, where a given period should be started to be processed. This need not necessarily be the “first” entry A with the lowest **Phase** field. For instance, in the initialization vector of Figure 22 the copy of B_1 lays at address 0x01. Consequently, period 1 will be started at B_1. According to its ordering in the circular linked list, after the processing of B_1 the next entry to be handled will be C_1. As a result, at initialization of the dispatching circuitry of the TISS we can begin to execute a period at any entry of its corresponding circular linked list.

4.3 Burst Configuration Memory

The Burst Configuration Memory contains comprehensive information about bursts of fragments of messages. It is not relevant for dispatching of task triggers. It consists of 256 entries. Figure 23 depicts the layout of a data word of the Burst Configuration Memory.

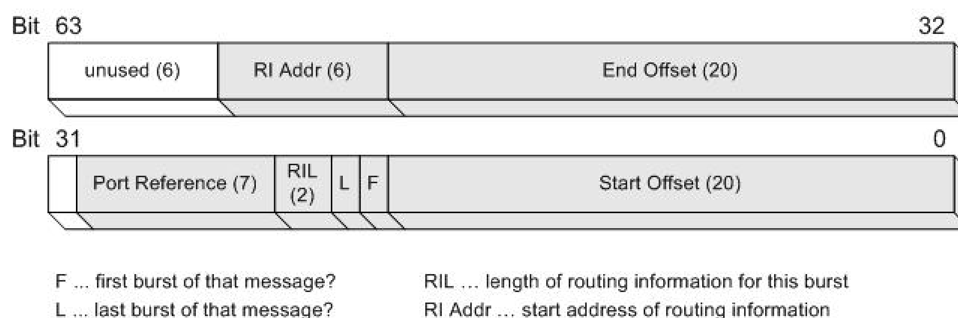

Figure 23: layout of a memory entry of the Burst Configuration Memory

The field **Start Offset** denotes the start offset relative to the port base address of the message, the fragment associated with the burst belongs to, within the physical address range of the Port Interface. The field **End Offset** is its counterpart holding the end offset. Both fields have the same width of 20 bit as the port address in the Port Configuration Memory, so that a burst can span the total address range of the Port Interface.

For outgoing messages the routing information for each burst is stored in the Routing Information Memory. The TISS injects this routing information before the payload to establish the route of the communication channel through the TTNoC. The field **RI Addr** is a reference into the Routing Information Memory to the beginning of that routing information. Its width of 6 bit indicates the size of 64 data words of the Routing Information Memory. In this context, the field **RIL+1** states how many consecutive routing words the routing information is made of starting at **RI Addr**. According to **RIL**’s width a range of 1 to 4 routing flits is feasible. For incoming messages these fields have no meaning.

The field **Port Reference** contains the numeric index of a port to which the current communication activity belongs. The TISS uses this information as pointer to the Port Configuration Memory, Port Parameter Memory, and Port Synchronization Memory in order to find the proper entry in these memories.

If the message of a port must be split into more than one fragment, single bit fields **F** and **L** indentify the very first and the very last of those fragments. If the message is made of exactly one fragment, both fields indicate the value '1' in the same entry of the Burst Configuration Memory. This information is vital for the TISS to trigger the port synchronization and message complete interrupts. The existence of this information implies an ordering of fragments, which belong to the message of the same port.

In this context we impose the restriction that the bursts of all fragments of a message must belong to the same period, i.e., must be linked within the same circular linked list.

4.4 Routing Information Memory

The Routing Information Memory stores the information, how communication channels are routed through the TTNoC. For communication activities of outgoing messages the TISS fetches data words from this memory and inserts them into the TTNoC. Thus, the route of the communication channel is established due to switching of the Fragment Switches. Details of switching in the TTNoC can be found in deliverable D1.6.

The Routing Information Memory consists of 64 data words of 32 bit width, which is the same as the width of the data wires of lanes in the TTNoC. These data words do not posses any particular layout. Instead of a predefined structure, each data words holds a routing flit, which is composed of switching opcodes for Fragment Switches. The routing information that describes the route of a communication channel is spread across at least one routing flit respectively data word of the Routing Information Memory.

4.5 Register File of Protected Memories

Figure 24 shows how the four protected memories are mapped as dedicated register file into the address space of the Control Interface.

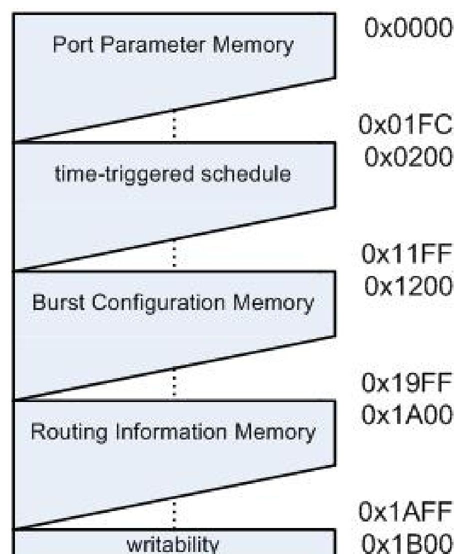

Figure 24: layout of the register file of the protected memories

As the Control Interface has a fixed width of 32 bit, we run into problems when the host wants to access an entry of a memory with wider entries like the memory of the time-triggered schedule and the Burst Configuration Memory. For this purpose, an extra-wide entry is split into a lower (i.e., the lower 32 bit) and upper half (i.e., the upper 32 bit), whereas the lower half gets the lower address and the upper the very next higher word-wise address assigned. For example, bit 0 throughout 31 of entry 0 of the Burst Configuration Memory are mapped to address offset 0x0A00, while the upper half (including unused bits) is available at 0x0A04. As a result, an access to such an extra-wide entry comprises two consecutive accesses to the lower and upper half. Note that Figure 24 already considers this splitting of extra-wide memory entries, because the corresponding memories seem to have the double number of entries than specified.

4.6 The Exception – Write-Access to Protected Memories

As mentioned earlier in this chapter, hosts only have read access to protected memories. Modifications of the contents of these memories can only be undertaken by the TRM during on-the-fly reconfiguration of the inter-component channel configuration.

This policy is sufficient for hosts. Actually, also the TRM is equipped with a standard TISS implementation. When the TRM updates the contents of protected memories of the TISSs of other components in the MPSoC, it is could be necessary to reflect changes by different application scenarios for critical communication channels that are required to uphold the basic infrastructure of core services like the component control and health reporting channels. However, the TISS does not grant local write access to the protected memories of the TRM's TISS so that the TRM is locked out.

To circumvent such situation, it is possible to allow write access to protected memories at design time for selected TISSs only, i.e., for the TISS of the TRM. This setting is hard-coded and can not be revoked or activated at the TISSs of other components. Any host attached to a TISS respectively the TRM can find this setting in the least significant bit of the register **writability** of the register file of the protected memories. If this bit is 1, the host respectively TRM has write permissions to protected memories.

5 The Interrupt Mechanism of the TISS

The core services produce events during their operation, which are worthy to be noticed by the host. The TISS of an ACROSS component favours a notification by means of interrupts rather than polling of status flags that are maintained by core services.

In this chapter we explain the interrupt mechanism implemented by the TISS and how an ACROSS component's host has to cope with the occurrence of some interrupt.

5.1 Interrupt sources

The ACROSS architecture defines 11 core services, whereas in the implementation of the TISS the two basic communication services are grouped together. Hence, we count 10 core services, whereas 6 of them are able to raise distinct sources of interrupts. In the following we describe these 6 interrupt sources.

- *communication interrupt*: This interrupt becomes active if at least one message of some port has completely been transmitted. It is the combination of all status flags of *message complete interrupts*.
- *task trigger interrupt*: the interrupt associated with the periodic activation of tasks of the task trigger service.
- *component control interrupt*: the interrupt associated with the component control service
- *timer interrupt*: the interrupt maintained by the timer interrupt service
- *error interrupt*: This interrupt is raised on occurrence of at least one error condition, i.e., at least one error flag becomes active, as defined by the error detection service.
- *reconfiguration interrupt*: This is the interrupt associated with the *reconfiguration instant* of the inter-component channel configuration.

Background information concerning interrupt sources can be found in the corresponding sections of chapter 2.

5.2 Register File of the Interrupt Control

The registers to handle interrupts are mapped in a dedicated register file of the TISS' Control Interface, which is illustrated in Figure 25. These registers can be organized in two groups:

1. The interrupt control registers of the global interrupt sources.
2. The interrupt control registers of the message complete interrupts.

We also learn from Figure 25 that the implemented interrupt mechanism involves 4 types of registers for the global interrupt sources listed in Table 9, whereas the bit positions across all of those registers are identical. In the following we describe each of these types.

Label	Name	Permissions
INTSTAT	Interrupt status register	read-only
INTACK	Interrupt acknowledge register	write-only
INTMASK	Interrupt mask register	read-/writeable

INTUNMASK	Interrupt unmask register	write-only
MSGSTATn	like INSTAT for message complete interrupts	read-only
MSGACKn	Like INTACK for message complete interrupts	write-only

Table 9: types of interrupt registers

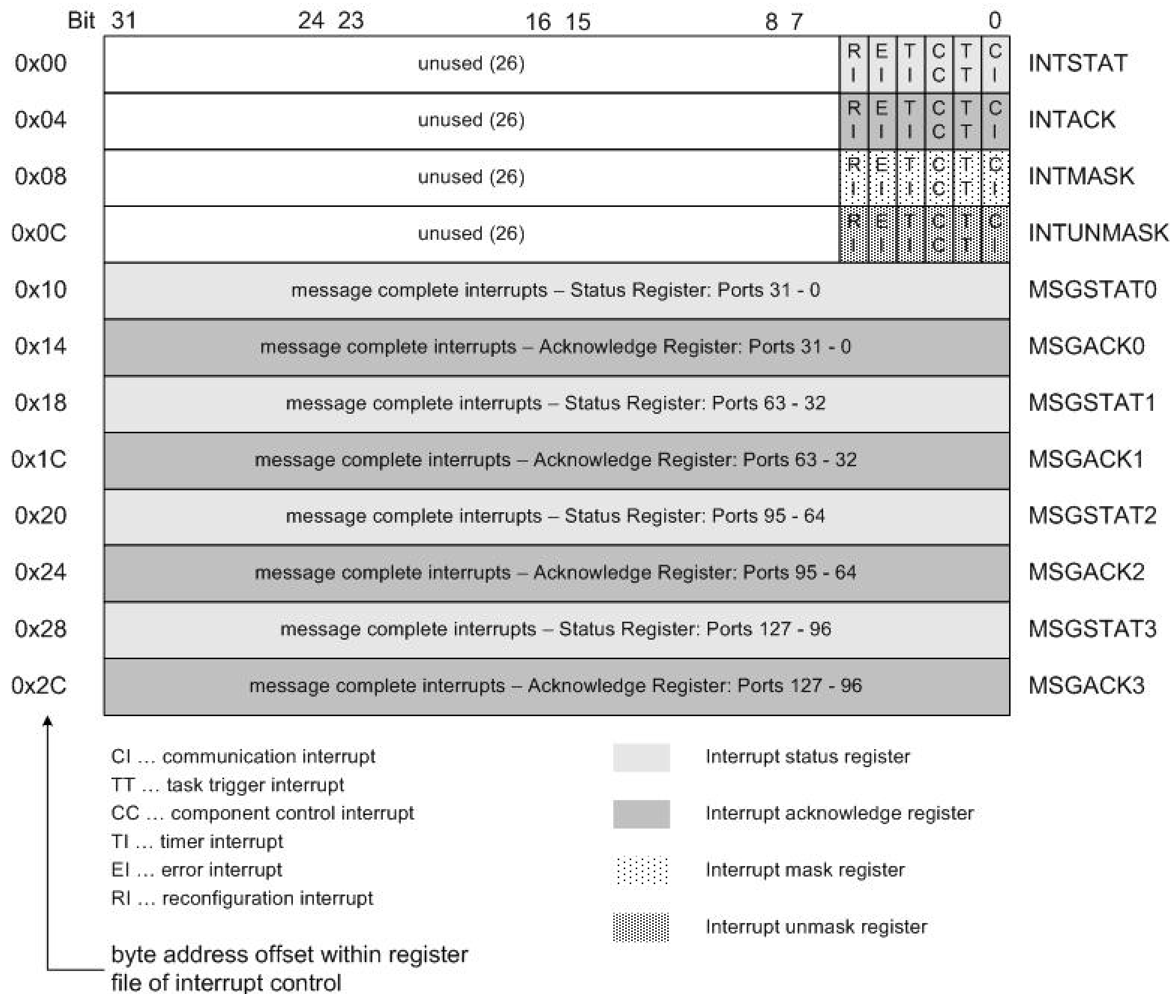

Figure 25: layout of register file of interrupt control

5.3 Handling Interrupt Flags

This section describes how the registers of interrupt control are used to acknowledge the occurrence of an interrupt or to mask an interrupt source. This handling is done by the host and encapsulated by the TISS driver functions associated with the interrupt call back mechanism, as introduced in section 3.4.

5.3.1 Interrupt Status Register (INTSTAT)

The register INTSTAT contains the actual interrupt status flags which cause the notification of the host via the dedicated interrupt lines. This register is read-only, write operations are ignored.

An interrupt status flag for a given interrupt source is set to 1 by the TISS, if the condition for the corresponding interrupt has occurred and the indication of the interrupt source has been enabled by

means of INTMASK. An active interrupt status flag holds the value 1, until it is cleared by means of INTACK.

If an active interrupt status flag already holds the value 1 when the same interrupt condition arrives again, the interrupt status flag remains in its state. There is no further action associated with such a situation.

An interrupt status flag remains unset (value 0) after it has been disabled by means of INTMASK, even though the corresponding interrupt condition occurs.

5.3.2 Interrupt Acknowledge Register (INTACK)

The register INTACK is needed to clear interrupt status flags in INTSTAT. This register is write-only, the return value of a read operation is undefined.

A write operation with value 1 to some bit in INTACK sets the related bit in INTSTAT to 0, if and only if that related bit in INTSTAT has already been 1. Otherwise this write operation has no impact on the related bit in INTSTAT. The clearing of that related bit in INTSTAT happens within the very next clock cycle after the arrival of the write operation at the Control Interface.

A write operation with value 0 to some bit in INTACK also has no impact on the related bit in INTSTAT.

5.3.3 Interrupt Mask Register (INTMASK)

Obviously, the register INTMASK implements a masking mechanism for interrupt sources. An interrupt source can only be triggered and therefore recorded in the INTSTAT register, if and only if the related bit in INTMASK is set to 1.

A bit in INTMASK is set by a write operation with value 1 at the according bit position. If the related bit in INSTAT has been 1 at the arrival of the mentioned write operation (to INTMASK) at the Control Interface, the bit in INTSTAT holds its value until cleared by means of INTACK. A write operation with value 0 to some bit in INTMASK has no effect on that bit. As a result, interrupt masks can not be cleared by accessing the INTMASK register.

After the activation of an interrupt source by the mentioned write operation to INTMASK, an interrupt can be indicated by means of INTSTAT at earliest after the very next clock cycle after arrival of that write operation to INTMASK at the Control Interface.

5.3.4 Interrupt Unmask Register (INTUNMASK)

Similar to INTSTAT and INTACK, where bits in INTSTAT can not be cleared by itself but by an explicit write operation to the other register INTACK, the revocation of an interrupt mask involves an explicit register, namely the register INTUNMASK. The reason for a dedicated “unmask register” is to relief the host of performing cumbersome and performance consuming bit operations when selectively toggling “mask bits”.

A write operation with value 1 to some bit in INTUNMASK causes the related bit in INTMASK to be set to 0. As a consequence, the corresponding interrupt source is regarded as disabled or “masked”. Any future interrupt will be suppressed, i.e., not indicated by the related bit in INTSTAT, at earliest after the very next clock cycle after the arrival of that write operation (to INTUNMASK) at the Control Interface.

If the related bit in INTSTAT has been 1 at the arrival of the mentioned write operation (to INTUNMASK), the bit in INTSTAT holds its value until cleared by means of INTACK.

A write operation with value 0 to some bit in INTUNMASK has no effect to the related bit in INTMASK. The result of a read operation from INTUNMASK is undefined, i.e., INTUNMASK is write-

only because the register does not store the values of each bit and there is no utility in reading that register.

5.3.5 Message Complete Interrupts

The message complete interrupts provide the same mechanism of interrupt status flags and acknowledgement as INTSTAT and INTACK. The message complete interrupts are grouped in pairs of status flag and acknowledgement registers, i.e., MSGSTATn and MSGACKn. As the Control Interface is 32 bit wide, each register pair covers 32 ports, thus leading to 4 pairs for 128 ports as shown in Figure 25.

The bits in all MSGSTAT registers influence the *communication interrupt* bit in the INTSTAT register (INSTAT.CI). Provided that the interrupt source is enabled by means of INTMASK, INTSTAT.CI becomes 1 if at least one bit among all MSGSTAT registers is 1. As long as there is at least one bit in some MSGSTAT register active, INTSTAT.CI will continuously hold value 1, even though it might be cleared by means of INTACK.CI. The clear of that interrupt status flag remains permanent until all bits in MSGSTATn have been revoked.

Similar to INTSTAT, active message complete flags in MSGSTATn must be cleared by write operations to the corresponding MSGACKn register.

Note that there are no explicit interrupt mask and unmask registers for message complete interrupts. The reason for this omission is that the “interrupt enable” field **IE** in the Port Configuration Memory models exactly the same behaviour.

5.4 Interrupt Lines

The TISS drives several interrupt lines that are derived from the interrupt status flags. These interrupt lines are level-sensitive at the native UNI, whereas the level of ‘1’ denotes an active interrupt line.

If a system interconnect requires semantics of edge-sensitivity, the wrapper that translates the native UNI to the target system interconnect must also convert the level-sensitive into edge-sensitive interrupt lines.

The TISS maintains three distinct interrupt lines, whereas two interrupt sources occupy one interrupt line each and the remaining interrupt sources share the third interrupt line. Table 10 describes which interrupt status flags are assigned to which interrupt lines.

<table>
<tr><th>Interrupt source</th><th>Interrupt line</th></tr>
<tr><td>communication interrupt</td><td>0</td></tr>
<tr><td>task trigger interrupt</td><td>1</td></tr>
<tr><td>component control interrupt</td><td rowspan="4">2</td></tr>
<tr><td>timer interrupt</td></tr>
<tr><td>error interrupt</td></tr>
<tr><td>reconfiguration interrupt</td></tr>
</table>

Table 10: mapping of interrupt status flags to interrupt lines

The assignment of distinct interrupt lines to interrupt sources is based on the expected frequency of interrupts. The *communication interrupt* and *task trigger interrupt* are derived from activities that

manifest in the time-triggered schedule. Consequently, their frequency is predetermined and deterministic and probably much higher than other interrupt sources. In order to facilitate a fast dispatching of the associated interrupt service routines, we devote a distinct interrupt line. As a result, the host directly relates these interrupt lines with the interrupt service routines tailored for the corresponding interrupt source. Conversely, it is no longer necessary to evaluate other interrupt status flags in the context of a single interrupt service routine, which unnecessarily consumes processing time.

6 Data types, structures and constants

This chapter lists the data types, structures, constants, etc. as they are used by the TISS driver. There are data types that are platform-specific and must be ported to each target architecture. The platform-independent data types are generic and written in portable C-code. It is possible that such generic data types contain elements of platform-specific data types.

6.1 Platform-specific data types

In fact, platform-specific data types are type definitions ("typedefs") of data types defined by the software framework of the target architecture, i.e., the Altera HAL for NiosII CPUs.

```
typedef alt_u32 tiss_register;
typedef alt_u8 across_portid;
typedef alt_u8 across_errcode;
```

6.2 Platform-independent data types

```
enum regfiles_t {
      regfile_intctrl = 0,     /* interrupt control */
      regfile_protmem = 1,     /* protected memories */
      regfile_commontime = 2, /* common time service */
      regfile_timerint = 3,    /* timer interrupt service */
      regfile_comm = 4,        /* basic communication services */
      regfile_boot = 5,        /* basic boot service */
      regfile_channelcfg = 6, /* inter-component channel configuration */
      regfile_compctrl = 7,    /* component control service */
      regfile_errdet = 8,      /* error detection service */
      regfile_healthrep = 9}; /* health reporting service */

enum interrupt_sources_t {
      interrupt_comm = 0,           /* communication interrupt */
      interrupt_tasktrigger = 1,    /* task trigger interrupt */
      interrupt_compctrl = 2,       /* component control interrupt */
      interrupt_timer = 3,          /* timer interrupt */
      interrupt_error = 4,          /* error interrupt */
      interrupt_reconf = 5};        /* reconfiguration interrupt */

enum hooks_t {
      hooks_msgcmpl = 0,       /* message complete hooks */
      hooks_tasktrigger = 1,   /* task trigger hooks */
      hooks_generic = 2};      /* generic hooks */
```

6.3 Error codes of driver functions

```
enum errcodes_t {
      errcode_success = 0,            /* generic "success" code */
      errcode_invalid_portid = 1,     /* send & receive functions */
      errcode_queue_full = 2,
      errcode_queue_empty = 3,
      errcode_busy = 4,
      errcode_unknown_hooks = 5,      /* management of ISR hooks */
      errcode_invalid_item = 6};
```