Integration of System-On-Chip Simulation Models

Department of Informatics and Mathematical Modelling
Technical University of Denmark

M.Sc. Thesis No.13

Michael Storgaard (s011934)

28th February 2005

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Reaching deep sub-micron technology within the near future makes it possible to implement complex embedded Multiprocessor System-on-Chip (MPSoC) as a single chip solution. Combined with the requirements for short time to market and low production cost, make designs rely on IP core re-usability. To cope with the increasing complexity of the software and hardware design space, the SoC designer rely on simulation tools to be able to make crucial design decisions at an early stage in the design phase; especially related to the SoC communication platform. For efficient and powerful design space exploration, the ultimate simulation tool consists of a library from where the SoC designer can freely select from a variety of different SoC models, representing IP cores at different abstraction level and then be able to integrate these into a common SoC communication platform (e.g. NoC) having the same interface to the different models. Thus constructing a simulation framework for a particular design space can be fully customized, relative to representing the abstraction level of the different IP cores as desired.

This project work contributes to reaching this goal by proposing a methodology for extending a SystemC based high-level RTOS model for MPSoC[7] to support inter-processor communication using OCP2.0 at TL1 and TL0. Also presented is a methodology for configure a simulation framework in a fast and easy manner, based on a configuration file. Further, a new SoC communication platform model is proposed, allowing abstract modeling of different topologies, such as bus and mesh, while still being able to support communication of real data; also at cycle true level. Finally, different design space exploration experiments are presented with the aim of showing the capabilities of the new models.

Preface

The work presented in this Masters thesis has been carried out by Michael Storgaard and supervised by Jan Madsen, Proffessor, Ph.D. Special thanks to Shankar Mahadevan and Kashif Virk for help and support through the project.

Date and My signature

Contents

Chapter 1

Introduction

In embedded system design the SoC communication platform is becoming an important aspect of consideration, due to the increasing numbers of IP cores. Selecting an optimal topology and IP core placement is crucial for the system performance. Thus the SoC designer rely heavily on different modeling techniques and design tools to be able make decisions about the topology, which should be done at an early stage in the design phase. The ultimate design tool for a SoC designer, would consist of a library containing SoC models representing IP cores as well as SoC communication platforms at different abstraction level. Based on this library the designer would have the ability to construct a fully customized simulation framework, integrating the different types of models through a common SoC communication interface. Having a common SoC interface allows for easy and fast model exchange and thus abstraction level refinement, as desired. The flexibility of this methodology is indeed very powerful for design space exploration experiments as well as in-depth SoC communication platform analysis. An example of such a customized simulation framework is illustrated in figure 1.1, with the different model types described next.

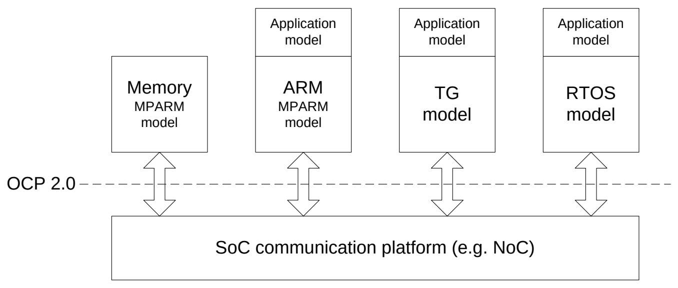


Figure 1.1: Example of a simulation framework integrating different SoC models into a common SoC communication platform.

Building up such library tool set is an ongoing action at IMM, DTU. There are currently following SystemC based models available for MPSoC simulations:

- MPARM[1] which is a cycle-true homogeneous MPSoC simulation framework, modeling IP cores such as ARM-processor, private processor memory, shared semaphore memory as well as Network-on-Chip (NoC) architectures based on AMBA, STBus and cross-pipes.
- OCP2.0 cycle true traffic generator[2], for ARM processor emulation. This model precedes from the MPARM simulation framework.
- Abstract Real-Time Operating System (RTOS) model[3]. The model forms the foundation for the ARTS simulation framework, defining an abstract multiprocessor architecture, operating at transaction level and with applications expressed as task graphs.

The MPARM and Traffic Generator models support SoC communication using the OCP 2.0 protocol. However, it is not possible to use the abstract RTOS model jointly with the other models, for SoC communication platform analysis, since it has no SoC communication interface.

To deal with this issue, this project aiming integration of the abstract RTOS model together with an OCP2.0 based SoC communication platform. The thesis proposes a methodology for expanding the abstract RTOS model to support OCP2.0 based SoC communication (inter-processor communication), related to inter-task dependencies. The methodology emphasizes on modularity to support backward compatibility with the original model as well as making it easy to incorporate support for other SoC communication protocols. Additionally, a methodology is proposed for doing fast, easy and flexible configuration of a MPSoC simulation framework, based on the abstract RTOS model. The foundation for the methodology is based on a configuration file, written in a simple script language, defining design space parameters such as task declarations/partitioning, scheduling policies etc. In conjunction to this, a dedicated parser has been developed.

As an extension to the project, an abstract SoC communication platform model is proposed as well. The model favorers from being able to support communication of real data, at cycle true level, at the same time as having an abstract description of the communication topology (e.g. simple bus or NoC). This model also emphasizes on modularity, making it easy to implement support for new topologies as well as different SoC communication protocols; also at different abstraction levels.

It must be clearly emphasized that this thesis does not cover integration of the abstract RTOS model *together with* an ARM model, in the sense to emulate or support communication with an ARM processor model. Doing so has not been possible, because getting access to the MPARM model was not possible before

[1]Developed at DEIS, University of Bologna.

[2]Developed in corporation with IMM, DTU and DEIS, University of Bologna [14]

[3]Developed at IMM DTU [7]

at a very late stage in the project phase. However, since both module now supports inter-processor communication through a common SoC interface (OCP2.0), implementing this feature is indeed possible.

The rest of this thesis serves to document the work carried out in this project. The report is organized in the following way:

- Chapter 2 presents related work.
- Chapter 3 gives an introduction to the system level description language (SystemC and the Master/Slave library) used for modeling as well as the motivation for this for.
- Chapter 4 gives an introduction to the OCP protocol, highlighting some of the important features. It also gives a short introduction to the SystemC based transaction level library, used for abstract OCP channel modeling.
- Chapter 5 gives an introduction to the abstract RTOS model for MPSoC and highlights its characteristics and features.
- Chapter 6 presents a discussion of the approach used to extend the abstract PE model, based on the RTOS model, to support low-level inter-processor communication, related to inter-task dependency.
- Chapter 7 presents an overview of the new MPSoC simulation framework, based on the extended abstract PE model, by giving a brief introduction to the different modules, their behavior and how they interact.
- Chapter 8 presents the configuration file script language, used for configuring an abstract PE based MPSoC simulation framework.
- Chapter 9 presents the abstract SoC communication platform model, including and overview description.
- Chapter 10 presents three design space exploration experiments, based on simulation frameworks integrating the abstract PE and SoC communication platform models for showing their capabilities.
- Chapter 11 presents implementation specific details for the extended abstract PE model. This also includes modifications and improvements done to the original RTOS model.
- Chapter 12 presents implementation specific details for the abstract SoC communication platform model.
- Chapter 13 wraps up and gives a final conclusion
- Chapter 14 presents suggestions for future work and improvements.

The report is relative long due to the implementation descriptions. Here is a suggested way to this report: Chapter 2,3,4,5 serves as general introductions and may be skipped or skimmed by readers already familiar with related work, SystemC, OCP and the abstract RTOS model. Chapter 6,7,8,9,10 and 11 are essential for understanding the extended abstract PE model and the SoC communication platform model. Chapter 11 and 12 may be skimmed or used as reference for readers not interested in in-depth implementation specific details.

Chapter 2

Related Work

Different SoC models and frameworks have been proposed for MPSoC simulation at different level of abstraction. [12] presents a SystemC based MPSoC simulation framework for analyzing on-chip communication with cycle and bit true accuracy. The framework (SWARM) consists of an adapted version of the ARM Instruction Set Simulator [3] for processor modeling. It also consists of memory, interrupt and semaphore devices as well as interconnection modeling based on AMBA AHB or STBus. The authors demonstrate that the platform is suitable for doing benchmarking and quantitative analysis (performance comparison and architectural design space exploration between AMBA AHB and STBus), based on realistic workloads. [14] focuses on performance improvement for cycle and bit true simulations, using an OCP2.0 compliant traffic generator (TG) for ARM processor emulation. The model precedes from the MPARM simulation framework, which is an extended version of [12], also using OCP 2.0 protocol in the SoC communication interface. The traffic generator (TG) model favorers from being reactive and able to handle unpredictable network behavior like resource contention etc. Based on a reference simulation, using the ARM processor model to emulate, the RTL communication trace is analyzed and a TG program is generated using appropriate tools. The advantage of using a TG is, that the complex application specific details in the IP model is abstracted away, thus reducing simulation time with a factor of 2..4. In [13] an abstract modular RTOS model for MPSoC is presented. It operates at transaction level and uses task graphs for application modeling. The RTOS models basic RTOS services, covering synchronization, resource allocation and task scheduling. It has been implemented using SystemC and the Master/Slave library. The flexibility of the model is clearly demonstrated in [23], where it forms the foundation of an abstract Network-on-Chip (NoC) simulation framework for MPSoC. In the simulation framework, all low level network details are abstracted away and network communication is simulated using message tasks. The NoC communication is managed and modeled by a dedicated communication processor, also based on the abstract RTOS model. Additional, the abstract RTOS model can also be used in conjunction to wireless sensor network modeling as demonstrated in [9].

The model presented in [10] also covers RTOS modeling and is similar to [13]. However, it has been implemented *on top of* SystemC, to overcome the lack of support for modeling dynamic real-time behavior, like task synchronization and preemption. This approach supports even higher level of abstraction (un-timed system specification). It also features true multitask execution as well as power consumption estimation for different scheduling algorithm, available from an associated RTOS library. Similar approach is presented in [8], but this model is based on SpecC [4] as system-level-description-language (SLDL), with extensions added to original SpecC language. Another, yet closely related work, is presented in [16] and consists of a generic RTOS model. The model has been implemented on top of the SystemC kernel, but using a set of generic classes instead. Compared to [8], this model provides higher accuracy modeling of the RTOS and preemption, taking into account parameters such as context switching and scheduling algorithm duration. The model also integrates into a graphical tool set [1], previously developed by the same authors. This tool set features automatic code generation, of SystemC based models, as well as graphical and numerical analysis of the simulation results.

Characteristic for the previous frameworks and models is that only [14] explores the possibility of mixing and integrating different abstraction level SoC models, into a common SoC communication platform. Thus the work presented in this thesis makes good sense, in conjunction to this, by proposing some new modeling methodologies within the field of mixed abstraction level modeling.

Chapter 3

System-Level Description Language

This section gives a short introduction to SystemC and the Master/Slave library, used in this framework for modeling. Please refer to [22] and [21] for more information.

3.1 SystemC

SystemC is a system-level description language (SLDL), intend for Co-design. It is implemented as a set of classes, on top of the ANSI C++ programming language, to support event driven simulation and threaded execution. The methodology of SystemC makes it suitable for creating accurate executable specifications, algorithm exploration, system-level models at multiple abstraction levels. It was introduced in 1999 and had back then close similarities with VHDL and Verilog, thus useful for RTL simulations. With the introduction of SystemC 2.0, the language became more suitable for abstract modeling as well. However the current version of SystemC 2.1 still lacks support for dynamic real-time behavior, found in embedded system, using RTOS. This feature, however, is expected to be implemented in a future release of SystemC (version 3.0) [15].

Today SystemC has grown high popularity and emerged to become an industry standard for system-level modeling.

3.2 Master-Slave library

The Master/Slave class library is an abstract communication channel model for SystemC. The library aiming simulation of SoC platforms, which uses bus communication in the producer/consumer style manner. It supports all abstraction levels, ranging from un-timed down to cycle-accurate. The methodology introduced by the library allows for easy and flexible separation of communication (bus protocol) from behavior (IP core), which is very useful for abstraction level refinement of the communication channel, during the design process.

Chapter 4

SoC communication platform

This section highlights the main features of the OCP2.0 protocol, being used in this project work in the SoC communication interface. Also presented is a brief introduction to the SystemC based Transaction Level (TL) Communication Library, which will be used for modeling an OCP TL1 channel in the project. Further information about the protocol and library can be found in [17] and [18] respectively.

4.1 Open Core Protocol

The Open Core Protocol (OCP), provided by OCP International Partnership (OCP-IP) [2], is a protocol for on-chip synchronous RTL communication, between IP cores. The communication is point-to-point and requires a master and slave device, connected to the channel. The master initiate commands (e.g. read or write requests) to the slave, which in return may provide a response (e.g. response data for a read request). The slave cannot initiates commands. A simple master/slave setup is shown in figure 4.1.

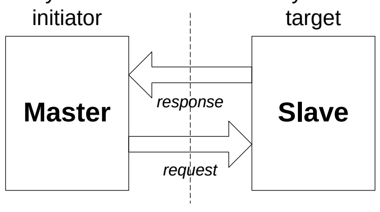


Figure 4.1: Master/slave point-to-point communication

The protocol has gained high popularity, due to its flexible configuration abilities and refinements of data, communication and test signals; all important aspect in today design methodology, focusing on IP core reuse and easy integration. The protocol supports many types of communication schemes, such as simple and burst transactions, multi-threaded out-of-order transaction, pipelined and non-pipelined communications etc. Examples can be found in the OCP Specification [17].

The protocol also provides a methodology for documenting the property of an IP core (address space encoding etc.) and it's OCP interface (signals supported etc.). This is done using an *interface* and *RTL configuration file* respectively, created using a set of predefined conventions. The simplicity of the configuration file makes it easy for the SoC designer to determine if an IP core, for an example, is compatible with a certain OCP configuration.

4.2 OCP Transaction Level Communication Library

The OCP Transaction Level (TL) Communication Library is an OCP channel model for SystemC, provided freely by OCP-IP [2]. The library targeting system level models, using the OCP protocol as a SoC communication platform. It supports modeling at transaction level 1 (TL1) and TL2 [5], which is suitable for close-to cycle true modeling, but significantly faster. The methodology used for channel communication is based on a set of dedicated commands (function calls), making OCP transaction modeling easy, since protocol implementation details are abstracted away. The channel model is very easy to configure (signal wise) and has incorporated a real-time OCP checker, checking for non-compliant OCP transactions.

Members of the OCP-IP community also have access to a set of library extensions, consisting of an OCP monitor and a set of TL adapters. The OCP monitor is used for monitoring the channel and saves the channel state, at each clock cycle, into a file. This format is somewhat similar to a timing trace and can be analyzer either using a text editor or the CoreCreator tool set, provided by OCP-IP. The TL adapters are used for TL adaption between TL0/TL1 and TL1/TL2. However, the adapters need to be customized manually, since the default channel support is restricted to simple configurations only.

Chapter 5

The abstract PE model

The foundation for the MPSoC framework proposed in this thesis precedes from the abstract RTOS model for MPSoC simulations, developed by Virk and Gonzalez [7]. This chapter serves to give an introduction to the abstract *process element* (PE) model, which is based on the abstract RTOS model. Readers already familiar with the model may skip this chapter without any lose of consistence.

Figure 5.1 below the architecture of the abstract PE model.

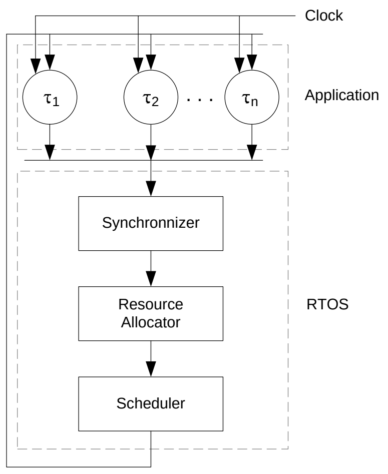


Figure 5.1: Architecture for the abstract PE model

The model works at transaction level and consists of an abstract RTOS, used for modeling basic RTOS services, covering synchronization, resource allocation and scheduling. The applications running on top of the RTOS is modeled using task graphs. Characteristic for the model is the modularity, which makes module exchange an easy matter (for an example exchanging the scheduler module for scheduling algorithm exploration). Modules communicating using high-level messages, based on structs. This approach is described later.

5.0.1 Periodic Task

For application modeling a *periodic task mode* is available. It models periodic execution of a group of instructions. The model support preemption. Figure 5.2 shows the task model, with the different timing parameters described next.

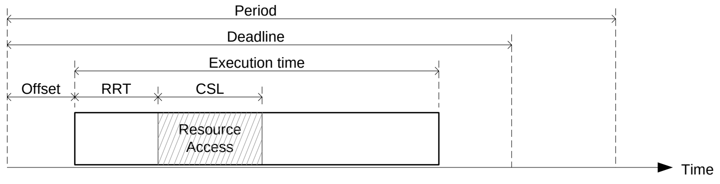


Figure 5.2: Task model timing for the first execution cycle.

Timing constrains

In the model, the exact functionality is abstracted away and instead described using the following set of timing constrains:

- **Execution time**. The amount of time it takes to execute the set of instructions. Determined randomly (uniform) within a specified best-case and worse-case execution time.
- **Offset**. A time offset, determining when the task is ready for being released for execution. This offset is relative to zero-time and is only applicable for the first execution cycle.
- **Deadline**. A time boundary within the execution must complete. The deadline is relative to the release of the task.
- **Period**. The time interval determines when the task should start executing again.

The model also support even more accurate modeling, taking into account context switch overhead, e.g. added by the scheduling algorithm. However it is default not being used.

Resource requirement

A task may requires access to one ore more resource, during execution. Examples of such resources are memory and peripheral devices (e.g. printers). In this model, the abstract description of a task resource requirement is expressed by the following parameters:

- **Resource ID** A number identifying the resource to request.

- **Resource Request Time (RRT)** The time offset, relative to the start of task execution, when the task will request the resource.

- **Critical section length (CSL)** The amount of time the resource will be occupied by the task.

Whenever a running task requires access to a resource, it sends a requests to the resource allocator.

5.0.2 Implementation

The periodic task model has been implemented using a 4-state FSM, as shown in figure 5.3.

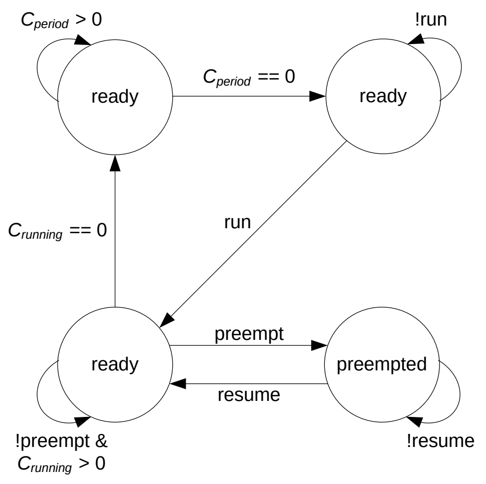


Figure 5.3: State machine for the periodic task model.

State transition depends on control messages from the RTOS and local watchdog timers, used for managing the execution timing constrains. The watchdog timers are C_{period}, $C_{running}$, $C_{deadline}$ associated with task period, execution time and deadline monitoring respectively. C_{period} decrements in all states, $C_{running}$ decrements in running state and $C_{deadline}$ decrements in running state and preempt state. If $C_{deadline}$ reaches zero, before execution finishes, a UI message will be generated, informing that the deadline has been missed. Beside the execution watchdog timers, the model also uses a series of watchdog timers for managing RRT and CSL for each resource requests. They are only applicable and decrement in running state. The meaning of the different states are summarized next.

- **Idle**. Task waiting to release itself. This happens when C_{period} becomes zero. The task issues a READY message to the synchronizer, indicating ready for execution, and goes to ready state.

- **Ready**. Task has been released and waits for execution. When a `RUN` message is received from the scheduler, it goes to running state and execution starts.
- **Running**. Task executing. When $C_{running}$ reaches zero, the execution completes and the task issues a `FINISH` message to the scheduler. If the task receives a `PREEMPT` message from the scheduler, the execution is preempted, and the task goes to preempt state.
- **Preempted**. Task execution has been preempted. When a `RESUME` message is received from the scheduler, the task goes back to running state again and resumes execution.

5.1 RTOS model

The abstract RTOS model consists of three modules: synchronizer, resource allocator and scheduler.

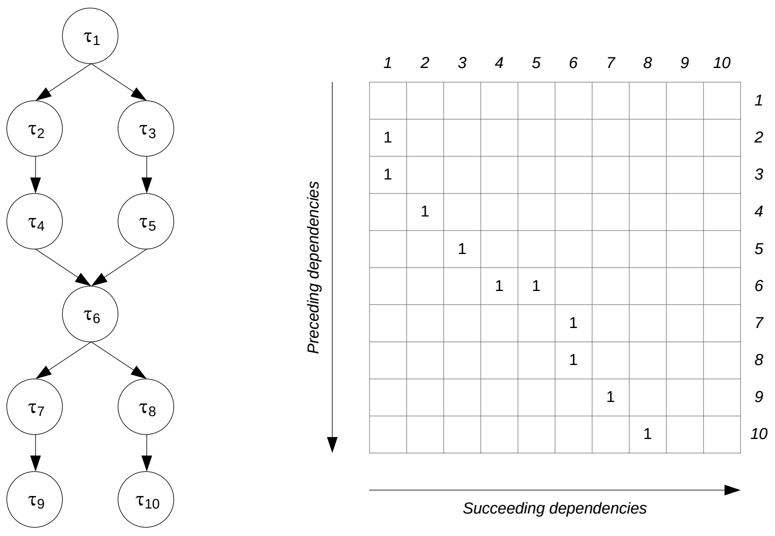
5.1.1 Synchronizer

The synchronizer manages the dependencies between tasks. It ensures that a task is not released for execution, before all data dependencies has been resolved. The current synchronizer implements the Direct Synchronization (DS) protocol, proposed by Sun and Liu [11].

The dependency database

Tasks dependencies are expressed using task graphs. In the synchronizer, task dependencies are managed using a dependency database, somewhat similar to a task graph. The dependency database is a boolean NxN matrix, where N equals the number of tasks. The row and column number maps to the task ID. Columns entries are associated with *preceding* dependencies, while row entries are associated *succeeding* dependencies. If entry (i, j) is true, data dependency exists, thus task i cannot execute before task j finishes execution. Figure 5.4 shows an example of a task graph and the dependency database equivalent.

Tasks ready for execution, but with unresolved data dependencies, are kept in a pending task queue. Each time a running task finishes execution, the queue is checked up against the dependency database to check if any pending tasks can be released for scheduling. When task j finishes execution, the dependency database is updated, by clearing (setting to false) all row entries in column j. Checking if all dependencies have been resolved for task i is done by performing an OR operation of all column entries in row i. If the result is false, all dependencies have been resolved and task i may be released to the scheduler for execution scheduling. Otherwise all dependencies have not been resolved yet and the task stays in the queue.



	1	2	3	4	5	6	7	8	9	10
1										
2	1									
3	1									
4		1								
5			1							
6				1	1					
7						1				
8						1				
9							1			
10								1		

(A) Task graph (B) Dependency database equivalent

Figure 5.4: Task graph and dependency database equivalent

5.2 Resource Allocator

In real-time systems, resource contention often occurs, since multiple tasks are competing over the same shared resource. Typically these resources are non-preempt able, which means that lack of resource allocation eventually could lead to data corruption, in situations with resource contention. In conjunction to this, incautious management may lead to unbound priority inversion; a situation where a low priority task blocks for a high priority task, because the high priority task waits access to a resource currently occupied by the low priority task.

The resource allocator models the protocol for managing these situations. It cooperates with the scheduler and ensures that only one task can have access to a shared non-preemptive resource at any time. Whenever a running task has a resource request, it sends a request message to the resource allocator. The resource allocator either grant resource access or refuses the request, causing the scheduler to preempt the task, until the resource becomes available. The protocol implemented in the current model is a simplified version of the Basic Priority Inheritance protocol, suggested by Sha, Rajkumar and Lehoczky [19]. In conjunction to this, the current implementation of the resource allocator does not support nested resource requirement.

5.3 Scheduler

The scheduler manages the real-time scheduling of task, ready for execution, based on the task priority. All tasks ready for execution are kept in a queue and sorted with respect to their assigned priorities. Currently RM and EDF scheduling is available for the model. The characteristics of RM and EDF scheduling is summarized below.

- **Rate-Monotonic (RM)**. Highest priority assigned to the task with the *shortest period*. The priority is *static*, meaning that the priority of a task waiting for execution does not change.
- **Earliest-Deadline-First (EDF)** Highest priority assigned to the task with the *closest deadline*. The priority is *dynamic*, meaning the priority of a task waiting for increases each clock cycle, since the deadline is getting closer.

5.4 Communication link and the message struct

Communication between the different modules in the PE module is based on `sc_link_mp` communication link, provided by the SystemC Master/Slave library [21]. The module communication is based on high-level struct messages. Table 5.1 shows the struct encoding and gives a brief description of the different entries. Depending on the receiver of the message and the action type, some fields are not applicable. Additionally, table 5.2 described the different types of high-level messages, issued by the task and RTOS model, identified by the `comm` entry.

Type	**Name**	**Description**
`unsigned int`	`messageID`	Receiver of the message (e.g. task or synchronizer)
`unsigned int`	`snum`	Target scheduler and resource allocator
`unsigned int`	`tnum`	Task ID
`unsigned int`	`comm`	Action type (e.g. RUN or READY)
`unsigned int`	`resourceID`	Resource ID
`unsigned int`	`tper`	Task period
`unsigned int`	`tdl`	Task deadline
`unsigned int`	`priority`	Task priority
`char*`	`text`	A message describing the action. For monitoring purpose.

Table 5.1: High-level message struct encoding.

Action type	**Producer**	**Consumer**	**Description**
`READY`	Task	Scheduler	Task, `tnum` notifies the synchronizer/scheduler, that it is ready for execution.
`RUN`	Scheduler	Task	CPU time has been granted by the scheduler. Execution of task, `tnum` may start.
`REQUEST`	Task	Resource allocator	Task, `tnum` requests access for resource, `resourceID`. Issued during running-state when RRT has been reached for this resource.
`GRANT`	Resource allocator	Scheduler	Resource, `resourceID` requested by the running task, `tnum` has been granted and execution may continue.
`REFUSE`	Resource allocator	Scheduler	Resource, `resourceID` requested by the running task, `tnum` is already occupied by another task. The scheduler must preempt execution of the task.
`PREEMPT`	Scheduler	Task	The running task, `tnum` must preempt execution, since a higher priority task has been released for execution or a resource request has not been granted.
`RESUME`	Scheduler	Task	The preempted task, `tnum` must resume execution now.
`FINISH`	Task	Scheduler	Task, `tnum` notifies the scheduler that execution has completed.

Table 5.2: High-level message type descriptions.

5.5 Monitor module

Not shown in figure 5.1 is the monitor slave-module, connecting to the different communication links. The module monitors the real-time state of the system-level module, during simulation. Thus messages issued by the different module triggers the monitor to prompt an associated UI message to the screen. As such the monitor module is not a part of the model and may be left out. Missed deadlines will still be reported by the periodic task module.

Chapter 6

Inter-processor communication methodology

This chapter discusses the methodology used to make the abstract PE model, presented in chapter 5, supporting inter-processor communication at a lower transaction level. The discussion presented forms the foundation for the implementation.

6.1 Application partitioning

In a distributed multiprocessor system, application partitioning is a very important aspect of the design space exploration, since it concerns optimizing and balancing the execution of the different applications running on top of an architecture. However, partitioning requires inter-processor communication, due to the data transferring between the partitioned parts of the application. For an example, selecting a multi-processor architecture using slow processors can reduce the product cost, but may cause an application not to meet its deadlines, if it is to be executed on a single processor. For that matter task partitioning is essential, if the application allows for parallel execution of some tasks.

Modeling this in a MPSoC simulation framework, using the abstract PE model from chapter 5 for processor modeling, would be equivalent to partitioning a task graph with parallel branches onto multiple processors. However, when the models interface to a SoC communication platform model having a low level interface (e.g. OCP2.0 TL0), transmission of dedicated data is required to do accurate modeling. This model illustrated in figure 6.1, showing a partitioned task graph and two PE's connecting to SoC communication platform. The selected partitioning requires inter-processor communication, since data dependency exists between τ_1 and τ_2 and both are mapped onto different PE's. From a high-level perspective, the inter-processor communication can also be considered as a task and is in the example identified by τ_{io}.

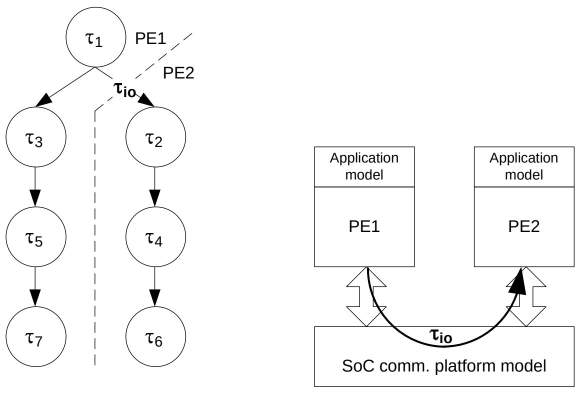


Figure 6.1: Example of task graph partitioning in a MPSoC simulation framework using the abstract PE model.

6.2 SoC communication interface extension modules

Making the abstract PE model support low-level inter-processor communication, requires a dedicated communication interface. The chosen approach has been to refine this module into an *IO device* and *IO task* model, added on top of the existing model. This approach illustrated in block diagram in figure 6.2

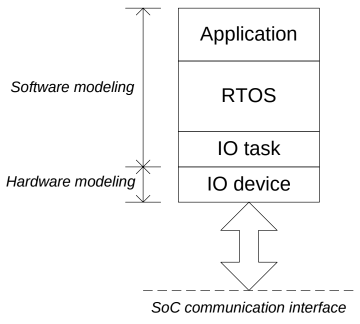


Figure 6.2: Block diagram showing the abstract PE module including IO task and IO device modules for SoC communication support.

The IO device model connects physically to the SoC communication platform thus modeling a hardware IO port and managing the communication protocol. The IO task models an IO device driver, controlling the IO device whenever there is an inter-processor communication event (receiving/transmitting). It handles protocol at application level, which here consists of encoding/decoding of data to/from the SoC communication platform, being synchronization messages between tasks with inter-dependencies. The approach is illustrated in figure 6.3.

When there is preceding inter-task dependency, the RTOS issues synchronization message to the IO task, containing information relevant for the inter-processor communication. Based on this message the IO task encodes a certain traffic patter,

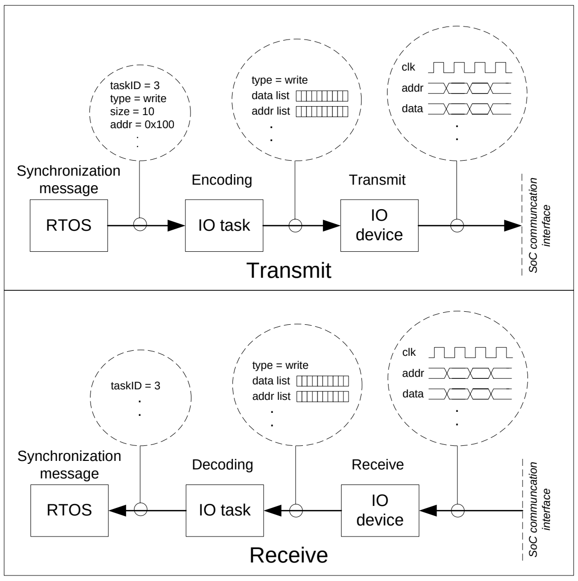


Figure 6.3: Block diagram showing the abstract PE module including IO task and IO device modules for SoC communication support.

forwards this to the IO device, which starts the transmission. The procedure for receiving is just the other way around.

6.2.1 IO task synchronization and execution

Integrating the IO task with the abstract RTOS model requires some small extensions to the synchronizer to support messages to/from the IO task. It also requires some extensions to the message structs (figure 5.1, page 26) as well as the periodic task model. Below is a general description of the approach used for synchronizing the IO task execution, in conjunction to inter-processor communication and inter-task dependency handling. Understanding this description requires familiarity with the behavior of the abstract RTOS model.

Transmit data

A task having preceding inter-task dependencies must issue a `SOC_TRANSFER` message when it completes execution. This new message type notifies task completion (equal to a `FINISHED` message) but *also* that an inter-processor communication event must start. Further, it must contain inter-processor communication related information such as transfer type (write, read or response), data transfer size, target PE addresses etc. The message should cause the synchronizer to release the IO task for execution immediately afterward. The scheduler starts the execution of the IO task and the inter-processor communication starts by interacting with the IO device. Any local pending task must not start executing before the inter-processor communication event has completed, which is identified by a `FINISHED` message issued from the IO task. This means that the IO task has the highest priority and is non-preemptive. The duration of the IO task execution depends on data transfer size as well as bandwidth.

Receive data

When data is received from the SoC communication interface, the IO task sends a `READY` message to the synchronizer, notifying it is ready be process data. This should cause the synchronizer to release the IO task for execution immediately afterward. Any running tasks should be preempted by the scheduler, if no buffering mechanism has been implemented in the IO device model. The IO task executes until the request/response phase completes, after which it issues an `FINISHED_EXT` message to the synchronizer. This new message is similar to a `FINISHED` message, except that it is associated with the non-local task, initiating the inter-processor communication and not forwarded to the scheduler. Also, the non-local task ID is decoded by the IO task. The message should causes the synchronizer to release any pending tasks having succeeding dependencies to this non-local task. After having issued the `FINISH_EXT`, the IO task completes execution by issuing a `FINISHED` message and any pending task may start executing afterward.

Figure 6.4 shows how the described approach applies to the timing of two task having inter-dependencies, where $\tau_1 \prec \tau_2$ and $\tau_1 \mapsto PE1$ and $\tau_2 \mapsto PE2$.

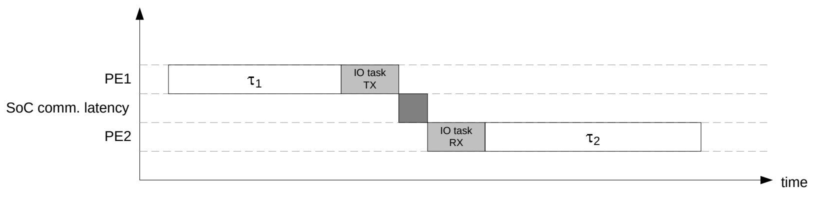


Figure 6.4: Timing for inter-task dependency.

To summarize, the important characteristics related to the integration of the IO task extension of the RTOS model is listed below:

- The synchronizer does not have any prior knowledge about when the IO task is going to be launched, in the sense that it is encoded into the dependency database. Thus the IO task can be considered as being *dynamically* released for execution, relative to the message received from a task or from the IO task itself.

- SoC communication specific information (transfer type, data transfer size etc.) is stored in the task having preceding inter-task dependency. Thus any tasks having preceding inter-task dependency need to be configured before simulation starts.

- The message struct must be expanded to carry inter-processor communication related information.

More implementation specific details regarding the RTOS model and the message struct expanding is presented in section 11.1, page 87, while implementation specific details for the IO task is presented in section 11.5, page 108.

6.3 Task graph abstraction level refinement

At the SoC communication interface, the abstract PE model must be able to support read and write request and response. Response means in this context the returned data to a read request.

The precedence nature of a task graph is somewhat equivalent to a write request, when considering an edge to be associated with data transfer. Thus a SoC communication event related to an inter-task dependency can easily be modeled using a write request. However, a task graph, like the one in figure 6.1 does not obviously support read requests, since this would requires bidirectional edges, for request and response phase respectively. An elegant solution to this problem is to use end-to-end task, which is just an abstraction level refinement of a task graph.

6.3.1 End-to-end task

An end-to-end task is series of subtasks, connected in a chain. The definition is an extension of the existing basic periodic task model, to make it more suitable for distributed systems modeling [20]. It can be considered either as clustering a group of preceding tasks together or refining the functionality of a task, into even smaller subtasks. However, it is allowed for an end-to-end task to have only one subtask. In this special case it is identical to the original periodic task model. From a low-level perspective, an example of an end-to-end task, consisting of 4 subtasks, could be to (1) generate some data in PE1, (2) transfer the data to PE2, (3) process the data in PE2 and (4) output the data to a peripheral device, connected to PE2 (e.g. printer). Below summarizes the main formal definitions of an end-to-end task [20]:

- An end-to-end task, T_i consists of a series of subtasks, connected in a chain. Subtasks are always executed in a precedence order. Thus subtask $T_{i,j+1}$ cannot execute before subtask $T_{i,j}$ completes execution.
- The end-to-end deadline for T_i is *relative* to release of the first subtask, $T_{i,1}$.
- The execution time of a subtask, $T_{i,j}$ is bounded and must not exceed a maximum execution time, $\tau_{i,j}$.
- All subtasks have the same priority.
- Subtasks are statically assigned to PE's.

Figure 6.5 shows an example of an end-to-end system consisting of four subtasks mapped onto three PE's.

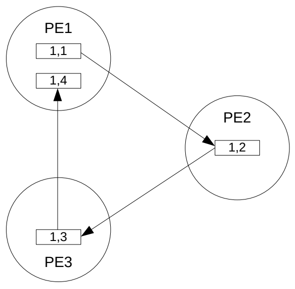


Figure 6.5: Example an end-to-end system.

Since a clustered group of subtasks always belongs to the same task group (end-to-end task), T_i, the abstraction level provided from this, makes the data dependencies between subtask suitable for modeling read transfer. However, some restrictions apply to usage and mapping of subtasks, when used for read transfer modeling. These are summarized below:

1. A read transfer is always associated with three adjacent subtasks: $T_{i,j}$ triggers the read request, $T_{i,j+1}$ receives the request and generates the response data, $T_{i,j+2}$ receives the response data.

2. A subtask, $T_{i,j}$, triggering a read request must always be located in the same PE as the subtask, $T_{i,j+2}$, receiving the response data.

3. A read transfer (covering request and response) must be kept within the same end-to-end task, T_i. Thus triggering a read request after the third last subtask is not allowed, since a complete read transfer requires three subtasks, as stated in 1. This also means that an end-to-end task must have at least three subtasks, to model a read transfer.

Dependencies between subtasks can also be used to modeling write transfer. Here there are no restrictions with respect to usage and mapping of subtask, since only two tasks with dependencies are required (a producer and consumer task).

Figure 6.6 illustrates an example of an end-to-end system, consisting of four subtask, modeling write and read transfers.

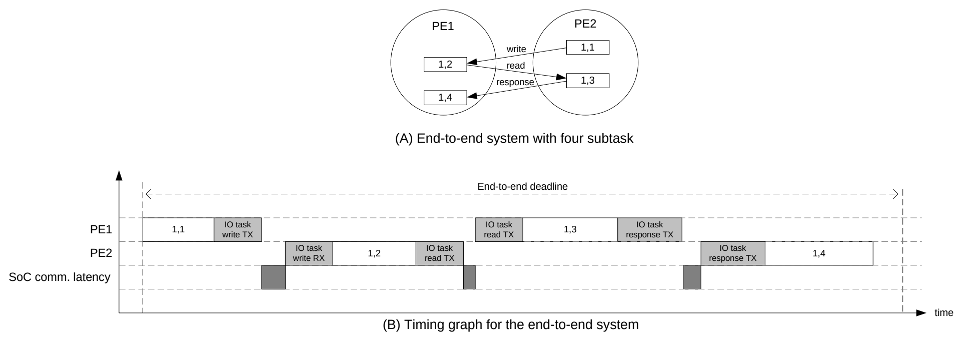


Figure 6.6: Example of an end-to-end system modeling write and read transfers.

6.4 Inter-dependency synchronization protocol

To manage synchronization between tasks having inter-task dependency, a set of simple rules have been defined for the address encoding as well as the data encoding, related to response data. These rules describe the protocol at application level, implemented by the IO task.

6.4.1 Task ID encoding

Each subtask has an unique task ID. This task ID carries information about the (end-to-end) task group ID as well as the subtask ID. The task ID encoding has been selected in such way that the lower and upper bits define the task group ID and subtask ID respectively. Figure 6.7 shows an example for a 16-bit task ID. In this example the subtask ID is defined by bit[0:3], which allows addressing up to 15 subtasks (subtask ID equal to zero is not allowed).

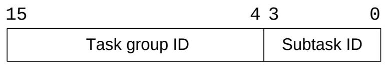


Figure 6.7: Example of a 16-bit task ID encoding.

6.4.2 Address encoding

Task having succeeding inter-task dependencies, related to a request, rely on transfers to be done to a particular location in the address space, assigned to the PE. This address location is always relative to the task ID.

The address encoding is very simple and defined as *the sum of the SoC communication base-address of the target PE and the task ID of the subtask, issuing the requests*. If it is a burst request, the address remains constant.

Example: Subtask, $\tau_{1,1}$ finishes execution and triggers a write request to subtask, $\tau_{1,2}$, located in a PE having a base-address of 0x100h. The task ID for $\tau_{1,1}$, using 4-

bit subtask encoding, is 0x11h. Thus the address associated with the write request becomes 0x111h. When the target PE receives the request, the IO task finds the task ID of the non-local subtask, simply found by doing the reversed procedure (i.e. subtracting the PE base address from the address, associated with the request).

6.4.3 Data encoding

Since a task in the abstract PE model does not implement any functionality, the data to transmit for write transfer are dummy (e.g. zero or random). However, for a response, the transmitted data must equal the task ID of the subtask, issuing the response. This applies as well to all data packets, in multiple responses (burst read).

Chapter 7

MPSoC framework overview

This chapter gives an introduction to the new MPSoC simulation framework, based on the abstract PE model, extended to support low-level inter-processor communication. A brief introduction to the different new modules and extensions done will be presented. The aim is to give an overview of the framework, before presenting the implementation specific details in the following chapter.

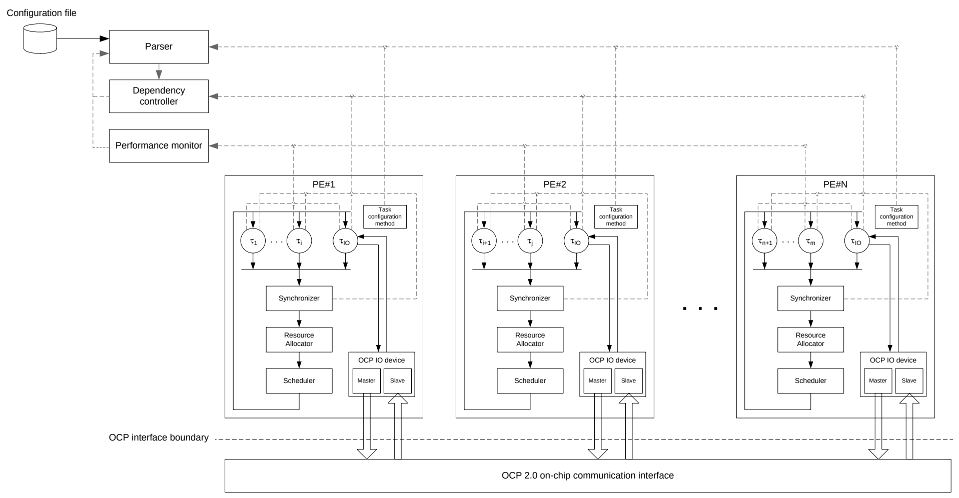


Figure 7.1: Simplified framework block diagram

Figure 7.1 shows simplified block diagram of the framework; here with N PE's instantiated. Solid lines between objects are `sc_link_mp` communication channels while dotted lines indicates objects access through pointers. Relative to the block diagram with figure 5.1, presented in chapter 5, an abstract PE has been extended with an IO task and an OCP2.0 compliant IO device model.

At top level, three other new modules have been incorporated: a parser, a global synchronization database and a performance monitor. A single instance of each of these modules connect to all PE. This connection is established through pointers to the modules, provided to the PE module constructor, during object creation. Through the pointers, different public methods are accessed in the modules.

7.1 Top-level modules

The top level module combines the different modules in to a structural, defining the simulation framework as illustrated in figure 7.1. However, this abstract PE based MPSoC simulation framework relies on an OCP2.0 based SoC communication platform to be complete, unless two PE modules are connected in a back-to-back configuration. Different examples of simulation framework configurations are found on the enclosed CD-ROM in `/ARTS_Model/builds`. These are:

`pe_ocp_tl0/`	Two PE's connected in a back-to-back configuration, using OCP2.0 TL0
`pe_ocp_tl1_clk/`	Two PE's connected in a back-to-back configuration, using OCP2.0 TL1
`example1/`	Two PE's connected to a OCP2.0 TL0 bus.
`example2/`	Four PE's connected to a OCP2.0 TL0/TL1 bus/1D mesh/2D mesh (mixed interface).
`example3/`	Nine PE's connected to a OCP2.0 TL0 bus/1D mesh/2D mesh, using OCP2.0 TL1.

Example 1,2 and 3 are based on the SoC communication platform model, to be presented in chapter 9, page 53. They are also being used in the design space exploration experiments, presented in chapter 10, page 63.

7.2 Parser

In the original abstract PE model, RTOS configuration and task graphs were assigned statically in the sense that they were hard-coded. It meant, for an example, that whenever a task graph modification was required, the model had to be rebuild again. To avoid this very time consuming step and to introduce overall greater configuration flexibility, a parser module has been developed.

The parser accepts a configuration file as an input, written in a simple script language. This file defines the boundaries of a simulation with respect to parameters such as task declarations/partitioning, resource requirements as well as RTOS configuration (selection of scheduling policy etc.) for the different PE's. It also contains other parameters such as SoC communication address space assignment, data logging filename declaration etc. An example of a configuration file is found in figure 8.8, page 51. If parsing of a configuration file is successful, the different parameters can be obtained via dedicated public methods and then used for dynamic object creation (e.g. task modules) etc, before a simulation starts.

Section 11.3, page 99 presents the implementation specific details for parser module.

7.3 Dependency controller

The dependency controller module manages the database, describing the dependencies between tasks, assigned to a simulation. It can be considered as a *global dependency database*, since it connects to all synchronizers. A synchronizer access module when a database entry has to be cleared (task finished) or when a dependency-resolved check is performed, to see if a task can be released for execution. In the original synchronizer, the database was located locally in a synchronizer. However this approach is only suitable for intra-task dependencies and will not work for inter-task dependencies, unless the synchronizer is common to all PE's or the synchronizer is modified significantly. To maintain a modular approach and still keep the original simplicity of the synchronizer, the approach has been to implement a global synchronization database module, added on top of the existing synchronizer. Database access is done indirectly through method calls to the dependency controller module, using a pointer. This pointer is provided to the synchronizers, during object creation. Using this approach, only very few changes have been required in the original synchronizer (e.g. removal of the dependency database and exchanging some functionality with methods call to the dependency controller module).

Another problem with the original synchronizer was, that it did not allowed periodical execution of task graphs: once a task graph completed, the dependencies were lost, and uncontrolled and concurrent task execution would follow afterward (if the tasks were periodically). This problem has been solved in the new dependency controller module, since the dependency database for a task graph is restored whenever the task graph execution completes.

In conjunction to this, a new *task blocking/unblocking* feature has been implemented. That is, a task *with* dependencies will automatically block itself, after completed execution. By blocking meaning that a task cannot issues a `READY` message to the synchronizer, once it has completed execution. This is to avoid, that a task does not accidentally starts executing again, if the task period becomes shorter than the total task graph execution time[1]. Unblocking is managed by the dependency controller and is initiated immediately after a task graph completes. When this happens, all tasks belonging to the task graph gets unblocked. This is accomplished by accessing a dedicated method in the periodic task module for this purpose. In conjunction to this, the dependency controller has a database containing pointers to all tasks objects. The task pointer database in being initialized, during task object creation in the different PE's. This is done by passing a pointer to the task object, from the PE to the dependency controller, as soon as the task object has been created.

Section 11.7, page 128 presents the implementation specific details for the dependency controller.

[1]This would otherwise happens, since the dependencies remain lost, until the database is restored again.

7.4 Performance monitor

The performance monitor module serves to monitor different figures, covering PE performance and the end-to-end deadline for task groups with multiple subtasks. The figures are a part of the data logging and useful when doing design space exploration. All tasks and PE's access this module through a pointer to the object.

7.4.1 PE performance

PE performance covers *utilization* and *IO task execution* figures.

- **Utilization** is a measure for how efficient a PE is being used, defined as the ratio between the no.of.clock cycles, used for task execution and the total no.of simulation clock cycles. Thus an utilization of 1 would indicate a PE has been in use for the entire simulation period, while an utilization of 0 would indicate that a PE has not been used at all.

- **IO task execution** covers a series of activity figures, related to inter-processor communication. They relate to the usage of the IO task for a particular action (e.g. write transmit/receive etc.). The figure for a particular action is defined as the ratio between the no.of clock cycles, used for this action and the total no.of simulation the PE has been in use.

The PE performance figures are calculated, based on activity reporting done by the tasks, when execution starts and finished. This is done by calling dedicated reporting methods in the performance monitor module.

7.4.2 End-to-end deadline

The performance monitor module also monitors the end-to-end deadline for task groups with multiple subtask. For this purpose, it keeps a database containing information about task groups having multiple subtask (subtask count and end-to-end deadlines). This database is initialized before simulation starts, by fetching the information from the parser module. The database is being updated whenever a task becomes ready or finishes execution. This is done by calling a dedicated reporting method in the performance monitor module, causing the database entry for the particular task group to be updated. At each clock cycle the database is checked to see if any end-to-end task groups have missed their end-to-end deadline. If a deadline has been missed (that is not all tasks have completed execution), the module reports missed end-to-end deadline.

The performance module is not mandatory and may be left from the simulation framework, if end-to-end task deadlines or PE performance figures are without interests.

Section 11.8, page 134 presents the implementation specific details for the performance monitor module.

7.5 IO task

The IO task models an IO device driver. It is used for protocol management at application level, in conjunction to inter-processor communication. This covers encoding/decoding of synchronization messages between local and external tasks, having preceding/succeeding inter-task dependencies. It is based on the protocol described in section 6.4, page 35. IO task execution follows the approach described in section 6.2.1, page 31.

Section 11.5, page 108 presents the implementation specific details for the IO task module.

7.6 IO device model

The IO device models the hardware IO device, implementing the protocol used in the SoC communication interface. In this project the target protocol is OCP 2.0 and an IO device model has been developed for TL0 and TL1 respectively. Both models have a fully multi-threaded OCP interface and are configurable (signal-wise), relative to the channel they connect to. An IO device consists of an OCP master and slave, to handle write and read requests. Further, buffers have been implemented for received write and response data. A buffer exists for each thread and the sizes are configurable. Usage of buffers allows for out-of-order thread execution as well as IO task prioritizing, related to receiving data (not considered in this framework).

Section 11.6, page 115 presents the implementation specific details for the IO device modules.

7.7 IO task-IO device communication link

The communication link between the IO task and IO device is based on two `sc_link_mp` channels for transmitting/receiving messages between the modules. The channels are not used for transporting physical, data related to inter-processor communication (e.g. address/data), but used for high-level interrupt-like messages only. It means that whenever a *new* inter-processor communication event starts (e.g. a new request phase), a message will be send from the IO task or IO device or vise versa. Access to the physical data and addresses is provided *indirectly* through pointers, encapsulated in the message. These pointers point at buffers (deque objects) from where address and/or data can be fetched. The advantage of this approach is speed improvement, due to reduced `sc_link_mp` channel activity.

Section 11.4, page 106 presents the communication link channel approach in more detail.

7.8 Periodic task model

Small extensions have been added on-top of the original periodical task model. These are summarized below:

- **Self-blocking**. Means that whenever a task with dependencies finished execution, it cannot start executing again, before it gets unblocked by the dependency controller. Self-blocking has been implemented for synchronization reasons and previously been described in section 7.3. Enabling/disabling of self-blocking as well as unblocking is controlled through calls to dedicated methods in the periodic task module.

- **Dynamic resource requirement allocation** This is a new improvement, allowing support for a fully user defined number resource requirements. The original task model always had three resource requirements, no matter what.

- **Inter-dependency configuration** A task having preceding interdependencies must be configured to initiates an inter-processor communication event, when task execution finishes (see also section 6.2.1, page 31). In conjunction to this, it must hold all information related to the this (transaction type, data transfer size, target PE addresses etc.). For this matter, a method has been implemented for passing the SoC transaction information onto the task, which will be stored in a database. Task configuration is done, before the simulation starts and is managed by a task configuration method in the PE module.

- **End-to-end task identification** Due to the new support for end-to-end task, a task is now defined by group ID and subtask ID.

7.9 PE module

The PE module connects the different submodules into a structural, forming the PE system-level model. RTOS modules and tasks are selected and created dynamically, relative to the declarations done in the configuration file. Module objects are created and connected in the PE module constructor.

The PE module contains a method for configuration of assigned tasks with preceding inter-dependencies. This method is called after the construction of the PE. An algorithm scans the dependency database, obtained from the parser, and determines if outgoing inter-dependencies exists for any of the assigned tasks. If so, information about the target task(s) (e.g. base address of the target PE) is provided to the assigned task. The configuration ensures that an inter-processor communication will be initiated when task execution finishes.

7.10 Simulation data logging

The framework supports logging of different types of simulation data, ranging from a text based log file, containing real time state of the different PE's to a VCD timing file showing the different task states versus time. Depending on the SoC communication platform being OCP TL0 or TL1, different types of communication trace logging is possible as well. The different types of simulation data will be presented in section 10.1.3, page 65.

Chapter 8

The configuration file

This chapter gives a description of the configuration file as well as the syntax to use when creating a configuration file. The configuration file defines the boundaries of a simulation, with respect to task declaration, partitioning, RTOS configuration, PE address assignment etc. It is being read by the parser module, and if parsing is successful, the different information, from the file, can be obtained from the parser and used for configuration as desired. In conjunction to this, it must be emphasized that the *meaning* of the different arguments, for a particular declaration, is only applicable to implementation of this framework, due to the way it has been integration with the parser module[1].

Readers not interested in a more detailed description of the declaration syntax may read section 8.1 and then skip ahead to figure 8.8, page 51, showing an example of a configuration file.

8.1 Declaration types

The configuration file environment is very simple and based on a set of *declaration-types mnemonics*, identifying *what* to declare (e.g. dependency database or PE module behavior). After a declaration-type mnemonic follows by the *actual* declaration, which for some declaration-types may consists of a series declarations. The syntax to use for a declaration depends upon the declaration-type mnemonic. In general the syntax is very simple and easy to understand and use.

The different types of declarations to include in the configuration file, for this framework, is summarized in table 8.1 below. Some declarations may be left out, while others are mandatory.

[1]The meaning of the different arguments are not dictated by the parser, thus other implementations might use the configuration data, available from the parser, differently.

Declaration-type mnemonic	Description	Requirement
`module`	PE module behavior declaration (e.g. what kind of protocols to use in the RTOS)	Mandatory
`sub_task_map`	Task declaration (timing constrains etc.)	Mandatory
`ee_deadline`	End-to-end deadline declaration	Optional†
`dependency_map`	Dependency database declaration	Optional‡
`vcd_file`	VCD trace filename declaration (task state timing)	Optional
`log_file`	Monitor log file declaration (message monitoring)	Optional
`screen_dump`	Message monitoring screen dump enable/disable	Optional
† *Mandatory if an end-to-end task consists of multiple subtasks.*		
‡ *Mandatory if there are end-to-end tasks with dependencies.*		

Table 8.1: Declaration-type and requirement overview.

8.2 Declaration syntax

This section describes the declaration syntax used in the configuration file. Before presenting the syntax for the different types of declarations, a set of general rules are summarized.

- Declaration-type mnemonics are not case sensitive.
- The order of the different types of declarations (identified by the declaration-type mnemonic) may be arbitrary.
- The configuration file may include comments. A comment must always start with #. Everything afterward is treaded as a comment until reaching newline. Comments are in general allowed anywhere in the configuration file, also after a completed declaration.
- Space and tab are allowed anywhere in the configuration file, also in a declaration before and after a parameter separator (e.g. comma).
- Newline is also allowed anywhere in the configuration file, except in the middle of an incomplete declaration (e.g. before or after a parameter separator). For declaration-types supporting multiple declarations, each declaration must be separated by newline.

8.2.1 `module`

The module declaration-type mnemonic, `module` is used for defining the behavior of the different PE to instantiated. This includes address assignment as well as the types of protocols to use in the synchronizer, resource allocator and scheduler respectively. An unique `module` declaration is required for each PE, where the PE is identified by ID argument associated with the `"peID"` declaration name. In conjunction to this, all behavior declarations must be done within the boundary of the declaration region, identified by the closed brace, {....}. The declaration order may be arbitrary. Note that declaration names are case sensitive. Figure 8.1 shows the syntax and an example for subtask declaration.

Syntax and example

Module behaviour declaration

Syntax:

```
module {
  "peID"               = <peID>
  "address"            = <low>:<high>
  "synchronizer"       = <synchronizer_type>
  "resource_allocator" = <allocator_type>
  "scheduler"          = <scheduler_type>
  "monitor"            = <flag>
}
```

Example:

```
module {
  "peID"               = 1
  "address"            = 0x0000:0x0ffc
  "synchronizer"       = 0
  "resource_allocator" = 0
  "scheduler"          = 0
  "monitor"            = yes
}
```

Argument description	
`<peID>`	ID of the target PE for the behaviour description †
`<low>`	Lower address boundary. †
`<high>`	Upper address boundary. †
`<synchronizer_type>`	Synchronizer protocol identifier† 0 = DS
`<allocator_type>`	Resource allocator protocol identifier† 0 = Simplified basic PI
`<scheduler_type>`	Scheduler protocol identifier† 0 = RM 1 = EDF

`<flag>`	Message monitoring enable/disable flag for the PR, identified by <peID>.
	
	Disable = {`0|no|false`}
	Enable = {`1|yes|true`}
	
	Boolean mnemonic not case sensitive.

† Parameter expressed using decimal or hexadecimal (e.g. `0xf` or `0xF`) notation.

Figure 8.1: Syntax and example for `module` declaration.

8.2.2 `sub_task_map`

The subtask declaration-type mnemonic, `sub_task_map` must be declared before doing any task declaration. The mnemonic is only allowed to be declared once in the configuration file (multiple declaration would otherwise introduce ambiguity). Thus all task declarations must be done within the boundary of the declaration region, identified by the closed brace, {....}.

If a subtask has a resource requirement, it must be specified at the next line following the task declaration (comments in-between is allowed). Multiple resource requirements must also be separated by a newline.

The subtask ID of a subtask belonging to a particular task group is not specified. The reason is, that the ID will be assigned automatically by the parser *in the same order as the subtasks are declared*.

Figure 8.2 shows the syntax and an example for subtask declaration.

8.2.3 `ee_deadline`

The end-to-end deadline declaration-type mnemonic is `ee_deadline` and must be used when declaring the end-to-end deadline for end-to-end tasks with multiple subtasks. The mnemonic is only allowed to be used once in the configuration file (multiple declaration would otherwise introduce ambiguity). Thus all deadline declarations must be done within the boundary of the declaration region, identified by the closed brace, {....}. For end-to-end tasks only consisting of one subtask, the declaration will be ignored and the deadline, specified in the task declaration, will be used instead. If all assigned end-to-end tasks only consist of one subtask, the

Syntax and example

Subtask declaration

Syntax:

```
sub_task_map {
  “<name>”,<peID>,<parentID>,<p>,<BCET>,<WCET>,<dl>,<offs>,<comm>,<data>
    <resourceID>,<RRT>,<CSL>
               .
               .
  “<name>”,<peID>,<parentID>,<p>,<BCET>,<WCET>,<dl>,<offs>,<comm>,<data>
               .
               .
}
```

Example:

```
sub_task_map {
  “EndToEnd_Task1_1”,2,1,6000,476,698,100000,0,write   ,23
  “EndToEnd_Task1_2”,1,1,  * ,476,698,   *  ,*,read    ,10
  “EndToEnd_Task1_3”,2,*,  * , 93,117,   *  ,*,response,10
      1,10,15
      2,20,27
      3,50,60
  “EndToEnd_Task1_4”,1,*,  * ,108,216,   *  ,*,null    ,0
}
```

Argument description	
`<name>`	Task name. Letters, digits and underscore (_) is allowed. Space or tab is not allowed.
`<peID>`	ID of the target PE where to assign the task. †
`<parentID>`	ID of the parent (end-to-end) task, to which the subtask belongs to. †
`<p>`	Task period, specified in no. of clock cycles. †
`<BCET>`	Best-case execution time, specified in no. of clock cycles. †
`<WCET>`	Worse-case execution time, specified in no. of clock cycles. †
`<dl>`	Subtask deadline, specified in no.of clock cycles. Only applicable if the parent task consists of one subtask. †
`<offs>`	Release offset for the first instance of the subtask. Specified in no. of clock cycles and relative to zero time. †
`<comm>`	SoC transaction type identifier. The identifier is either a mnemonic or digit. Valid mnemonics are `null`,`write`,`read`,`response` (not case sensitive), while the corresponding digits are `0`,`1`,`2`,`3`.
`<data>`	Amount of data associated with a SoC transaction. Only applicable, if the subtask has interdependency. †
`<resourceID>`	ID of the resource to request. †
`<RRT>`	Resource request time, relative to the start of task execution. Specified in no.of clock cycles. †
`<CSL>`	Critical section length. Specified in no. of clock cycles. †
† Parameter expressed using decimal or hexadecimal (e.g. `0xf` or `0xF`) notation.	
* is a value-inheritance operator, causing the argument from the previous declared subtask to be inheritated. Cannot be used for resource declaration arguments.	

Figure 8.2: Syntax and example for `sub_task_map` declaration.

`ee_deadline` declaration may be left out. Figure 8.3 shows the syntax and an example for the end-to-end deadline declaration.

Syntax and example

End-to-end deadline declaration

Syntax:

```
ee_deadline {
  <parentID> = <ee_dl>
           .
           .
}
```

Example:

```
ee_deadline {
  1 = 20000
  2 = 33000
}
```

Argument description	
`<parentID>`	Parent ID of the end-to-end task to assign an end-to-end deadline. †
`<ee_dl>`	The end-to-end deadline, specified in no. of clock cycles. †
† Parameter expressed using decimal or hexadecimal (e.g. `0xf` or `0xF`) notation.	

Figure 8.3: Syntax and example for `ee_deadline` declaration.

8.2.4 `dependency_map`

The dependency database declaration-type mnemonic is `dependency_map`. The dependency database declaration is a symmetrical NxN matrix, with boolean entries. The matrix declaration must be done within the boundary of the declaration region, identified by the closed brace, {....}. The row and column index maps to the task group ID and a marked entry indicates a dependency between two end-to-end tasks. Dependencies between subtasks in an end-to-end task *are not to be specified*, since they will be assigned automatically by the parser. If no dependency exists between any of the declared end-to-end task, the dependency declaration may be left out. A description of the dependency database encoding is presented in section 5.1.1, page 24. Figure 8.4 shows the syntax and an example for the dependency database declaration.

Syntax and example	Argument description	

Dependency declaration
<u>Syntax:</u>
`relation_map {`
`  <boolean>,<boolean>, . . . ,<boolean>`
`  <boolean>,<boolean>, . . . ,<boolean>`
`.`
`.`
`}`
<u>Example:</u>
`relation_map {`
`  0,0,0,0,0`
`  0,1,0,0,0`
`  0,0,1,0,0`
`  0,0,0,1,0`
`}`
	`<boolean>`	A Boolean value, expressing if a dependency exists between two end-to-end tasks. Valid values are 0 and 1. A marked entry indicates a dependency.

Figure 8.4: Syntax and example for `dependency_map` declaration.

8.2.5 `log_file`

The monitoring log file declaration-type mnemonic is `log_file`. It must be declared, if the message monitoring is to be logged to a file. Argument to be provided is the filename. Only one log file declaration-type mnemonic is allowed. If the declaration is left out, no log file will be created. Figure 8.5 shows the syntax and an example for the log filename declaration.

Syntax and example	Argument description	

Log filename declaration
<u>Syntax:</u>
`log_file = “<filename>”`
<u>Example:</u>
`log_file = “MP3_Decoder.log”`
	`<filename>`	The filename of the UI monitor log file. Letters, digits and underscore (_) is allowed. Space or tab is not allowed.

Figure 8.5: Syntax and example for `log_file` declaration.

8.2.6 `vcd_file`

The VCD file declaration-type mnemonic is `vcd_file`. If the declaration is included in the configuration file, the states of the assigned tasks will be logged to the VCD file, during simulation. Argument to be provided is the filename. Only one VCD file declaration-type mnemonic is allowed. If the declaration is left out, no VCD file will be created. Figure 8.6 shows the syntax and an example for the VCD filename declaration.

Syntax and example	Argument description	

Log filename declaration
 	
Syntax:
`vcd_file = "<filename>"`
 	
Example:
`vcd_file = "MP3_Decoder"`
	`<filename>`	The filename of the VCD trace file. Letters, digits and underscore (_) is allowed. Space or tab is not allowed. Extension is added automatically by the framework. (`.vcd`)

Figure 8.6: Syntax and example for `vcd_file` declaration.

8.2.7 `screendump`

The screen dump declaration-type mnemonic, `screendump` is used for enabling or disabling the message monitoring dumping to the screen, during a simulation. If the declaration is left out, screen dumping will be enabled as default. For long simulations it is recommended to disable screen dumping, since it will increase performance significantly. In this case it is recommended to enable the log file option instead. Figure 8.7 shows the syntax and an example for the screen dump declaration.

Syntax and example	Argument description	

Screendump flag declaration
 	
Syntax:
 	
`screen_dump = <flag>`
 	
Syntax:
 	
`screen_dump = true`
	`<flag>`	UI on-screen monitor enable/disable Boolean flag.
		
		Disable = {`0|no|false`}
		Enable = {`1|yes|true`}
		
		Boolean mnemonic not case sensitive.

Figure 8.7: Syntax and example for `screendump` declaration.

```
screen_dump = no               # if this declaration is left out, default will be yes
log_file    = "MP3_logfile"    # if this declaration is left out, no log file will be created
vcd_file    = "MP3"            # if this declaration is left out, no vcd file will be created

module {
  "peID"               = 1
  "address"            = 0x0000:0x0ffc
  # --------------------------------------
  # module behavior configuration for PE#1
  # --------------------------------------
  "synchronizer"       = 0
  "resource_allocator" = 0
  "scheduler"          = 0     # 0=RM|1=EDF
  "monitor"            = yes
}

module {
  "peID"               = 2
  "address"            = 0x1000:0x1ffc
  # --------------------------------------
  # module behavior configuration for PE#2
  # --------------------------------------
  "synchronizer"       = 0
  "resource_allocator" = 0
  "scheduler"          = 0     # 0=RM|1=EDF
  "monitor"            = yes
}

ee_deadline {
  17 = 25000  # for end-to-end task, groupID 17
  18 = 25000  # for end-to-end task, groupID 18
}

sub_task_map {
# NOTE: * is an value-inheritance operator, causing the argument from the
#       previous declared task to be inherited.
# +-----------------------------------------------+
# |  MP3 Decoder -> Will be mapped to task graph#0 |
# +-----------------------------------------------
# <name>,<peID>,<groupID>,<per>,<BCET>,<WCET>,<dl>,<offset>,<transer_type>,<transfer_size>
  "MP3_Decoder_Task1" ,1, 1, 30000,   45,   45, 25000,0,write,100
  "MP3_Decoder_Task2" ,1, 2,   *  ,   19,   20,   *  ,0,null ,0
  "MP3_Decoder_Task3" ,2, 3,   *  ,   19,   20,   *  ,0,null ,0
  "MP3_Decoder_Task4" ,1, 4,   *  , 1471, 1545,   *  ,0,null ,0
  "MP3_Decoder_Task5" ,2, 5,   *  , 1471, 1545,   *  ,0,null ,0
  "MP3_Decoder_Task6" ,1, 6,   *  ,  567,  595,   *  ,0,null ,0
  "MP3_Decoder_Task7" ,2, 7,   *  ,  567,  595,   *  ,0,write,100
  "MP3_Decoder_Task8" ,1, 8,   *  , 2557, 2685,   *  ,0,write,100
  "MP3_Decoder_Task9" ,2, 9,   *  ,  103,  108,   *  ,0,null ,0
  "MP3_Decoder_Task10",1,10,   *  ,  103,  108,   *  ,0,null ,0
  "MP3_Decoder_Task11",2,11,   *  ,  852,  895,   *  ,0,null ,0
  "MP3_Decoder_Task12",1,12,   *  ,  852,  895,   *  ,0,null ,0
  "MP3_Decoder_Task13",2,13,   *  , 5797, 6087,   *  ,0,null ,0
  "MP3_Decoder_Task14",1,14,   *  , 5797, 6087,   *  ,0,null ,0
  "MP3_Decoder_Task15",2,15,   *  ,10667,11200,   *  ,0,null ,0
  "MP3_Decoder_Task16",1,16,   *  ,10667,11200,   *  ,0,null ,0
# +-------------------------------+
# |  end-to-end task, groupID 17  |
# +-------------------------------+
  "EndToEnd_Task17_1" ,1,17, 50000, 1471, 1545,100000,0,read    ,10
  "EndToEnd_Task17_2" ,2,17,   *  ,   71,  105,   *  ,0,response,10
  "EndToEnd_Task17_3" ,1,17,   *  , 1471, 1545,   *  ,0,null    ,0
# +-------------------------------+
# |  end-to-end task, groupID 18  |
# +-------------------------------+
  "EndToEnd_Task18_1" ,1,18, 60000,  476,  698,100000,0,write   ,37
  "EndToEnd_Task18_2" ,2,18,   *  , 1071, 1105,   *  ,0,read    ,29
  "EndToEnd_Task18_3" ,1,18,   *  ,  931, 1245,   *  ,0,response,29
        1,10,5     # resource request 1
        2,20,17    # resource request 2
        3,200,8    # resource request 3
        4,700,33   # resource request 4
  "EndToEnd_Task18_4" ,2,18,   *  ,  931, 1245,   *  ,0,null    ,0
}

relation_map {
# --------------------------------------------------
# 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 <-- this is the same as the groupID
# --------------------------------------------------
# MP3 Decoder dependencies
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 0
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 1
  0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 2
  0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 3
  0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 4
  0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 5
  0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 6
  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 7
  0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0  # 8
  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0  # 9
  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0  # 10
  0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0  # 11
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0  # 12
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0  # 13
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0  # 14
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0  # 15
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0  # 16
}
```

Figure 8.8: Example of a configuration file.

Chapter 9

SoC communication platform model

This chapter presents a SystemC based model for abstract SoC communication modeling, developed in conjunction to the extended abstract PE model presented in the previous chapter. The SoC communication platform model has the following features and characteristics:

- Topology modeling (currently available):
  1. 1D/2D mesh NoC based on packed switched traffic and minimal path routing, where links are granted on a first-come-first-served principle and routers are assumed to have infinite buffers and zero latency.
  2. Single shared bus granted on a first-come-first-served principle.
- Communication:
  1. Transmission of real data between nodes (e.g. RTL).
  2. Support for multi threaded out-of-order communication.
  3. Support for OCP 2.0 at TL0 and TL1.
- Implementation approach:
  1. Modular architecture which makes it easy to implement different SoC communication topologies as well as different communication protocols; also at different abstraction levels.
  2. Fully configurable node count.

Figure 9.1 shows the architecture of the SoC communication platform model. What makes this model especially interesting is the ability to support communication at low level (e.g. RTL) while still maintaining an abstract description of the underlying communication topology (e.g. mesh or bus). An advantage from this approach is speed improvement simulation wise (relative to low level NoC implementation). The drawback may be reduced accuracy, depending on the implementation detailedness in the SoC allocator.

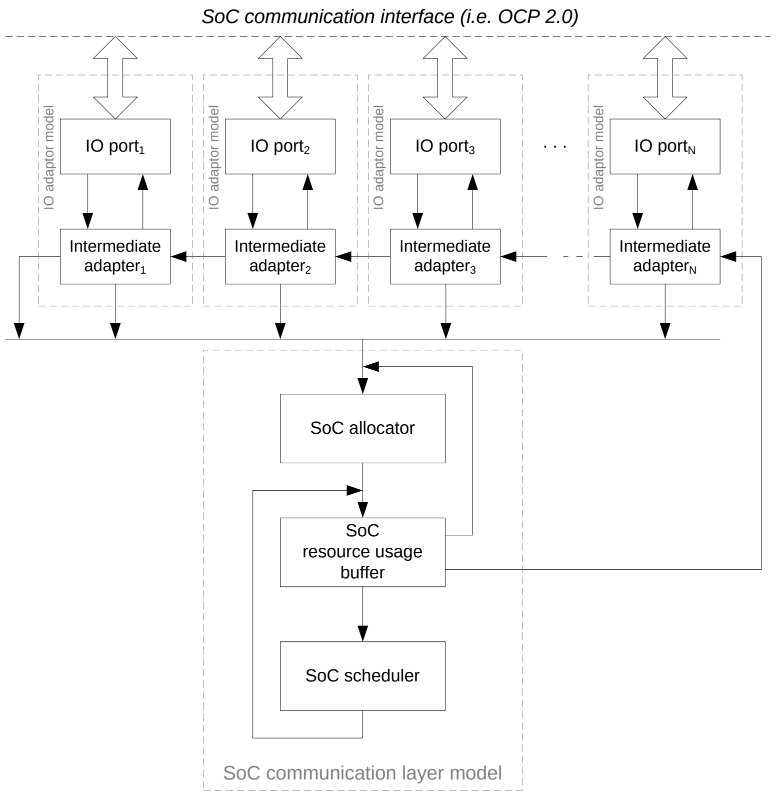


Figure 9.1: The SoC communication platform model architecture.

9.1 Module descriptions

From a top level point of view, the model consists of two main modules: a *IO adapter model* (composed of an IO port and an intermediate adapter) and a *communication layer model* - or SoC communication processor (composed of an allocator, resource usage buffer and scheduler). The characteristics of the different sub modules are briefly summarized below:

IO port Models the physical hardware port and implements and manages the SoC communication protocol.

Intermediate adapter Manages request/response messages to/from the IO port and serves as a message converter between the IO port and the SoC communication platform model. For an example, when a request is received from the IO port, a corresponding *transport message* (equivalent to a data package) will be created and issued to the NoC at the same time as the intermediate adapter starts fetching the actual data coming from the IO port.

SoC allocator Models the SoC communication topology (e.g. mesh or bus) and serves to minimize conflicts over shared communication resources (e.g. links and routers). It also manages the routing of the transport messages.

SoC resource usage buffer Models occupation of a shared communication resource (e.g. when a link is being used), by transport message buffering during the resource usage period.

SoC scheduler Models the scheduling of transport messages in case of resource contention. For an example, for a NoC topology it could be used to model the management of network service requirements such as guaranteed service (GS).

The motivation for having refined the IO adapter model into an IO port and intermediate adapter has been done for modular reason; that is to separate the communication handling to/from the SoC communication layer model from communication handling to/from the IP core. This also simplifies the implementation of the IO port model.

The exact behavior of the different modules and how they interact will described later in section 9.3, page 57.

9.2 Module communication

The communication between the modules are based on the `sc_link_mp` model available from the SystemC Master/Slave library. Data transferred through links are *pointers* to message objects, similar to the approach being used in the extended abstract PE model. See also section 11.1.1, page 87.

The communication link between the IO port and the intermediate adapter differs from the remaining communication links in the SoC communication model, with respect to the message type. This communication link follows the same protocol being used in the abstract PE model, between the IO task and IO device model. See also section 11.4.2, page 107. This also means that an IO device model used in the extended abstract PE model may be used as an IO port in the IO adapter model and vise versa.

The remaining links in the model serves to communicate pointers to transport messages. A transport message is generated by the intermediate adapter whenever a new inter-processor communication event starts; that is when a new response/request phase is being received by the IO port to which it connects. The transport message encapsulates information about the response/request and may float around in the SoC communication layer model, until it is ready to be released to the destination IO adapter. The transport message may also be considered as a single request/response package encapsulating all the data, since *only one* transport message will be generated for a particular inter-processor communication event (e.g. burst write).

A transport message is being defined by the struct, `noc_message_type` described in table 9.1. The struct consists of entries related to the inter-processor communications, used and maintained by the intermediate adapter. It also consists of entries used and maintained by the SoC communication layer model for routing management. The upper and lower part of table 9.1 shows the entries related to the inter-processor communication and routing management respectively.

Since a transport message encapsulates routing information as well as all the data related to the inter-processor communication, it might be considered equivalent to packed switched transmission. This however depends on the topology modeled by the SoC allocator, and how it manages a transport message.

Type	Name	Description
`unsigned int`	`type`	Identifying the transaction type. Valid mnemonics are: WR, RD, RESP.
`unsigned int`	`threadID`	The thread ID associated with the particular transaction.
`bool`	`singleReq`	Only applicable for a burst read request, and identifies, if the request type is single (true) or non-single (false).
`unsigned int`	`dataUnits`	Data transfer size. Equivalent to the burst length.
`deque<unsigned int>*`	`addrQ`	Pointer to a buffer containing the addresses associated with a request. The buffer is created and maintained by the intermediate adapter, initiating the transport message.
`deque<unsigned int>*`	`dataQ`	Pointer to a buffer containing data associated with a request/response. The buffer is created and maintained by the intermediate adapter, initiating the transport message.
`unsigned int`	`comm`	Action identifier. Valid mnemonics are READY, REFUSE, GRANT and RUN.
`unsigned int`	`from`	The *source* node ID (IO adapter) from where the transport messages originate from.
`unsigned int`	`to`	The *destination* node ID (IO adapter) where the message is hitting at.
`unsigned int`	`now`	A node ID identifying/modeling the current location of the transport message in the SoC communication layer.
`unsigned int`	`CSL`	Used as a critical section length watchdog timer in conjunction to usage of a shared communication resource (e.g. link).
`unsigned int` (1x10 array)	`resourceID`	Identifying the shared communication resource currently used/required by the transport message. The ID is an array, due to the implementation of the 1D/2D mesh allocator (described later).

Table 9.1: Transport message struct, `noc_message_type`. Entries in the upper part of the table relates to the inter-processor communication, while entries in the lower part is used by the SoC communication layer model for routing management.

9.3 Model behavior description

This section presents a behavior description of the SoC communication platform model, starting with the approach for communication task modeling. This is followed by a more in-depth behavior description of the different modules.

9.3.1 The communication task

From a high-level perspective, an inter-processor communication can be considered as a task. Assume two PE's, PE1 and PE2 are connected to the framework and that a simple application, consisting of two tasks, τ_1 and τ_2, where $\tau_1 \prec \tau_2$, have been partitioned in such way that $\tau_1 \mapsto$ PE1 and $\tau_2 \mapsto$ PE2. Thus in this simple scenario, inter-processor communication is required, due to the emergence of an inter-task dependency. The inter-processor communication can modeled as a communication task, τ_{NoC} in between τ_1 and τ_2. This is shown in figure 9.2.

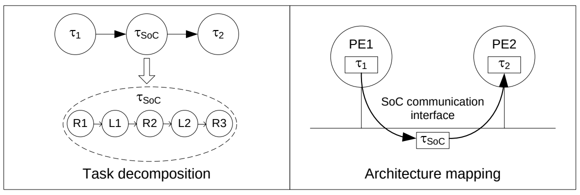


Figure 9.2: inter-processor communication modeling.

Depending on the topology, the communication task, τ_{SoC} can be further decomposed into a subset of tasks, connected in a chain. These tasks correspond to the usage of shared communication resources like links and routers, used when the data float around in the network layer. In this framework, the foundation for the modeling of the communication task, τ_{SoC} is based on the transport message issued by the intermediate adapter. This message will float around in the SoC communication layer model, until it is ready for being released to the destination IO adapter. The time interval between when an intermediate adapter issues a transport message and when the message is being released to the target IO adapter and the inter-processor communication has been done, corresponds to the communication task execution time. This time interval is dynamic and depends on the following basic factor:

- Data transfer size and bandwidth available.
- Topology and distance (link-wise) between source and destination IO adapter.
- Communication resource contention, if any.

The message floating is managed by the SoC allocator (since it models the topology) and simply models the different subtasks that a communication task may consists of (e.g. the usage of links and routers) In conjunction to the modeling of shared communication resource usage, a transport message get buffered in the SoC resource usage buffer, each time a new resource has been granted. The amount of time a message gets buffered, corresponds to a certain resource occupation time/critical section length (CSL), determined by the SoC allocator (relative to data transfer size and bandwidth).

The next sections explain how the communication modeling has been implemented, by giving a brief behavior description of the different modules. Implementation details are presented in section 12, page 139.

9.3.2 IO adapter model

The IO adapter model consists of the IO port and intermediate adapter model.

Receiving from the SoC interface

When an IO port receives a new request/response phase, it sends a `READY` message to the intermediate adapter. See table 11.6, page 111 for the types of messages coming from the IO port model. This causes the intermediate adapter to perform two operations:

1. Create buffer objects for address/data storage and initiates the fetching of address/data available from the IO port (in each clock cycle, new address/data is fetched pushed onto the buffers. This process continues until all the addresses/data, associated with the inter-processor communication event has been fetched).

2. Create a `READY` transport message to the SoC communication layer model, targeting the NoC allocator.

The transport message contains the pointers to the address/data buffer objects as well as inter-processor communication information likes transaction type, data transfer size and thread ID. The pointers will be used by the destination IO adapter when fetching the address/data associated with the inter-processor communication event. Besides the inter-processor communication information, the message also contains a *routing information*, identifying the source and destination node ID's as well as the current initial position of the message (which initially equals the source node ID). The routing information is store in the `from`, `to` and `now` fields in the message and used by the SoC allocator for the actual routing management. See also table 9.1, page 57. Determining the destination node ID depends on the inter-processor communication event being a request or a response to a previously initiated read request. If it is a request, the intermediate adapter finds the destination node ID from the address using a look-up table defining the address space mapping of the different IO adapters node ID (initialized before the simulation starts). For a response, the node ID is found from a look-up table holding the source ID of the IO adapter previously issued read request transport message. The look-up table is addressed using the thread ID. Thus using the thread ID associated with the response, the destination node ID is fetched from the table.

Transmitting to the SoC interface

When an intermediate adapter receives a `RUN` transport message from the SoC resource usage buffer, it means that the transport message has reached the destination IO adapter. This causes the intermediate adapter to issue a message to the IO port, starting the actual inter-processor communication afterward. The message contains the SoC communication information fetched from the transport message (address/data buffer pointers, transaction type, thread ID etc.).

If the transport message relates to a read request, the source ID of the IO adapter, initiating the transport message, is captured and stored in a look-up table for later use, when the response is coming. See also the description in the previous section, regarding the response handling procedure.

9.3.3 SoC allocator

The SoC allocator manages the usage of shared communication resources such as link. Transport messages received by the SoC allocator are always `READY` messages and can be considered as a communication resource release *and* request message at the same time.

When the NoC allocator receives a READY message it looks at the position information (`from`, `to` and `now`) and determine what the next routing action should be for the message and thus which resource to use. In conjunction to this, three possible position scenarios exists:

- `now == from`: The transport message has been released from the initial node position; that from the source IO adapter.
- `now != from AND now != to`: The message has reached a certain node in the SoC communication layer. Here the message has been released by the SoC resource usage buffer.
- `now == to`: The transport message has reached the destination IO adapter and the inter-processor communication has been processed as well (i.e. inter-processor communication completed). This message comes from NoC resource usage buffer.

If the message comes from the intermediate adapter (`now == from`) the SoC allocator performs the following operations:

- Selects the next shared communication resource to use and updates the `resourceID` entry in the message with the corresponding resource ID. The resource selection is done relative to the current (`now`) and destination (`to`) position and reflects the topology modeled by the NoC allocator
- Updates `now` to reflect the message position as it will be *after* the resource has been used. The is also topology dependent as well.

- Initialize the CSL watchdog timer, relative to the data transfer size, available from the transport message (`dataUnits`).
- Check if the requested resource is already occupied by evaluating an *reservation counter* for this particular resource. If already in use (the reservation counter is non-zero), the `READY` transport message is changed to a `REFUSE` and forwarded to the SoC scheduler, where it waits until the resource becomes available. If the resource is free, the `READY` transport message is changed to a `GRANT` and forwarded to the SoC resource usage buffer instead. In both scenarios the reservation counter, associated with the resource, will be incremented.

The processing of a transport message coming from the SoC resource usage buffer, when `now != from AND now != to` follows the same procedure as described above. However, since this message also relates to the release of a previously used resource, two operations are performed first:

1. Decrementing the reservation counter associated with the resource, identified by the entry, `resourceID` from the transport message.
2. Issuing a `RELEASE` message to the SoC scheduler, if the reservation counter is non-zero. If the counter is non-zero, it means that there is a transport message waiting in the SoC scheduler for this resource to become free. The `RELEASE` message contains the ID of the released resource and causes (in the current SoC scheduler implementation) the first pending transport messages, waiting for this resource, to be released to the SoC resource usage buffer.

If the message indicates completion of the inter-processor communication at (`now == to`), the allocator releases the associated resource and delete the message afterward.

9.3.4 SoC resource usage buffer

The SoC resource usage buffer models the actual resource usage mechanism, for an example when using a link. Whenever a transport message has been granted a shared resource by the SoC allocator, it gets forwarded to the SoC resource buffer, which buffers the message during the critical section length (CSL). At each clock cycle the `CSL` entry in the different buffered messages will be decremented, and whenever CSL reaches zero for a message (i.e. when the resource usage occupation time has been reached), it gets released and forwarded back to the SoC allocator again.

However, before buffering the message, it first checks if the next target node equals the destination IO adapter (`now == to`). If so, it means that the inter-processor communication also should start. Thus, it creates a copy of the transport message, changing it to a RUN message and forwards it to the destination

IO adapter. This ensures that the inter-processor communication starts when the resource has been granted.

9.3.5 SoC scheduler

The SoC scheduler manages the scheduling of transport messages in case of resource contention. The current scheduler implementation does not support features such as guaranteed service (GS). Thus pending transport messages, waiting for a shared resource to become free, will be served in a first-come-first-served manner.

Whenever the SoC allocator refuses a transport message to use a shared resource, it gets forwarded to the SoC scheduler and buffered, until the resource becomes available. Releasing a transport message happens when a `RELEASE` message is received from the SoC allocator, indicating the release of a shared resource. The scheduler searches through the pending transport messages, until the first message waiting for this resource has been found. The transport message is removed from the buffer, changed to a `GRANT` message and forwarded to the SoC resource usage buffer.

Chapter 10

Design space exploration experiments

In this chapter, the capabilities of the extended abstract PE model and SoC communication platform model will be demonstrated, through some design space exploration examples. The examples integrate the abstract PE and SoC communication models, forming different simulation frameworks for distributed MPSoC architecture and application modeling. The examples demonstrate how the models easily can be used for architecture optimization relative to the application running on top of it.

Three examples will be presented in the mentioned order:

1. A basic introduction using a simple simulation framework for presenting the output data available from a simulation.

2. SoC communication topology exploration and mixture of abstraction level in the SoC communication interface.

3. Complex system performance behavior analysis.

For all examples, the following applies as well:

- Task ID encoding, used in conjunction to address generation, has been selected such that bit[3:0] contains the subtask ID. See also section 6.4, page 35 for more information.

- Fixed OCP2.0 (TL1 and TL0) channel configuration for all example. See also TL0 and TL1 configuration files in appendix C, page 167.

- Fictive selected data sizes associated task graph edges.

10.1 Example 1: Introduction

This example serves as an introduction to the frameworks and presents the different results available from a simulation. It is based on modeling a simple architecture, consisting of two extended abstract PE's communicating through a 32-bit OCP2.0 TL0 compliant bus. Application running on top is based on the MP3 Decoder task graph available from [23].

The source code and configuration file for this example can be found on the enclosed CD-ROM in the directory: `/ARTS_Model/builds/example1`.

10.1.1 The simulation framework

Figure 10.1 shows a block diagram of the simulation framework. The bus model is based on the SoC communication platform model using the SoC allocator model for a single shared bus. Both RTOS uses RM scheduling policy. The modeled clock period will be $1ns$.

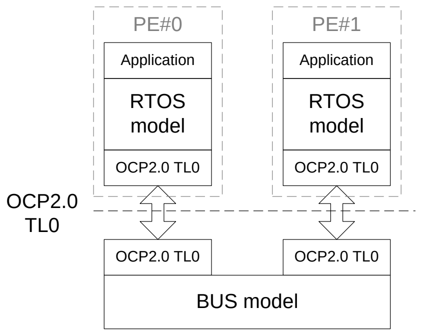


Figure 10.1: The system level model.

10.1.2 Application model

Assuming the BCET and WCET figures for the MP3 decoder application applies to the PE's to model, it can be found that the application is not able to execute successfully on a single PE, since the BCET is $41554ns$ while the deadline $25000ns$. See also the configuration file, `example1.task` or figure 10.2. Thus parallel tasks must execute on different processors in order to meet the deadline requirement. For this example the partition showed in figure 10.2 will be used. As it can be seen, the selected partitioning introduces inter-task dependency between three tasks. All inter-processor communication related inter-task dependency will be modeled as write transaction of 10x32-bit data words.

Task ID	BCET [ns]	WCET [nS]	Data [32bit]	dl [ns]	T [ns]
1,1	45	45	10	25000	30000
2,1	19	20	10	25000	30000
3,1	19	20	10	25000	30000
4,1	1471	1545	10	25000	30000
5,1	1471	1545	10	25000	30000
6,1	567	595	10	25000	30000
7,1	567	595	10	25000	30000
8,1	2557	2685	10	25000	30000
9,1	103	108	10	25000	30000
10,1	103	108	10	25000	30000
11,1	852	895	10	25000	30000
12,1	852	895	10	25000	30000
13,1	5797	6087	10	25000	30000
14,1	5797	6087	10	25000	30000
15,1	10667	11200	10	25000	30000
16,1	10667	11200	10	25000	30000

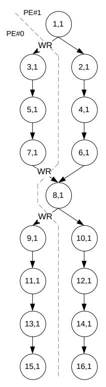


Figure 10.2: MP3 Decoder task graph (partitioned).

10.1.3 Simulation output data

When running the executable system level model, `example1.x` for 30000ns and using the configuration file for this example, `example1.task`, the following files will be generated:

- **Text based log file** (`ex1_logfile`), containing the real time state of the system level model, presented in a readable text format. From this file it is possible to see:
  - Missed subtask and end-to-end deadlines (if any).
  - Task graph execution completion time.
  - RTOS states of the different PE's versus time (e.g. task execution, preemption etc.)
  - Inter-processor communication event information (e.g. address information, burst length, thread ID etc.).
  - PE utilization figures, including IO task usage.
  - Address space map for tasks having succeeding inter-task dependencies.
- **PE task scheduling VCD timing file** (`ex1_PE.vcd`), contains the task execution in the different PE's versus time. From this file it is possible to see which task is executing on a certain PE at a certain time.
- **OCP2.0 TL0 VCD timing files** (`ex1_RTL0_PE#0` and `ex1_RTL0_PE#1`), containing the TL0/RTL trace in the OCP interface between the bus and the PE0 and PE1 respectively.
- **Task state VCD timing file** (`ex1_task.vcd`), containing the states of the assigned tasks (idle, running or preempted) versus time.

10.1.4 Analyzing the log file

The text based log file may be useful for gaining information about the state of the system level model at a particular time. The logging is done in a sequential time-wise manner. Figure 10.3 shows a section of the log file for this example. For convenience, the complete log file has been included in appendix D, page 169.

```
45 ns     PE#0: task(1,1) (request to NoC: task(3,1),addr=0x1011,dataUnits=10)-> adaptor
45 ns     PE#0: scheduler (start NoC write request) -> task(IO)

45 ns     |OCP| PE0_TL0.IOdevice.master: sent BURST request.
          | M | Data handshake: yes
          | A | MCmd: WR, MAddr: 0x1011, MThreadID: 0x1
          | S | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


46 ns     |OCP| soc_comm.tl0_io_a.slave: receiving BURST request.
          | S | Data handshake: yes
          | L | MCmd: WR, MAddr: 0x1011, MThreadID: 0x1
          | A | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


46 ns     |OCP| soc_comm.tl0_io_b.master: sent BURST request.
          | M | Data handshake: yes
          | A | MCmd: WR, MAddr: 0x1011, MThreadID: 0x1
          | S | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


47 ns     |OCP| PE1_TL0.IOdevice.slave: receiving BURST request.
          | S | Data handshake: yes
          | L | MCmd: WR, MAddr: 0x1011, MThreadID: 0x1
          | A | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1

47 ns     PE#1: task(IO) (write data ready) -> scheduler
47 ns           synchronizer: READY from IO task
47 ns     PE#1: scheduler (fetch data from SLAVE) -> task(IO)

55 ns     |OCP| PE0_TL0.IOdevice.master: Request completed

55 ns     PE#0: task(IO) (IO task finished) -> scheduler
55 ns           synconizer: releases task(2,1)
55 ns     PE#0: scheduler (run) -> task(2,1)

56 ns     |OCP| soc_comm.tl0_io_b.master: Request completed
57 ns     PE#1: task(1,1) (external task finished) -> scheduler
57 ns           synconizer: releases task(3,1)

57 ns     PE#1: task(IO) (IO task finished) -> scheduler
57 ns     PE#1: scheduler (run) -> task(3,1)
```

Figure 10.3: A section of the log file.

From the log file section it can be seen, that when task(1,1) in PE0 finishes execution at 45ns, the inter-processor communication consisting of a burst write starts afterward. When the request is received in PE1, the IO task is launched at 47ns and data fetching starts. After 55ns, the IO task in PE0 has completed the burst write, and task(2,1) starts executing. At 57ns, when all data associated with the burst write have been received, the IO task completes and task(3,1), depending on the data, starts to execute.

For small and simple simulations like this one, it may be easy to get an overview of the system level performance from the log file alone, since only two PE's and a single task graph is considered. However, for more complex simulation this becomes almost impossible (or at least very difficult), due to the very high level of detailedness in the log file. In these situations, the VCD files containing the task scheduling/state information should be used instead, while the log file serves as a reference.

Simulation summary

The end of the log file contains a small simulation summary report. This includes the address map for tasks having interdependencies as well as PE utilization and IO task usage figures. Figure 10.4

```
TASK ADDRESS MAP
   PE#0 :
   Task(  1, 1) :
   Task(  2, 1) :
   Task(  4, 1) :
   Task(  6, 1) :
   Task(  8, 1) : 0x71
   Task( 10, 1) :
   Task( 12, 1) :
   Task( 14, 1) :
   Task( 16, 1) :

   PE#1 :
   Task(  3, 1) : 0x1011
   Task(  5, 1) :
   Task(  7, 1) :
   Task(  9, 1) : 0x1081
   Task( 11, 1) :
   Task( 13, 1) :
   Task( 15, 1) :


PE UTILIZATION
   PE#0 : 75.7967% (22739@30000)
   PE#1 : 67.0833% (20125@30000)

IO TASK USAGE
   PE#0 : 0.131932% (30@22739)
   Write data TX    : 0.0879546% (20@22739)
   Write data RX    : 0.0439773% (10@22739)

   PE#1 : 0.149068% (30@20125)
   Write data TX    : 0.0496894% (10@20125)
   Write data RX    : 0.0993789% (20@20125)
```

Figure 10.4: Log file summary.

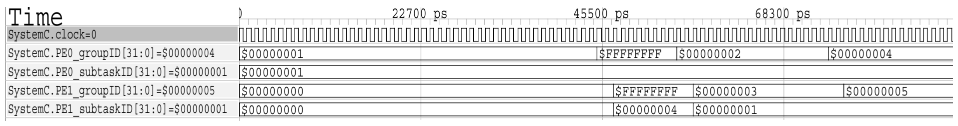
The address map shows the automatically assigned addresses the tasks mapped to a particular PE. If no address has been assigned to a task, it means that it does not have any succeeding inter-task dependency (i.e. does not depends on data from a non-local task for being able to execute). In this example, the address map indicates that task(3,1), task(8,1) and task(9,1) has succeeding inter dependencies, which is also expected due to the selected task partitioning shown in figure 10.2. See also section 6.4, page 35 for more information regarding address assignment.

The PE utilization figures indicate the amount of time a particular PE has been used for task execution, relative to the total simulation time. The IO task usage figures indicate the amount of time spend on inter-processor communication handling. These figures are relative to the total time used by the PE for task execution. Thus in this example, the utilization of PE0 and PE1 is 75.7% and 67.1%, where 0.13% and 0.15% of this time has been spend on SoC communication handling respectively.

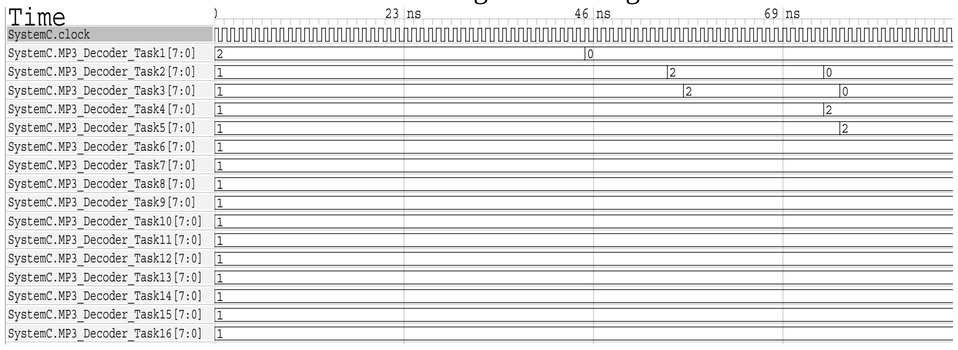
The current summary report does not include information about task graph execution times and no.of missed deadlines. But for this example there were no missed deadlines and the task graph completed execution after 22775ns (also obtained from the text based log file).

10.1.5 Analyzing the task scheduling and state

From the *PE task scheduling* and *task state VCD timing files* it is possible to get quick overview of the tasks executing on the different PE's as well as the state of the tasks. In figure 10.5, a section of the VCD plots for this example is shown.


Time
22700 ps
45500 ps
68300 ps
SystemC.clock=0
SystemC.PE0_groupID[31:0]=$00000004
SystemC.PE0_subtaskID[31:0]=$00000001
SystemC.PE1_groupID[31:0]=$00000005
SystemC.PE1_subtaskID[31:0]=$00000001
$00000001
$FFFFFFFF
$00000002
$00000004
$00000000
$00000003
$00000005
$00000001

PE task scheduling VCD timing file.


Time
23 ns
46 ns
69 ns
SystemC.clock
SystemC.MP3_Decoder_Task1[7:0]
SystemC.MP3_Decoder_Task2[7:0]
SystemC.MP3_Decoder_Task3[7:0]
SystemC.MP3_Decoder_Task4[7:0]
SystemC.MP3_Decoder_Task5[7:0]
SystemC.MP3_Decoder_Task6[7:0]
SystemC.MP3_Decoder_Task7[7:0]
SystemC.MP3_Decoder_Task8[7:0]
SystemC.MP3_Decoder_Task9[7:0]
SystemC.MP3_Decoder_Task10[7:0]
SystemC.MP3_Decoder_Task11[7:0]
SystemC.MP3_Decoder_Task12[7:0]
SystemC.MP3_Decoder_Task13[7:0]
SystemC.MP3_Decoder_Task14[7:0]
SystemC.MP3_Decoder_Task15[7:0]
SystemC.MP3_Decoder_Task16[7:0]

Task state VCD timing file,

Figure 10.5: VCD plot sections.

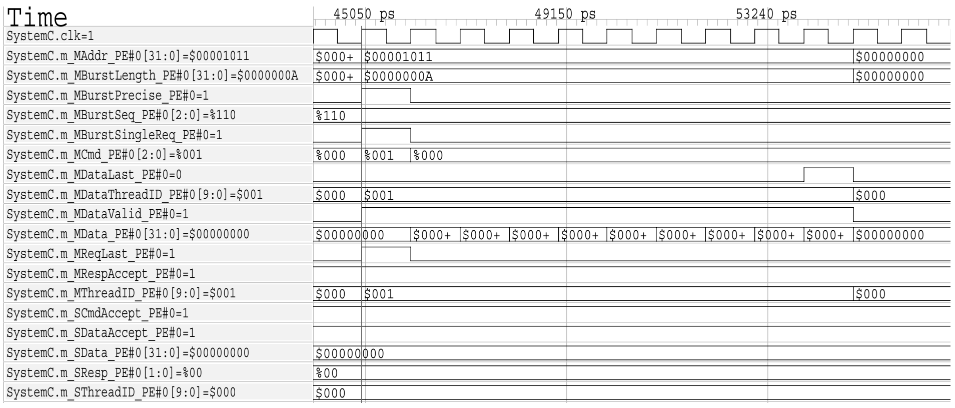
In the VCD plot associated with the PE task scheduling VCD timing file, each PE is identified by two traces showing the group ID (e.g. `PE0_groupID`) and the subtask ID (e.g. `PE0_subtaskID`) of the task currently running. The IO task execution is identified by the group ID being `0xFFFFFFFFh`, while the subtask ID here identifies the action performed by the IO task (i.e. doing a write or read transaction, fetching response data etc.). When the group ID and subtask ID is zero, it means that the PE is in idle. For example, task(1,1) is executing on PE1 in the time between 0 to 45ns, since the group ID and subtask ID both equal 1. Afterward starts the IO task execution, since the group equals `0xFFFFFFFFh`. The action performed by the IO task is a write, since the subtask ID equals 1. See also `Parameters.h` identifying the mnemonic values for the IO task actions.

In the VCD plot associated with the task state VCD timing file, each task is identified by a trace showing the state. The possible states are: 0 = idle, 1=ready, 2=run and 3=preempted. For example, task(1,1) is executing in the time between 0 to 45ns, since the state 1. Afterward the task goes into idle, since the state is 0.

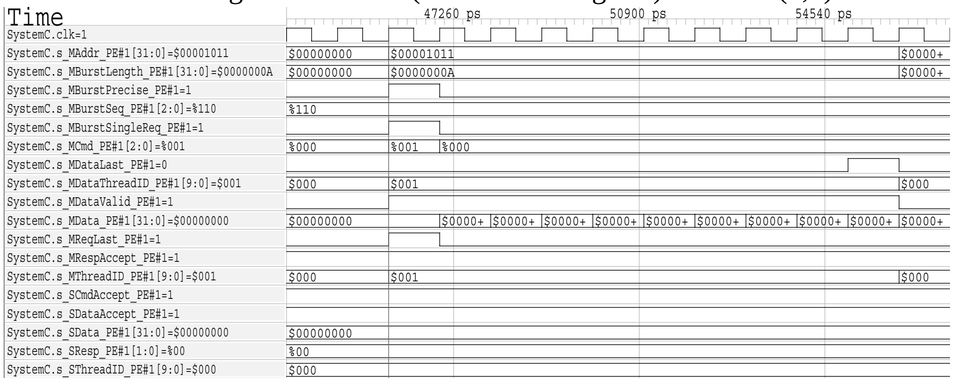
Analyzing SoC communication

From the *text based log file* it can be found that inter-processor communication is initiated at 45ns, 2140ns and 4721ns corresponding to when task(1,1) in PE0, task(7,1) and PE1 and task(8,1) in PE0 finishes execution respectively. This is also expected, since the tasks have preceding inter-task dependencies. See also figure 10.2.

The corresponding OCP2.0 communication traces can be observed from the *OCP2.0 TL0 VCD timing files*. These are shown in figure 10.6, to 10.8. For each SoC communication event, the trace is shown for the producer and consumer processor respectively.

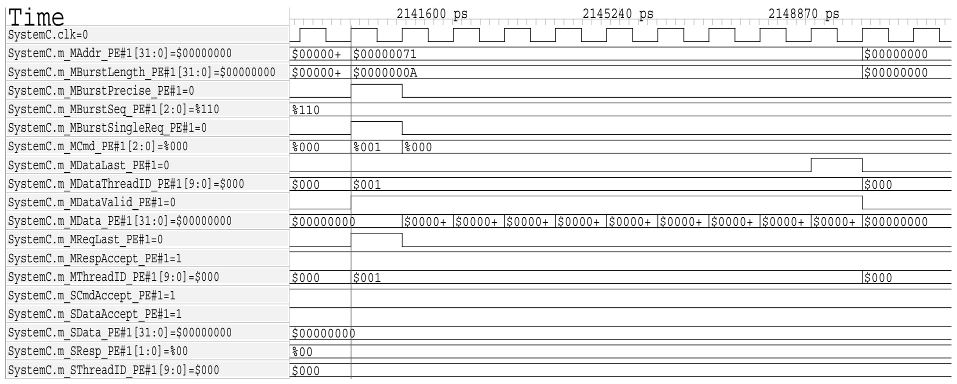


45ns: PE0 Starting burst write at (OCP Master signals) after task(1,1) finishes.

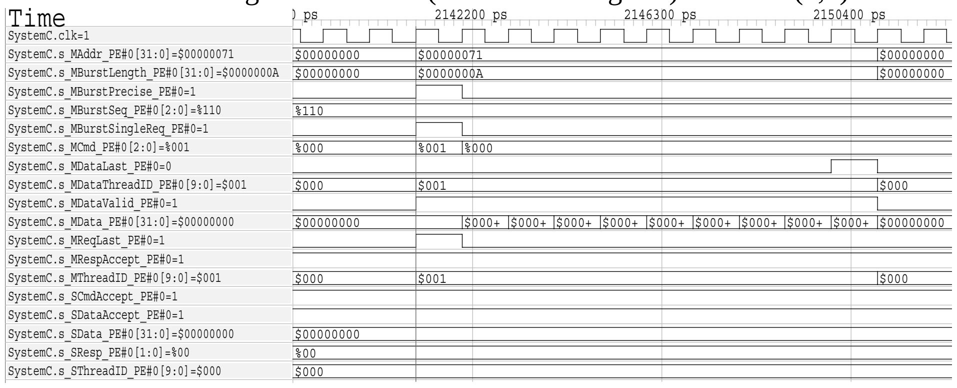


46ns: PE1 Receives the burst write data from task(1,1)

Figure 10.6: Inter-task dependency, task(1,1)→task(2,1)


Time
2141600 ps
2145240 ps
2148870 ps
SystemC.clk=0
SystemC.m_MAddr_PE#1[31:0]=$00000000
SystemC.m_MBurstLength_PE#1[31:0]=$00000000
SystemC.m_MBurstPrecise_PE#1=0
SystemC.m_MBurstSeq_PE#1[2:0]=%110
SystemC.m_MBurstSingleReq_PE#1=0
SystemC.m_MCmd_PE#1[2:0]=%000
SystemC.m_MDataLast_PE#1=0
SystemC.m_MDataThreadID_PE#1[9:0]=$000
SystemC.m_MDataValid_PE#1=0
SystemC.m_MData_PE#1[31:0]=$00000000
SystemC.m_MReqLast_PE#1=0
SystemC.m_MRespAccept_PE#1=1
SystemC.m_MThreadID_PE#1[9:0]=$000
SystemC.m_SCmdAccept_PE#1=1
SystemC.m_SDataAccept_PE#1=1
SystemC.m_SData_PE#1[31:0]=$00000000
SystemC.m_SResp_PE#1[1:0]=%00
SystemC.m_SThreadID_PE#1[9:0]=$000
$00000+ $00000071 $00000000
$00000+ $0000000A $00000000
%110
%000 %001 %000
$000 $001 $000
$00000000 $0000+ $0000+ $0000+ $0000+ $0000+ $0000+ $0000+ $0000+ $0000+ $00000000
$000 $001 $000
$00000000
%00
$000

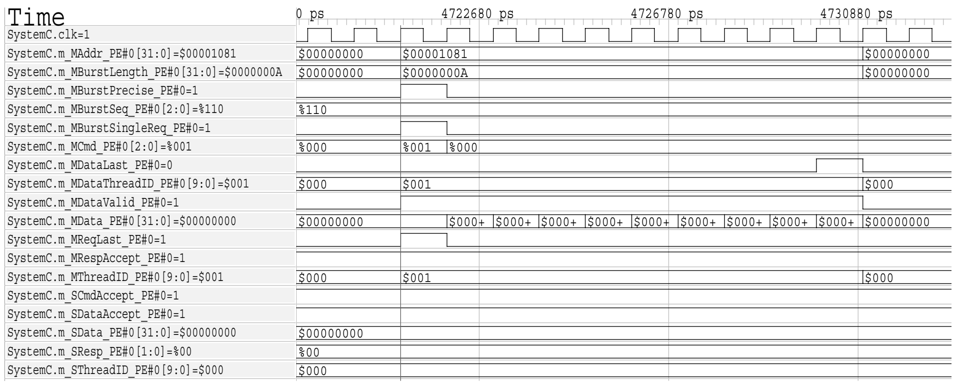
2140ns: PE1 Starting burst write at (OCP Master signals) after task(1,1) finishes.


Time
) ps
2142200 ps
2146300 ps
2150400 ps
SystemC.clk=1
SystemC.s_MAddr_PE#0[31:0]=$00000071
SystemC.s_MBurstLength_PE#0[31:0]=$0000000A
SystemC.s_MBurstPrecise_PE#0=1
SystemC.s_MBurstSeq_PE#0[2:0]=%110
SystemC.s_MBurstSingleReq_PE#0=1
SystemC.s_MCmd_PE#0[2:0]=%001
SystemC.s_MDataLast_PE#0=0
SystemC.s_MDataThreadID_PE#0[9:0]=$001
SystemC.s_MDataValid_PE#0=1
SystemC.s_MData_PE#0[31:0]=$00000000
SystemC.s_MReqLast_PE#0=1
SystemC.s_MRespAccept_PE#0=1
SystemC.s_MThreadID_PE#0[9:0]=$001
SystemC.s_SCmdAccept_PE#0=1
SystemC.s_SDataAccept_PE#0=1
SystemC.s_SData_PE#0[31:0]=$00000000
SystemC.s_SResp_PE#0[1:0]=%00
SystemC.s_SThreadID_PE#0[9:0]=$000
$00000000 $00000071 $00000000
$00000000 $0000000A $00000000
%110
%000 %001 %000
$000 $001 $000
$00000000 $000+ $000+ $000+ $000+ $000+ $000+ $000+ $000+ $000+ $00000000
$000 $001 $000
$00000000
%00
$000

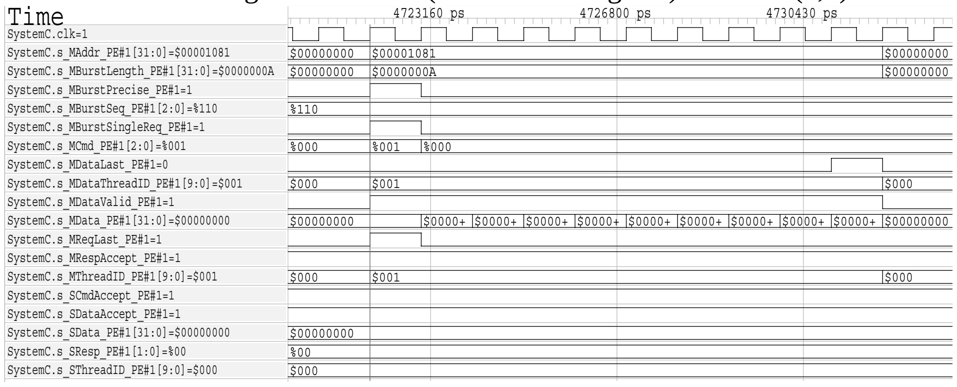
2141ns: PE0 Receives the burst write data from task(7,1)

Figure 10.7: Inter-task dependency, task(7,1)→task(8,1)

As it can be seen, all transactions are burst writes with a burst length equal to 10x32-bit data words. Further, the communication is based on single request burst writes with data-handshake, which is due to the selected channel configuration. Notice also the address fields encoding which reflects from which task the data are transmitted. In fact, it is possible to see the group and subtask ID directly, due to the selected task ID encoding, where the subtask ID is located in bit[3:0]. For an example, in figure 10.8 the address (`MAddr`) is `0x1081h`, indicating that it is data coming from task(8,1). `0x1000h` equals the base address for PE1. See also the configuration file.



4721ns: PE0 Starting burst write at (OCP Master signals) after task(8,1) finishes.



4722ns: PE1 Receives the burst write data from task(8,1)

Figure 10.8: Inter-task dependency, task(8,1)→task(9,1)

10.2 Example 2: SoC communication topology exploration

This example serves to demonstrate how the models can be used in a simulation framework for exploring different SoC communication topologies, relative to an initial design space consisting two applications and four abstract PE's. The example will also demonstrates the possibility to do abstraction level mixture in the SoC communication interface, by using a combination of OCP2.0 TL0 and TL1. Focus for the simulations will be on results showing:

- Task graph execution time and missed deadlines.
- SoC communication platform real-time state (link contention etc).
- SoC communication traces for inter-processor communication through a mixed abstraction level SoC communication interface (TL0-TL1).

In conjunction to monitoring the real-time state of the SoC communication model, it has been extended to print out relevant information for the allocator and scheduler to the screen.

The source code and configuration file for this example can be found on the enclosed CD-ROM in the directory: `/ARTS_Model/builds/example2`.

10.2.1 The simulation framework

The simulation framework is shown in figure 10.9. It consists of four PE's, where PE0 and PE2 interfaces to the SoC communication platform using OCP2.0 TL1. PE1 and PE3 interfaces to the SoC communication platform using OCP2.0 TL0. The SoC communication platforms to explore consist of a bus, 1D and 2D mesh. The example will also show how the choice of topology may affect the task scheduling as well. In all simulation scenario's the scheduling policy will be based on RM. Further, the modeled clock period will be $1ns$.

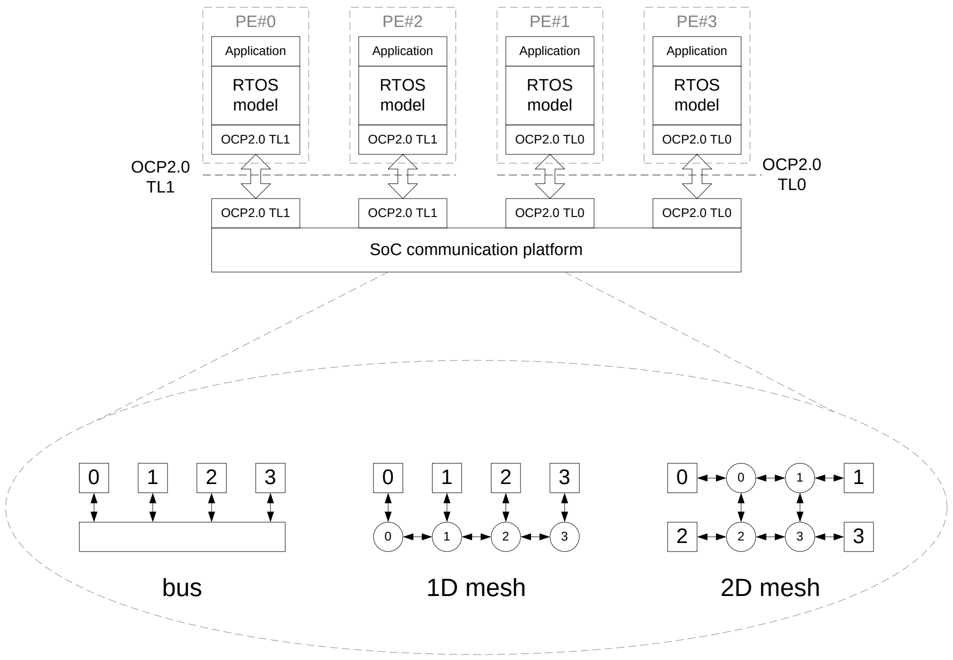


Figure 10.9: The system level model.

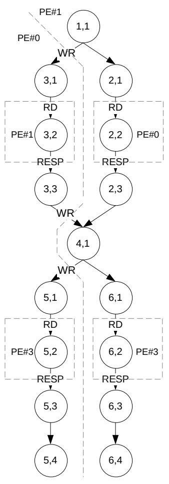
10.2.2 Application modeling

Two application models are being used in this example: The MP3 Decoder [23] also used in the previous example and a *fictive* end-to-end task consisting of four subtasks. In this example, the MP3 decoder task graph has been grouped into six end-to-end tasks to support read transactions. The grouping does not alter the task graph characteristics but is just another level of abstraction. The partitioning of the MP3 decoder is similar to the one being used in example 1, except that task(2,2) and task(3,2)[1] are mapped to PE0 and PE1 respectively and associated response data to the succeeding read request initiated by task(2,1) and task(3,1)[2] respectively. Further, both task(5,2) and task(6,2)[3] has been mapped to PE3 and also associated with a response data. For all cases of inter-task dependency, the data transfer size has been chosen to 10x32-bit data words.

The fictive application, modeled as an end-to-end task with four subtasks, runs independently of the MP3 decoder. It is defined by group ID 7 and with task(7,1) and task(7,3) mapped onto PE2 and task(7,2) and task(7,4) mapped onto PE3. The end-to-end task will initiate a read from PE3 followed by a write to PE3 when the response has been received. Characteristic for this end-to-end task is the large data transfer sizes associated with inter-task dependencies as well as having relative short period, compared to the MP3 decoder task graph (6 time smaller). The write requests and response consists of 200x32-bit and 5000x32-bit data words respectively. The data transfer sizes, relative to the period, is quite unrealistic but will later shows as a good example for how large data transfer may affect the simulated system performance relative to the selected SoC communication topology.

Figure 10.10 shows the two task graphs and their timing figures. Also note that BCET = WCET. This has been chosen to remove the random timing jitter to make it easier to compare the system performance, when using different SoC communication topologies.

Task ID	ET [ns]	Data [32bit]	dl [ns]	T [ns]
1,1	45	10	25000	30000
2,1	20	10	25000	30000
2,2	1545	10	25000	30000
2,3	595	10	25000	30000
3,1	20	10	25000	30000
3,2	1545	10	25000	30000
3,3	595	10	25000	30000
4,1	2685	10	25000	30000
5,1	108	10	25000	30000
5,2	895	10	25000	30000
5,3	6087	10	25000	30000
5,4	11200	10	25000	30000
6,1	108	10	25000	30000
6,2	895	10	25000	30000
6,3	6087	10	25000	30000
6,4	11200	10	25000	30000



Task ID	ET [ns]	Data [32bit]	dl [ns]	T [ns]
7,1	10	5000	15000	10000
7,2	500	5000	15000	10000
7,3	500	50	15000	10000
7,4	10	0	15000	10000

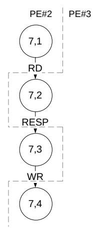


Figure 10.10: MP3 Decoder task graph and fictive end-to-end task (partitioned).

[1]equal to task(4,1) and task (5,1) in example 1

[2]equal to task(2,1) and task (3,1) in example 1

[3]equal to task(11,1) and task(12,1) in example 1

10.2.3 Bus topology simulation result

First simulation performed uses bus topology. This is done by setting the `module` declaration parameter, `"soc_allocator"` equal to zero in the configuration file, `example2.task`.

When running the simulation for 35000ns, the simulation result showed in figure 10.11 is obtained. Most of the data relate to the state of the SoC communication model, but it is also possible to see when task graph execution completes and missed subtask deadlines/end-to-end deadlines. The data related to the SoC communication consists of a time stamp, an action state, a link contention counter, transaction type, address or response data and finally the routing information (`posID`), identifying the source and destination node ID.

```
   11 ns  GRANT           RD    0x00003071   posID=(2,3)   # RD@Task(7,1)
   13 ns  BUS RELEASED    RD                 posID=(2,3)
   46 ns  GRANT           WR    0x00000011   posID=(1,0)   # WR@Task(1,1)
   57 ns  BUS RELEASED    WR                 posID=(1,0)
   76 ns  GRANT           RD    0x00000021   posID=(1,0)   # RD@Task(2,1)
   78 ns  BUS RELEASED    RD                 posID=(1,0)
   79 ns  GRANT           RD    0x00001031   posID=(0,1)   # RD@Task(2,1)
   81 ns  BUS RELEASED    RD                 posID=(0,1)
  513 ns  GRANT           RESP  0x00000072   posID=(3,2)   # Response@Task(7,2)
 1625 ns  ***REFUSE*** 1  RESP  0x00000022   posID=(0,1)   # Response@Task(2,2) REFUSED
 1626 ns  ***REFUSE*** 2  RESP  0x00000032   posID=(1,0)   # Response@Task(3,2) REFUSED
 5514 ns  BUS RELEASED    RESP               posID=(3,2)
 5514 ns  SCHD RELEASE    RESP  0x00000022   posID=(0,1,1) # Response@Task(2,2)
 5525 ns  BUS RELEASED    RESP               posID=(0,1)
 5525 ns  SCHD RELEASE    RESP  0x00000032   posID=(1,0,0) # Response@Task(3,2)
 5536 ns  BUS RELEASED    RESP               posID=(1,0)
 6016 ns  GRANT           WR    0x00003073   posID=(2,3)   # WR@Task(7,3)
 6067 ns  BUS RELEASED    WR                 posID=(2,3)

 6077 ns     Task graph 1 completed. Now preparing for a new cycle:
             Restoring relation matrix
 6132 ns  GRANT           WR    0x00001033   posID=(0,1)   # WR@Task(3,3)
 6143 ns  BUS RELEASED    WR                 posID=(0,1)
 8829 ns  GRANT           WR    0x00000041   posID=(1,0)   # WR@Task(4,1)
 8840 ns  BUS RELEASED    WR                 posID=(1,0)
 8947 ns  GRANT           RD    0x00003061   posID=(1,3)   # RD@Task(6,1)
 8949 ns  BUS RELEASED    RD                 posID=(1,3)
 8950 ns  GRANT           RD    0x00003051   posID=(0,3)   # RD@Task(5,1)
 8952 ns  BUS RELEASED    RD                 posID=(0,3)
 9844 ns  GRANT           RESP  0x00000062   posID=(3,1)   # Response@Task(6,2)
 9855 ns  BUS RELEASED    RESP               posID=(3,1)
10011 ns  GRANT           RD    0x00003071   posID=(2,3)   # RD@Task(7,1)
10013 ns  BUS RELEASED    RD                 posID=(2,3)
10513 ns  GRANT           RESP  0x00000072   posID=(3,2)   # Response@Task(7,2)
15514 ns  BUS RELEASED    RESP               posID=(3,2)
16016 ns  GRANT           WR    0x00003073   posID=(2,3)   # WR@Task(7,3)
16067 ns  BUS RELEASED    WR                 posID=(2,3)

16077 ns     Task graph 1 completed. Now preparing for a new cycle:
             Restoring relation matrix
16309 ns  GRANT           RESP  0x00000052   posID=(3,0)   # Response@Task(5,2)
16320 ns  BUS RELEASED    RESP               posID=(3,0)
20011 ns  GRANT           RD    0x00003071   posID=(2,3)   # RD@Task(7,1)
20013 ns  BUS RELEASED    RD                 posID=(2,3)
20513 ns  GRANT           RESP  0x00000072   posID=(3,2)   # Response@Task(7,2)
   25 us (!) Task(5,4) has missed its deadline
   25 us (!) Task(6,4) has missed its deadline
25514 ns  BUS RELEASED    RESP               posID=(3,2)
26016 ns  GRANT           WR    0x00003073   posID=(2,3)   # WR@Task(7,3)
26067 ns  BUS RELEASED    WR                 posID=(2,3)

26077 ns     Task graph 1 completed. Now preparing for a new cycle:
             Restoring relation matrix
30011 ns  GRANT           RD    0x00003071   posID=(2,3)   # RD@Task(7,1)
30013 ns  BUS RELEASED    RD                 posID=(2,3)
30513 ns  GRANT           RESP  0x00000072   posID=(3,2)   # Response@Task(7,2)

33608 ns     Task graph 0 completed. Now preparing for a new cycle:
             Restoring relation matrix
```

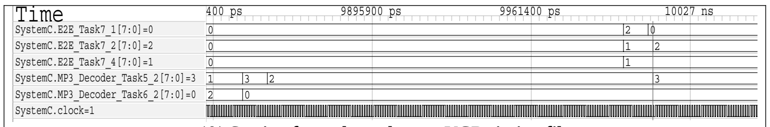
Figure 10.11: Simulation result using simple bus topology. Simulation runtime: 35000ns. Comments have been added manually to make it more readable. Comments follows #.

First observation is that the fictive end-to-end task graph manages to executes three times, before the MP3 decoder task graph completes at 33608ns. The fictive end-to-end task finishes execution at 6077ns, 16077ns and 26077ns corresponding to the expected period of 10000ns. Further, it always meets the deadline. However, the MP3 decoder task graph misses the deadline at 25000ns, during the execution of the last two task. This is because the actual task graph execution has been significantly delayed, due to bus contention caused by the large data transfer associated with the fictive end-to-end task. Referring back to the timing figures from figure 10.10, this should come as no surprise, since the response coming from task(7,2) starts *before* the responses coming from task(2,2) and task(3,2) (MP3 decoder). Thus the response from task(7,2) occupies the bus for 5000ns, causing the responses asserted by task(2,2) and task(3,2) to be blocked. This blocking time contributes to the delayed execution of the MP3 decoder task graph and eventually the deadline to be missed.

From the SoC communication simulation data in figure 10.11, this is also what can be observed: At 513ns the response from task(7,2) starts. At 1625ns and 1626ns task(2,2) and task(3,2) start the response but are refused to use bus. Instead the response data get buffered in the SoC communication interface. At 5514ns the response phase for task(7,2) has completed and the bus is granted to the buffered responses coming from task(2,2) followed by the responses from task(3,2) afterward. A quick calculation shows that the overhead added to the execution time for the MP3 decoder, due to bus contention, becomes approx. 5514ns-1625ns = 3889ns. However, remember from the previous example that the task graph execution time for the MP3 decoder was 22775ns. Assuming this to be about the same for this example and taking into account the bus contention overhead, the expected execution time should be approx. 3889ns + 22775ns = 26664ns. But this does not comply very well with the actual execution time of 33608ns. A closer look in the *text based log file* and *task state VCD timing file* reveals the root cause is due to preemption of task(5,2) in PE3. For convenience, the sections of interests from the log file and VCD file are shown in figure 10.12. Preemption occurs since:

- Task(7,2) becomes ready for execution at 10012ns, after PE3 has received the read request from task(7,1). This happens during execution of task(5,2) and is associated with the second cycle of fictive end-to-end task.
- The scheduling policy is based on RM (i.e. shortest period$\rightarrow$highest priority) and the period for task(5,2) is larger than the period for task(7,2); that is. 60000ns >10000ns).

Thus task(7,2) executes for 500ns and starts the response phase of 5000x32-bit data words afterward, causing task(5,2) to be preempted for 5500ns. This can also be seen indirectly from figure 10.11, considering the data for the time interval from 8840ns to 16320ns. The preemption due to bad scheduling in PE3 means that the total overhead added to the MP3 decoder task graph execution becomes approx. 3889ns + 5500ns = 9389ns! Changing scheduling policy in PE3 to EDF will not remove the preemption overhead, since the deadline for the fictive end-to-end task always will be smaller than the deadline for the MP3 decoder task graph. Thus task(7,2) still has higher priority than task(5,2). However, *offsetting* the execution of the fictive end-to-end task with 2000ns removes the preemption overhead, due to a more optimal scheduling. Here the MP3 decoder task graph execution time reduces to 26686ns; almost equal to the expected execution time, including the bus contention overhead. Still, it does not meet the deadline.



(A) Section from the task state VCD timing file

```
                              .
                              .
 10012 ns     |OCP| PE3_TL0.IOdevice.slave: receiving BURST request.
              | S | Data handshake: yes
              | L | MCmd: RD, MAddr: 0x3071, MThreadID: 0x7
              | A | MBurstSeq: UNKN, MBurstLength: 5000, MBurstSingleReq: 1

 10012 ns     PE#3: task(7,1) (external task finished) -> scheduler
 10012 ns           synconizer: releases task(7,2)
 10012 ns     PE#3: scheduler (preempt) -> task(5,2)
 10012 ns     PE#3: scheduler (run) -> task(7,2)

 10012 ns     |OCP| soc_comm.tl0_io_b.master: Request completed
 10512 ns     PE#3: task(7,2) (response to NoC: task(7,3),addr=0x2072,dataUnits=5000)-> adaptor
 10512 ns     PE#3: scheduler (resume) -> task(5,2)
 10512 ns     PE#3: scheduler (preempt) -> task(5,2)
 10512 ns     PE#3: scheduler (start NoC Response) -> task(IO)

 10512 ns     |OCP| PE3_TL0.IOdevice.slave: Starting response phase.
              | S | SThreadID: 0x7, dataUnits: 5000
                              .
                              .
```

(B) Section from the ext based log file

Figure 10.12: Preemption of task(5,2) as seen in the task state VCD timing file and the log file at 10012ns. Possible states shown in the VCD plot is 0=idle|1=ready|2=run|3=preempted. The preemption of task(5,2) occurring *before* it actually starts executing is because it is being released by the scheduler when task(6,2) finishes, but becomes preempted immediately afterward, due to the IO task being launched in conjunction to the transmission of the response data from task(6,2).

Conclusion

A simple bus topology, based on the first-come-first-served principle, can be concluded not to be suitable in this example, due to the large data transfers associated with the fictive end-to-end task. This caused the MP3 decoder execution to be substantially delayed and eventually to miss the deadline. The simulation also exposed the scheduling in PE3 as being a potential problem, since it was found that under certain conditions (indirectly related to the bus contention), task(5,2) from the MP3 decoder task graph could become preempted by task(7,2) for 5500ns, causing the delayed MP3 decoder task graph execution time to increase even further. A solution to this was to offset the execution of the fictive end-to-end task with 2000ns.

Regarding the bus contention, it is to believe that introducing a more complex bus, for an example supporting TDM based transfers, would solve missed deadline issues for the MP3 decoder task graph; especially because the data transfer, associated with MP3 decoder task graph inter-task dependencies is relative small.

10.2.4 1D mesh topology simulation results

Next topology to explore is a 1D mesh. Changing to this topology is done by setting the `module` declaration parameter, `"soc_allocator"` equal to `1,4` in the configuration file, `example2.task`. First argument selects the topology (mesh) while the second argument selects the span. See also section 12.6.1, page 146) regarding mesh layout.

Running the simulation for 25000ns yields the result showed in figure 10.14. The data format from the SoC communication model is similar to what has been described for figure 10.14, except that the routing information, `posID` also identifies the next target node position (middle value). Further, the link used/released is identified by `resID`, where the first value identifies if it is a *forward* (1) or *return* (0) link. The next values identify the row and column in the associated mesh database and indirectly tells which link it is in the mesh. See also section 12.6.2, page 146. For convenience, figure 10.13 shows the mapping between the 1D mesh databases and `resID`. For example, consider the SoC communication data in figure 10.14 at 11ns. `posID` indicates data transfer from node 2 to 3, while resID indicates using the forward link between node 2 and 3.

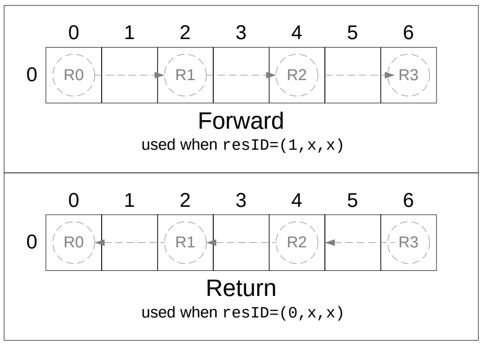


Figure 10.13: Mapping of `resID` to forward and return mesh databases.

The simulation result in figure 10.14 shows that no deadlines are missed. The fictive end-to-end task finishes execution at 6077ns and 16077ns, similar what was seen when using a bus topology (figure 10.11). More interesting is it to see that the MP3 decoder task graph now meets the deadline since no link contention occurs at all. Going back to the selected partitioning shown in figure 10.10, it is obvious that link contention is avoided, when using a 1D mesh topology, since the large data transfers associated with the fictive end-to-end task only happen between PE2 and PE3. Thus it does not affect the communication between PE0 and PE1, which was the case when using the bus topology. This is also what can be seen from

```
   11 ns  GRANT          RD    0x00003071   posID=(2,3,3) resID=(1,0,5) # RD@Task(7,1), R2->R3 link
   13 ns  LINK RELEASED  RD                 posID=(2,3,3) resID=(1,0,5)
   46 ns  GRANT          WR    0x00000011   posID=(1,0,0) resID=(0,0,1) # WR@Task(1,1), R1->R0 link
   57 ns  LINK RELEASED  WR                 posID=(1,0,0) resID=(0,0,1)
   76 ns  GRANT          RD    0x00000021   posID=(1,0,0) resID=(0,0,1) # RD@Task(2,1), R1->R0 link
   78 ns  LINK RELEASED  RD                 posID=(1,0,0) resID=(0,0,1)
   79 ns  GRANT          RD    0x00001031   posID=(0,1,1) resID=(1,0,1) # RD@Task(3,1), R0->R1 link
   81 ns  LINK RELEASED  RD                 posID=(0,1,1) resID=(1,0,1)
  513 ns  GRANT          RESP  0x00000072   posID=(3,2,2) resID=(0,0,5) # Response@Task(7,2), R3->R2 link
 1625 ns  GRANT          RESP  0x00000022   posID=(0,1,1) resID=(1,0,1) # Response@Task(2,2), R0->R1 link
 1626 ns  GRANT          RESP  0x00000032   posID=(1,0,0) resID=(0,0,1) # Response@Task(3,2), R1->R0 link
 1636 ns  LINK RELEASED  RESP               posID=(0,1,1) resID=(1,0,1)
 1637 ns  LINK RELEASED  RESP               posID=(1,0,0) resID=(0,0,1)
 2240 ns  GRANT          WR    0x00001033   posID=(0,1,1) resID=(1,0,1) # WR@Task(3,3), R0->R1 link
 2251 ns  LINK RELEASED  WR                 posID=(0,1,1) resID=(1,0,1)
 4937 ns  GRANT          WR    0x00000041   posID=(1,0,0) resID=(0,0,1) # WR@Task(4,1), R1->R0 link
 4948 ns  LINK RELEASED  WR                 posID=(1,0,0) resID=(0,0,1)
 5055 ns  GRANT          RD    0x00003061   posID=(1,2,3) resID=(1,0,3) # RD@Task(6,1), R1->R2 link
 5057 ns  LINK RELEASED  RD                 posID=(1,2,3) resID=(1,0,3)
 5057 ns  GRANT          RD    0x00003061   posID=(1,3,3) resID=(1,0,5) # RD@Task(6,1), R2->R3 link
 5058 ns  LINK RELEASED  RD                 posID=(1,3,3) resID=(1,0,5)
 5058 ns  GRANT          RD    0x00003051   posID=(0,1,3) resID=(1,0,1) # RD@Task(5,1), R0->R1 link
 5060 ns  LINK RELEASED  RD                 posID=(0,1,3) resID=(1,0,1)
 5060 ns  GRANT          RD    0x00003051   posID=(0,2,3) resID=(1,0,3) # RD@Task(5,1), R1->R2 link
 5061 ns  LINK RELEASED  RD                 posID=(0,2,3) resID=(1,0,3)
 5061 ns  GRANT          RD    0x00003051   posID=(0,3,3) resID=(1,0,5) # RD@Task(5,1), R2->R3 link
 5062 ns  LINK RELEASED  RD                 posID=(0,3,3) resID=(1,0,5)
 5514 ns  LINK RELEASED  RESP               posID=(3,2,2) resID=(0,0,5)
 6016 ns  GRANT          WR    0x00003073   posID=(2,3,3) resID=(1,0,5) # WR@Task(7,3), R2->R2 link
 6067 ns  LINK RELEASED  WR                 posID=(2,3,3) resID=(1,0,5)

 6077 ns     Task graph 1 completed. Now preparing for a new cycle:
             Restoring relation matrix
 6973 ns  GRANT          RESP  0x00000052   posID=(3,2,0) resID=(0,0,5) # Response@Task(5,2), link R3->R2
 6984 ns  LINK RELEASED  RESP               posID=(3,2,0) resID=(0,0,5)
 6984 ns  GRANT          RESP  0x00000052   posID=(3,1,0) resID=(0,0,3) # Response@Task(5,2), link R2->R1
 6995 ns  LINK RELEASED  RESP               posID=(3,1,0) resID=(0,0,3)
 6995 ns  GRANT          RESP  0x00000052   posID=(3,0,0) resID=(0,0,1) # Response@Task(5,2), link R1->R0
 7006 ns  LINK RELEASED  RESP               posID=(3,0,0) resID=(0,0,1)
 7373 ns  GRANT          RESP  0x00000062   posID=(3,2,1) resID=(0,0,5) # Response@Task(6,2), link R3->R2
 7384 ns  LINK RELEASED  RESP               posID=(3,2,1) resID=(0,0,5)
 7384 ns  GRANT          RESP  0x00000062   posID=(3,1,1) resID=(0,0,3) # Response@Task(6,2), link R2->R1
 7395 ns  LINK RELEASED  RESP               posID=(3,1,1) resID=(0,0,3)
10011 ns  GRANT          RD    0x00003071   posID=(2,3,3) resID=(1,0,5) # RD@Task(7,1), link R2->R3
10013 ns  LINK RELEASED  RD                 posID=(2,3,3) resID=(1,0,5)
10513 ns  GRANT          RESP  0x00000072   posID=(3,2,2) resID=(0,0,5) # Response@Task(7,2), link R3->R2
15514 ns  LINK RELEASED  RESP               posID=(3,2,2) resID=(0,0,5)
16016 ns  GRANT          WR    0x00003073   posID=(2,3,3) resID=(1,0,5) # WR@Task(7,3), link R3->R2
16067 ns  LINK RELEASED  WR                 posID=(2,3,3) resID=(1,0,5)

16077 ns     Task graph 1 completed. Now preparing for a new cycle:
             Restoring relation matrix
20011 ns  GRANT          RD    0x00003071   posID=(2,3,3) resID=(1,0,5) # RD@Task(7,1), link R2->R3
20013 ns  LINK RELEASED  RD                 posID=(2,3,3) resID=(1,0,5)
20513 ns  GRANT          RESP  0x00000072   posID=(3,2,2) resID=(0,0,5) # Response@Task(7,2), link R3->R2

24682 ns     Task graph 0 completed. Now preparing for a new cycle:
             Restoring relation matrix
```

Figure 10.14: Simulation result using 1D mesh topology. Simulation runtime: 25000ns. Comments have been added manually to make it more readable. Comments follows #.

the simulation results, where the link used for the response from task(7,2) is the R3→R2 link (513ns), while the links used for the responses from task(2,2) and task(3,2) is R0→R1 (1625ns) and R1→R0 (1626ns) respectively.

Another important issue to notice is the avoidance of link contention also does that the potential scheduling problem in PE3 will not occur. This can be seen, since the responses from task(5,2) and task(6,2) in PE3 starts at 6973ns and 7373ns respectively, while the second execution cycle of task(7,2) starts at 10012ns. Thus the release of task(7,2) does not cause preemption, since task(5,2) and task(6,2) has already completed. In conjunction to this, the timing headroom is approx. 10012ns - 7373ns = 2639ns.

Conclusion

The simulation using a 1D mesh NoC as SoC communication topology showed that all deadlines were met and that no link contention occurred. Besides this, the potential scheduling problem in PE3 did not occurred, due to the avoidance of the link contention. Thus is can be concluded that the 1D mesh topology is to prefer for this example.

10.2.5 2D mesh topology simulation results

The last SoC communication topology to explore is a 2D mesh. Changing to this topology is done by setting the `module` declaration parameter, `"soc_allocator"` equal to `1,2` in the configuration file, `example2.task`. First argument selects the topology (mesh) while the second argument selects the span. See also section 12.6.1, page 146) regarding mesh layout.

Running the simulation for $25000ns$ yields the simulation result showed in figure 10.16. The simulation result format is similar to figure 10.14. For convenience, figure 10.13 shows the mapping between the 2D mesh databases and `resID`. For example, consider the SoC communication data in figure 10.14 at 11ns. `posID` indicates data transfer from node 2 to 3, while resID indicates using the forward link between node 2 and 3.

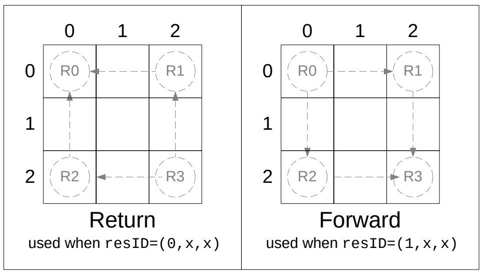


Figure 10.15: Mapping of `resID` to forward and return mesh databases.

As it can be seen from the simulation result in figure 10.16, it is almost identical when using 1D mesh topology (figure 10.14): no missed deadlines, no link contention and similar task graph execution times. This is also expected since the 2D mesh layout does not affect the communication between PE0-PE1 and PE2-PE (where most of the communication occurs). Notice however the differences in the link usage associated with read request from task(5,1) in PE0 and task (6,1) in PE1 and the associated responses from task(5,2) and task(6,2) in PE3.

Conclusion

Usage of a SoC communication topology based on a 2D mesh showed similar system performance as when using a 1D mesh. For the particular application, task mapping and no.of PE's, a 2D mesh topology is not to prefer, since nothing is

```
   11 ns  GRANT           RD    0x00003071   posID=(2,3,3)   resID=(1,2,1) # RD@Task(7,1), R2->R3 link
   13 ns  LINK RELEASED   RD                 posID=(2,3,3)   resID=(1,2,1)
   46 ns  GRANT           WR    0x00000011   posID=(1,0,0)   resID=(0,0,1) # WR@Task(1,1), R1->R0 link
   57 ns  LINK RELEASED   WR                 posID=(1,0,0)   resID=(0,0,1)
   76 ns  GRANT           RD    0x00000021   posID=(1,0,0)   resID=(0,0,1) # RD@Task(2,1), R1->R0 link
   78 ns  LINK RELEASED   RD                 posID=(1,0,0)   resID=(0,0,1)
   79 ns  GRANT           RD    0x00001031   posID=(0,1,1)   resID=(1,0,1) # RD@Task(3,1), R0->R1 link
   81 ns  LINK RELEASED   RD                 posID=(0,1,1)   resID=(1,0,1)
  513 ns  GRANT           RESP  0x00000072   posID=(3,2,2)   resID=(0,2,1) # Response@Task(7,2), R3->R2 link
 1625 ns  GRANT           RESP  0x00000022   posID=(0,1,1)   resID=(1,0,1) # Response@Task(2,2), R0->R1 link
 1626 ns  GRANT           RESP  0x00000032   posID=(1,0,0)   resID=(0,0,1) # Response@Task(3,2), R1->R0 link
 1636 ns  LINK RELEASED   RESP               posID=(0,1,1)   resID=(1,0,1)
 1637 ns  LINK RELEASED   RESP               posID=(1,0,0)   resID=(0,0,1)
 2240 ns  GRANT           WR    0x00001033   posID=(0,1,1)   resID=(1,0,1) # WR@Task(3,3), R0->R1 link
 2251 ns  LINK RELEASED   WR                 posID=(0,1,1)   resID=(1,0,1)
 4937 ns  GRANT           WR    0x00000041   posID=(1,0,0)   resID=(0,0,1) # WR@Task(4,1), R1->R0 link
 4948 ns  LINK RELEASED   WR                 posID=(1,0,0)   resID=(0,0,1)
 5055 ns  GRANT           RD    0x00003061   posID=(1,3,3)   resID=(1,1,2) # RD@Task(6,1), R1->R3 link
 5057 ns  LINK RELEASED   RD                 posID=(1,3,3)   resID=(1,1,2)
 5058 ns  GRANT           RD    0x00003051   posID=(0,2,3)   resID=(1,1,0) # RD@Task(5,1), R0->R2 link
 5060 ns  LINK RELEASED   RD                 posID=(0,2,3)   resID=(1,1,0)
 5060 ns  GRANT           RD    0x00003051   posID=(0,3,3)   resID=(1,2,1) # RD@Task(5,1), R2->R3 link
 5061 ns  LINK RELEASED   RD                 posID=(0,3,3)   resID=(1,2,1)
 5514 ns  LINK RELEASED   RESP               posID=(3,2,2)   resID=(0,2,1)
 6016 ns  GRANT           WR    0x00003073   posID=(2,3,3)   resID=(1,2,1) # WR@Task(7,3), R2->R3 link
 6067 ns  LINK RELEASED   WR                 posID=(2,3,3)   resID=(1,2,1)

 6077 ns     Task graph 1 completed. Now preparing for a new cycle:
             Restoring relation matrix
 6973 ns  GRANT           RESP  0x00000052   posID=(3,1,0)   resID=(0,1,2) # Response@Task(5,2), R3->R1 link
 6984 ns  LINK RELEASED   RESP               posID=(3,1,0)   resID=(0,1,2)
 6984 ns  GRANT           RESP  0x00000052   posID=(3,0,0)   resID=(0,0,1) # Response@Task(5,2), R1->R0 link
 6995 ns  LINK RELEASED   RESP               posID=(3,0,0)   resID=(0,0,1)
 7373 ns  GRANT           RESP  0x00000062   posID=(3,1,1)   resID=(0,1,2) # Response@Task(6,2), R3->R1 link
 7384 ns  LINK RELEASED   RESP               posID=(3,1,1)   resID=(0,1,2)
10011 ns  GRANT           RD    0x00003071   posID=(2,3,3)   resID=(1,2,1) # RD@Task(7,1), R2->R3 link
10013 ns  LINK RELEASED   RD                 posID=(2,3,3)   resID=(1,2,1)
10513 ns  GRANT           RESP  0x00000072   posID=(3,2,2)   resID=(0,2,1) # Response@Task(7,2), R3->R2 link
15514 ns  LINK RELEASED   RESP               posID=(3,2,2)   resID=(0,2,1)
16016 ns  GRANT           WR    0x00003073   posID=(2,3,3)   resID=(1,2,1) # WR@Task(7,3), R2->R3 link
16067 ns  LINK RELEASED   WR                 posID=(2,3,3)   resID=(1,2,1)

16077 ns     Task graph 1 completed. Now preparing for a new cycle:
             Restoring relation matrix
20011 ns  GRANT           RD    0x00003071   posID=(2,3,3)   resID=(1,2,1) # RD@Task(7,1), R2->R3 link
20013 ns  LINK RELEASED   RD                 posID=(2,3,3)   resID=(1,2,1)
20513 ns  GRANT           RESP  0x00000072   posID=(3,2,2)   resID=(0,2,1) # Response@Task(7,2), R3->R2 link

24672 ns     Task graph 0 completed. Now preparing for a new cycle:
             Restoring relation matrix
```

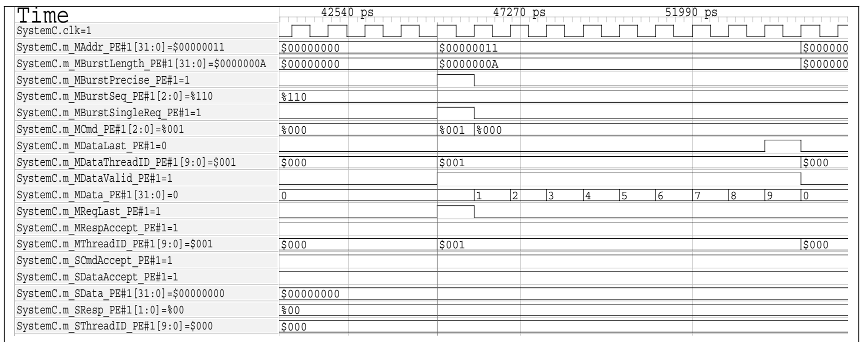
Figure 10.16: Simulation result using 2D mesh topology. Simulation runtime: 25000ns. Comments have been added manually to make it more readable. Comments follows #.

gained from the increased bandwidth. Thus a 2D mesh is an over-dimensioned solution, adding unnecessary production cost. However if more applications are to be added later and/or the SoC communication traffic increases, the 2D mesh might be the solution to prefer.

10.2.6 SoC communication interface TL mixture

This example also demonstrates the possibility of doing abstraction level mixture in the SoC communication interface. Referring back to architectures shown in 10.9, page 72. It can be seen PE0 and PE2 interface to the SoC communication platform using TL1, while PE1 and PE3 use TL0.

As an example on the abstraction level mixture, figure 10.17 shows the communication traces, related to the write request issued at 45ns from task(1,1) in PE1 to task(3,1) in PE0. The upper trace shows the TL0 trace from PE1 while the lower trace is a section from the OCP TL1 monitor file[4] of the OCP channel, connecting to the OCP slave in PE3. The traces also show that it is a single request burst write, using data handshake and with a burst length of 10.


Time
42540 ps
47270 ps
51990 ps
SystemC.clk=1
SystemC.m_MAddr_PE#1[31:0]=$00000011
SystemC.m_MBurstLength_PE#1[31:0]=$0000000A
SystemC.m_MBurstPrecise_PE#1=1
SystemC.m_MBurstSeq_PE#1[2:0]=%110
SystemC.m_MBurstSingleReq_PE#1=1
SystemC.m_MCmd_PE#1[2:0]=%001
SystemC.m_MDataLast_PE#1=0
SystemC.m_MDataThreadID_PE#1[9:0]=$001
SystemC.m_MDataValid_PE#1=1
SystemC.m_MData_PE#1[31:0]=0
SystemC.m_MReqLast_PE#1=1
SystemC.m_MRespAccept_PE#1=1
SystemC.m_MThreadID_PE#1[9:0]=$001
SystemC.m_SCmdAccept_PE#1=1
SystemC.m_SDataAccept_PE#1=1
SystemC.m_SData_PE#1[31:0]=$00000000
SystemC.m_SResp_PE#1[1:0]=%00
SystemC.m_SThreadID_PE#1[9:0]=$000
$00000000
$00000011
$000000
$0000000A
%110
%000
%001
$000
$001
0 1 2 3 4 5 6 7 8 9 0
%00

(A) OCP2.0 TL0 trace from PE1 (OCP Master)

```
                              .
                              .
44  0 xxxx x xxxx x x x x 0 xxxxxxxx 0 x x 0 0 x xxxxxxxx 000
45  0 xxxx x xxxx x x x x 0 xxxxxxxx 0 x x 0 0 x xxxxxxxx 000
46  0 xxxx x xxxx x x x x 0 xxxxxxxx 0 x x 0 0 x xxxxxxxx 000
47  1 0011 1 000a 1 6 1 1 1 00000000 1 1 0 1 0 x xxxxxxxx 000
48  0 xxxx x xxxx x x x x 1 00000001 1 1 0 1 0 x xxxxxxxx 000
49  0 xxxx x xxxx x x x x 1 00000002 1 1 0 1 0 x xxxxxxxx 000
50  0 xxxx x xxxx x x x x 1 00000003 1 1 0 1 0 x xxxxxxxx 000
51  0 xxxx x xxxx x x x x 1 00000004 1 1 0 1 0 x xxxxxxxx 000
52  0 xxxx x xxxx x x x x 1 00000005 1 1 0 1 0 x xxxxxxxx 000
53  0 xxxx x xxxx x x x x 1 00000006 1 1 0 1 0 x xxxxxxxx 000
54  0 xxxx x xxxx x x x x 1 00000007 1 1 0 1 0 x xxxxxxxx 000
55  0 xxxx x xxxx x x x x 1 00000008 1 1 0 1 0 x xxxxxxxx 000
56  0 xxxx x xxxx x x x x 1 00000009 1 1 1 1 0 x xxxxxxxx 000
57  0 xxxx x xxxx x x x x 1 xxxxxxxx 0 x x 1 0 x xxxxxxxx 000
                              .
                              .
```

(B) Section from the OCP2.0 TL1 monitor file for PE0 (OCP slave)

Figure 10.17: OCP2.0 TL0 and TL1 SoC communication traces from PE1 and PE0.

10.2.7 Summary

The aim with this example was to show how the extended abstract PE model and the SoC communication platform models can be integrated and used to evaluate different SoC communication topologies as well as doing abstraction level mixture in the SoC communication interface. The system level model was based on four PE's and two applications running on top (modeled using the MP3 decoder task graph and a fictive end-to-end task). Three topologies was considered: a simple bus and a NoC based on a 1D and a 2D mesh. The bus topology showed to introduce significant system performance degradation, due to bus contention. This eventually causing the deadline for the MP3 decoder task graph to be missed. It also revealed a potential scheduling problem i PE3. Simulation using a NoC based on a 1D and 2D mesh showed identical performance, where no deadlines were missed and

[4]Graphical analysis requires the SOCCREATOR tool available from OCP-IP corporation.

no link contention occurred. Assuming the SoC communication traffic will not increase/change (i.e. if a new application was to be added later), the simulations showed that a 1D mesh topology is to prefer for this example.

10.3 Example 3: Complex system performance behavior analysis.

The last example demonstrates how the models can be used for performing a more behavior based performance analysis of a complex system level model, by tweaking different system parameters. The simulation framework consists of nine PE's and two applications running on top. All inter-processor communication is based on OCP2.0 TL1.

In the example the system performance will be evaluated, relative to tweaking the following parameters:

- RTOS scheduling policy.
- Data transfer size associated with inter-task dependency.
- SoC communication topology.

Task graph partitioning will be fixed in the different simulation scenarios, but selected in an almost random fashion to introduce complex (or somewhat pseudo random) SoC communication traffic patterns.

The source code and configuration file for this example can be found on the enclosed CD-ROM in the directory: `/ARTS_Model/builds/example3`.

10.3.1 Application modeling

The two application models are based on the MP3 decoder and GSM decoder task graphs [23]. To support read transaction, the MP3 decoder task has been grouped into six end-to-end task, similar to the approach being used in example 2. See also figure 10.10, page 73. Any SoC communication associated with GSM decoder inter-task dependency will be modeled as write requests. Further, execution time for all tasks has been changed to from a random to a fixed execution time equal to WCET. This has been done to be able to compare the outcome from the different simulation scenarios.

Figure 10.18 shows the timing figures and the task graph for the GSM decoder, while figure 10.19 shows the task partitioning.

Task ID	ET [ns]	dl [ns]	T [ns]
1,1	121	20000	30000
2,1	8	20000	30000
3,1	8	20000	30000
4,1	8	20000	30000
5,1	8	20000	30000
6,1	4	20000	30000
7,1	41	20000	30000
8,1	17	20000	30000
9,1	124	20000	30000
10,1	41	20000	30000
11,1	17	20000	30000
12,1	124	20000	30000
13,1	41	20000	30000
14,1	17	20000	30000
15,1	124	20000	30000
16,1	41	20000	30000
17,1	17	20000	30000
18,1	124	20000	30000
19,1	84	20000	30000
20,1	6	20000	30000
21,1	121	20000	30000
22,1	5	20000	30000
23,1	23	20000	30000
24,1	638	20000	30000
25,1	5	20000	30000
26,1	23	20000	30000
27,1	688	20000	30000
28,1	5	20000	30000
29,1	23	20000	30000
30,1	638	20000	30000
31,1	44	20000	30000
32,1	23	20000	30000
33,1	5893	20000	30000
34,1	655	20000	30000

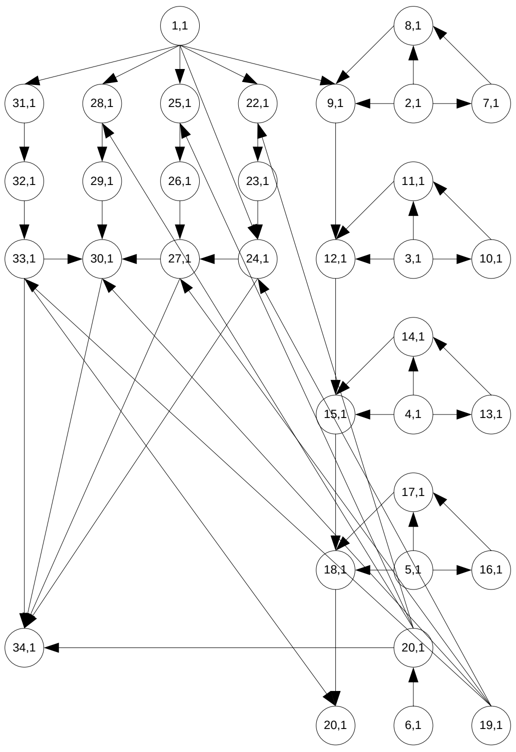


Figure 10.18: GSM decoder timing parameters and task graph.

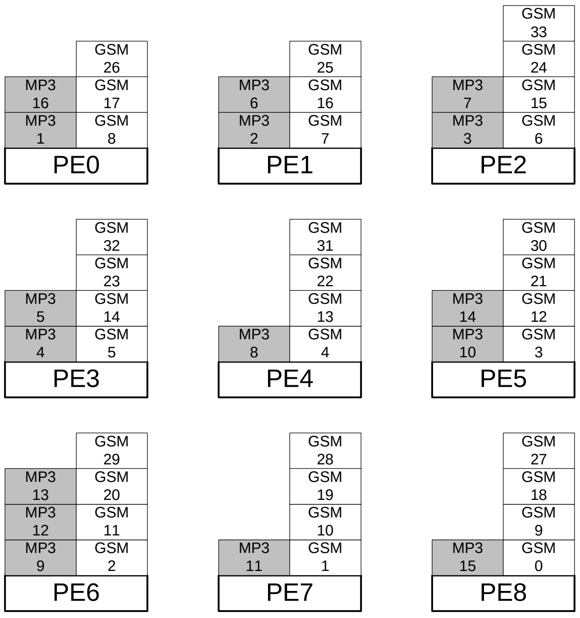


Figure 10.19: MP3 and GSM decoder task graph partitioning.

10.3.2 Simulation results

Simulations scenarios were done using a combination of the following conditions:

- Common inter-task dependency data transfer size: 10,20,50,100 x 32bit.
- RTOS scheduling policy: RM and EDF.
- SoC communication topology: bus, 1D mesh and 2D mesh.

Since the period of the GSM and MP3 decoder applications are 30000ns and 60000ns respectively, the simulation run time is selected to 60000ns. Using longer simulation run time will not provide any further useful information, since the applications are not mutual aperiodically.

The collected data for each simulation is the SoC communication contention count and the MP3 and GSM decoder task graph execution times. The results are summarized in table 10.20, 10.21 and 10.22.

	10x32-bit		20x32-bit		50x32-bit		100x32-bit	
	RM	EDF	RM	EDF	RM	EDF	RM	EDF
Bus	81	75	86	82	86	85	86	87
1D mesh	64	62	72	69	80	78	90	83
2D mesh	25	23	22	21	27	25	28	27

Figure 10.20: SoC communication refuse count.

	10x32-bit		20x32-bit		50x32-bit		100x32-bit	
	RM	EDF	RM	EDF	RM	EDF	RM	EDF
Bus	32833	31807	33373	32257	34107	33606	36607	35947
1D mesh	31988	31692	32333	31987	33454	32857	40605	34307
2D mesh	32773	31768	33171	32055	34346	33061	36387	34852

Figure 10.21: MP3 task graph execution time [ns].

	10x32-bit		20x32-bit		50x32-bit		100x32-bit	
	RM	EDF	RM	EDF	RM	EDF	RM	EDF
Bus	8528	8283	9001	8756	10890	10816	13090	13166
	8540	9929	8976	10502	10784	11228	12884	15108
1D mesh	9202	8313	9551	8552	10653	9277	10605	10473
	9202	10653	9551	11179	10652	12762	10605	13919
2D mesh	8477	8232	8760	8454	9310	9042	10428	9942
	8477	9915	8760	10312	9310	11647	10428	13889

Figure 10.22: GSM task graph execution times for first and second execution cycle [ns].

When running the different simulations it was found that the MP3 decoder task graph was not able to meet the deadline. This was also to expect, due to the selected partitioning and the fact that it has a tight deadline (25000n), relative to the expected execution time (approx. 22500ns). For the GSM decoder task graph the deadline was always met. But also this was expected, since the deadline (20000ns) is not so tight, relative to the expected execution time (approx. 10000ns).

From the tables the following general trends can be observed.

- Link contention as well as execution times increase as the data transfer size increases.
- Link contention reduces when moving from bus topology to 1D mesh and finally 2D mesh.
- Second execution cycle of the GSM decoder task graph increases when using EDF scheduling (since pending MP3 decoder tasks will have higher priority, because the tasks are closer to their deadlines that the GSM decoder tasks).

These results are expected. However, an interesting observation is the EDF scheduling in general introduces better system performance than RM scheduling, with respect to shorter execution times and reduced link contention.

Conclusion

This example showed how the models can be used for a behavior based performance analysis of a more complex simulation framework, consisting of nine PE's and two applications running on top. By selecting different RTOS scheduling policies, SoC communication topologies and data transfer sizes, associated with inter-task dependencies, it was possible to observe how the system performed, relative to the selected task partitioning and with respect to execution times and link contentions. The simulations showed a general better system performance using EDF scheduling policy compared to RM scheduling policy.

Chapter 11

Implementation: Abstract PE model

This chapter presents the implementation specific details of the different modules, forming the extended abstract PE model. It is *highly* recommended to use the source code for reference, when reading this chapter. The source code can be found on the enclosed CD-ROM and may be used as reference. Please consult the README file for a directory contents description.

11.1 Abstract PE model modifications

The aim has been to avoid modifying the original abstract PE model as much as possible, to keep the simplicity and modularity of the model. Thus new extensions like the IO task, parser module, performance monitor etc. must be able to be added/removed to the framework as desired.

The new model is not backward compatible. Thus modules from the original model cannot be used. However, simulation frameworks (e.g. a single abstract PE or the abstract MPSoC NoC model [6]) with a behavior similar to the originals, can be constructed using the new modules. The lack of backward compatibility is mostly due to a change in the link model, with respect to message passing, but also due to the introduction of the global synchronization database (dependency controller module), required by the new synchronizer to support periodical task graph execution. However these changes can be considered as improvement to the original model.

11.1.1 Communication link

To reduce simulation time, the `sc_link_mp` communication data-type, has been changes from struct messages (`message_type`) to *pointers* to struct messages (`message_type*`). Investigations have shown a speed improvement of 10...20% using this approach. See appendix B, page 165 for more information.

Since the communication is based on producer/consumer style, the approach has been create the message in the producer, using `new` operator and delete the message in the target consumer module, using `delete` operator, when it has been processed. In some situations, a message might be reused and send back again to the initiator module, to avoid creating a new message (further performance improvement). The two approaches are illustrated in figure 11.1A and 11.1B respectively.

11.1.2 High-level message struct extension

Extra fields have been added to the message struct (`message_type`) to control the IO task. These fields define for an example what action to be performed by the IO task, the data transfer size etc. It must be emphasized that these extensions are "invisible" to the RTOS modules (synchronizer, resource allocator, scheduler), in the sense that the modules does not access/uses these fields at all. Table 11.1 gives a summary of the field extension. For reference, the original struct message can be found in table 5.1 in section 5.4.

Type	Name	Description
`unsigned int`	`initiatorTaskID`	The ID of the task issuing an inter-processor communication event. †
`unsigned int`	`threadID`	A thread ID associated with the inter-processor communication.
`unsigned int`	`type`	Identifies the type of action to perform by the IO task (e.g. `WR` for write request). See also section 11.5, page 108.
`unsigned int`	`dataUnits`	Data transfer size for the inter-processor communication event.
`vector<soc_comm_nfo_type>*`	`soc`	Pointer to a vector containing struct objects with address information of the target PE's and the ID of the external tasks having data succeeding dependencies with the task initiating the inter-processor communication event. See also section 11.1.4, page 92.†
† *Only applicable when a local task initiates an inter-processor communication event.*		

Table 11.1: High-level message struct extensions.

The exact functionality of the different fields will be described in section 11.5, covering the implementation of the IO task.

11.1.3 RTOS modules

The RTOS modules are the synchronizer, resource allocator and scheduler. In general, only few changes have been done in these module. Common for all mod-

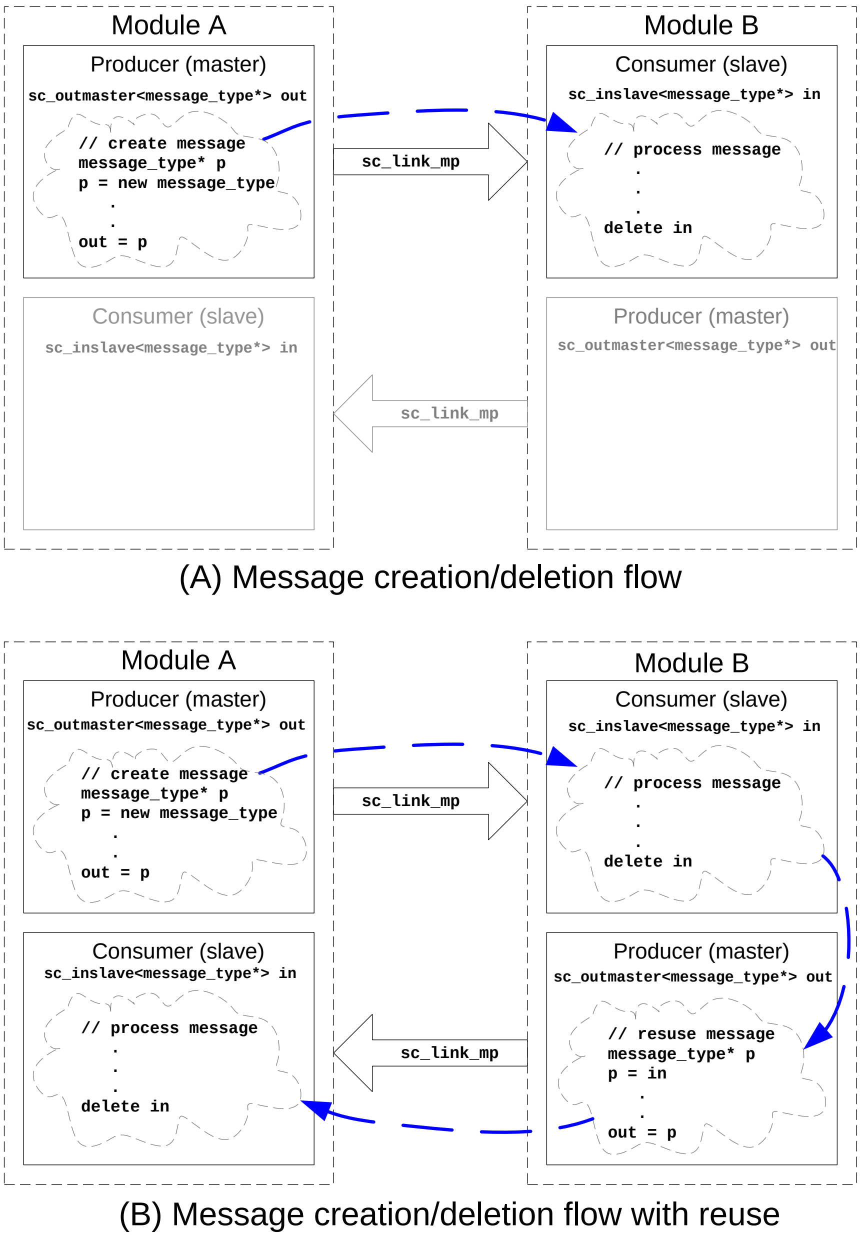


Figure 11.1: Message creation/deletion flow.

ule is the change in the communication link to manage the new approach using pointers to messages. The main change has been done in the synchronizer module to support communication with the dependency controller module and to support release of the IO task, when a task with inter-task dependencies requests for an inter-processor communication event.

A general bug fix: memory leakage

All original modules used dynamic memory allocation when creating communication link messages. This was done using the `new` operator. However, no memory clean-up, using `delete` operator, was performed after a message was asserted onto the communication link. Thus very long simulations could eventually cause an out-of-memory situation, since all previous messages remained intact in the memory.

EDF scheduler

Small errors were found in the EDF scheduler. One issue was related to an inverted priority handling, which caused the highest priority to be assigned to a task with the largest deadline. The error was due to an incorrect comparison `operator` declaration of $<$ and $\leq$ (declared in `message_type` struct), causing incorrect sorting of the task priority queue, implemented using a `vector` list.

Another issue was the lack of support for messages coming from the resource allocator. Thus a running task, requesting an already occupied resource would not be preempted, since the scheduler did not responded to the `REFUSE`-message from the resource allocator.

The above issues have been fixed in the new EDF scheduler module.

RM scheduler

The RM scheduler originally used a `priority_queue` for the task queue. However, the changed comparison `operator`-declaration of $<$ and $\leq$, to fix the EDF scheduler issues, caused inverted priority sorting, when using `priority_queue`. To fix this issue, a `vector` list is being used in the new RM scheduler module instead[1].

DS Synchronizer

In the synchronizer module, the local dependency database (`relations`) and associated management methods have been replaced with calls to methods in the dependency controller module, which manages the global synchronization database. Accessing the methods are done through a pointer to the object, provided to the constructor of the synchronizer. In conjunction to this, a new *local task ID list* (`taskID_list`) has been implemented in the synchronizer, containing the *encoded* task ID[2] of the local tasks, assigned to a PE. The list is a `vector` object and is being initialized, during the construction of the PE module, by calling the new method, `push_taskID` whenever a new task object is created in the PE module. Argument to the method is the encoded task ID. The list is being used in context

[1]Using `priority_queue` in the EDF scheduler is not possible, due to the priority updating approach.

[2]see section 6.4.1, page 35

with dependency controller method calls, to ensure that *only* entries associated with the local tasks are considered/affect when accessing/modifying the global dependency database.

The dependency controller methods called are `finished` and `mask`. The `finish` method is called when a task (local or global) finishes execution. This is managed in the `synchronize` method. Arguments to the `finish` method are the finished task ID and the local task ID list. Providing the local task ID list as argument ensures that only the dependency database entries, associated with local tasks will be cleared. The `mask` method is called in `check_pending_task_queue` when the synchronizer checks if the dependencies for a particular task, located in the pending task list, has been resolved. The method returns false, if all dependencies has been resolved. Otherwise true. Argument to the method is the encoded task ID.

The modifications done in the synchronizer to support release of the IO task are very simply. Whenever a local task, with preceding inter-task dependency, completes execution, it sends a `SOC_TRANSFER` message to the synchronizer, instead of a `FINISHED` message. This causes the synchronizer to perform the following actions:

1. The global dependency database is updated by calling the `finish` method, with the encoded task ID as argument. The pending task list is *not* checked afterward, since the IO task is to be released.

2. The message is changed to a `FINISHED` message (by altering the `comm`-field in the message) and then forwarded to the scheduler.

3. Immediately afterward a *duplicated* version of the initial `SOC_TRANSFER` message is changed to a `READY` message with the subtask ID (`tnum` now being equal to the IO task (defined by `_IOTASK_ID`). Also, the period (`tper` and deadline (`tdl`) is changed to 1 to ensure the IO task has the highest priority. Then the message is forwarded to the scheduler.

No other actions are required by the synchronizer, since the `SOC_TRANSFER` message contains all information required for the IO task to start an inter-processor communication event. See also table 11.1. This information comes from the task initiating this.

For debug purpose a clocked thread process, `check_task_running` has been implemented, used for monitoring the number of tasks running. If more than one task is running, the process terminates the simulation and asserts a UI notification error. At the rising edge of the clock signal, the process evaluates the variable, `task_running`. The variable serves as a counter and depends on the *state messages*, issued by the tasks. A state message will be issued whenever a task starts (run/resume) or stops (finish/preempt) execution, causing the counter to be incremented or decremented respectively. The process and transmission of state messages, from the tasks, can be excluded by setting the conditional compiler flag, `_DB_CHECK_TASK_RUNNING` to false, before building the framework. Doing so will also reduce simulation time.

Some UI message reporting, for monitoring purpose, was also done by the synchronizer, whenever a pending subtask was released for execution or when a subtask was pushed onto the pending task queue. The new synchronizer now supports logging to a file and disabling/enabling of UI message logging to screen. In case of file logging, a pointer to an `ofstream` object must be provided to the constructor of the module. Disabling/enabling of logging to screen is controlled using a boolean, also provided the constructor.

11.1.4 Periodic task

The behavior of the new periodic task module is very similar to the original, with some extra functionality added on top. These are:

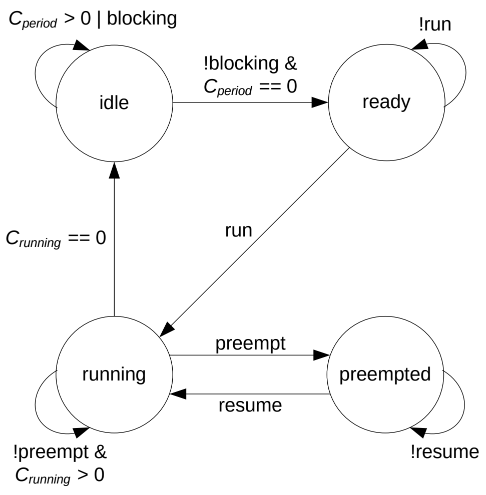
- Task self blocking.
- Dynamic resource requirement assignment.
- Support for inter-processor communication requesting and configuration.

Task self blocking

Task self blocking enabling/disabling is controlled by the boolean variable, `blocking_enable`. Default value, set in the module constructor is false, indicating blocking disabled. Thus the task behaves as default as the original task module. Enabling blocking is done by calling the task method, `set_blocking_flag`, with true as argument. This should normally be done before simulation starts. However, it is possible to disable/enable the blocking, during simulation, if it for some reason is needed (not done in this framework).

If blocking has been enabled, the actual state-of-blocking is controlled by the boolean variable, `blocking`. This variable is evaluated and set in the idle-state. If `blocking` is false (the default value, set in the constructor), the task is not blocked and it may issue a `READY` message whenever the period watchdog timer has expired. Simultaneously as the `READY` message is issued, the `blocking` variable will be set to true, indicating blocking. Thus when execution completes and the task goes back to idle-state, it remains in this state, until it becomes unblocked again and the period watchdog timer has expired. Unblocking a task is done by calling the task method, `unblock`. This clears the `blocking` variable.

For convenience the state machine for the new task module is shown in figure11.2, even though it is very similar to the original, shown in 5.3, page 23. By



$C_{period} > 0$ | blocking
!run
!blocking & $C_{period} == 0$
idle
ready
$C_{running} == 0$
run
preempt
running
preempted
resume
!preempt & $C_{running} > 0$
!resume

Figure 11.2: State machine for the new periodic task model.

comparing the figures, it can be seen the only change relates to the condition required for going from idle to ready state.

Dynamic resource requirement assignment

The new support for dynamic resource requirement assignment is based on the approach of having a vector, `resource_list` holding *resource requirement struct objects*, dynamically created and appended to the list. A resource requirement struct object (`resource_req_info_type`) contains the resource requirement information (RRT, CSL and resource ID) and the watchdog timers for RRT and CSL. The struct object is shown in 11.2. At each clock cycle, in the running-state, the resource list scanned and the watchdog timers in the different objects are checked and updated. If the RRT or CSL watchdog timer has expired, in a certain resource request object, a corresponding `REQUEST` or `RELEASE` message is issued to the resource allocator respectively.

Assigning a new resource requirement is done by calling the task method, `new_resource_requirement`, with resource ID, RRT and CSL as arguments. The method creates a new resource requirement object and push this onto the vector, `resource_list`. Assigning resources must be done before the simulation starts.

Inter-processor communication requesting and configuration

A task having one or more preceding inter-task dependencies must be configured to generate a `SOC_TRANSFER` message to the synchronizer when execution completes (causing the synchronizer to release the IO task afterward). The message is required to contain all relevant information, required by the IO task, to start an

Type	Name	Description
`unsigned int`	`ID`	Resource ID.
`unsigned int`	`RRT`	Resource request time.
`unsigned int`	`CSL`	Critical section length.
`unsigned int`	`RRT_timer`	Serves as a watchdog timer for RRT.
`unsigned int`	`CSL_timer`	Serves as a watchdog timer for CSL.

Table 11.2: Resource requirement struct, `resource_req_info_type`.

inter-processor communication event. This includes addresses for the target PE's, transfer type, data transfer size and thread ID. See also table 11.1 on page 88.

Transfer type, data transfer size and thread ID must always be provided to the constructor of the task module. They are kept in the variables, `soc_transfer_type`, `dataUnits` and `threadID` respectively and are to be included in the `SOC_TRANSFER` message. The new task module also keeps a vector, `soc` containing struct objects (`soc_comm_nfo_type`) with information about the addresses of the target PE's as well as the encoded task ID of non-local tasks, to which it has preceding inter-task dependency to. The struct object is shown in table 11.3.

Type	Name	Description
`unsigned int`	`subtaskID`	The unique subtask ID.
`unsigned int`	`addrLO`	Lower SoC communication address for the target PE.
`unsigned int`	`addrHI`	Upper SoC communication address for the target PE

Table 11.3: SoC communication struct object, `soc_comm_nfo_type`.

Pushing information on to the list is done by calling the method, `push_soc_comm_nfo` with the encoded non-local task ID and the low and high address of the target PE as arguments. This must be done before the simulation starts and is in this framework, managed by the task configuration method in the PE module. See also section 11.2, page 95.

Figure 11.3 shows an example of the contents of the vector, `soc` after configuration of task(1,1) having inter-task dependencies to task(2,1), task(3,1) and task(4,1).

Controlling if the task should issue a `FINISHED` of `SOC_TRANSFER` message is controlled by the state of the boolean variable, `soc_transfer_enable`. If a task has inter-task dependencies as described above the boolean must be set to true, causing the message to be a `SOC_TRANSFER`. This is done by calling the method, `set_soc_transfer` with true as argument. When a subtask issues a `SOC_TRANSFER` message, it includes a pointer to the `soc`-list as well as the other information, shown in table 11.1.

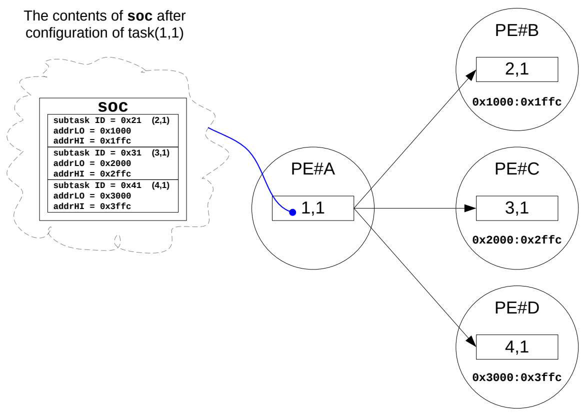


Figure 11.3: Example of the contents of `soc`-vector after configuration.

11.1.5 Monitor module

The monitor module, used for monitoring the real-time state of a PE, has been changed to support message logging to a file as well as disabling/enabling of logging to screen. A pointer to an `ofstream` object must be provided to the constructor of the module, in case of file logging. Disabling/enabling of logging to screen is controlled using a boolean, also provided to the constructor.

11.2 PE construction module

The PE construction module connects the RTOS modules, the periodic tasks, the IO task and IO device modules, into a structural forming the extended abstract PE model. Selection of RTOS modules and assignment of tasks is done dynamically, based on the configuration file declarations. The architecture of the module is shown in figure 11.4. See also figure 7.1, page 37 showing an example of a complete simulation framework, with multiple PE modules instantiated. Dotted lines indicate pointers to external objects, provided during module construction. For simplicity the monitor module as well as a pointer to an external `ofstream` object (for message monitoring logging to file) have been left out from figure 11.4.

A PE construction module is available for OCP2.0 TL0 and TL1. These are defined by the classes `PE_TL0` and `PE_TL1` respectively. Whenever a new abstract PE model is to be instantiated, these are the modules to use. The simplified UML class diagram for the abstract OCP2.0 TL0/TL1 PE module is shown in figure 11.5.

11.2.1 Module construction

The module construction can be divided into three steps:

1. **Module architecture construction**. RTOS, IO device and IO task modules

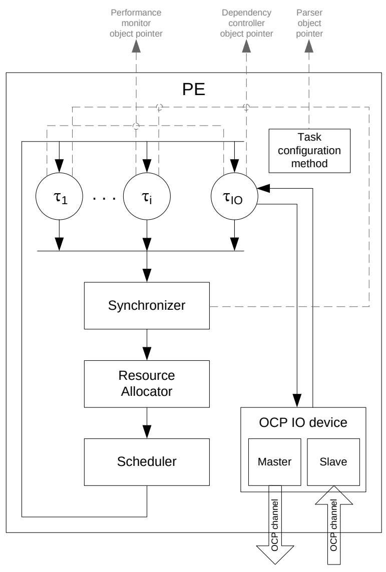


Figure 11.4: Architecture of the abstract PE module.

are created and connected. The selected RTOS modules depends on the protocol declarations done in the configuration file.

2. **Dynamic task creation**. Periodic task modules are created dynamically, based on the selected task partitioning.

3. **Inter-task dependency configuration**. Any created tasks having preceding inter-task dependency are configured to issue a `SOC_TRANSFER` message, when execution completes.

Step 1 and 2 is done in the module constructor, while step 3 is done by calling the method `configure_tasks`, after module construction. Step 1, 2 and 3 uses the module, task and dependency database declaration information, available from the parser, respectively. See also section 8.2.1, page 46.

Module architecture construction

The different modules, describing the architecture, are created and connected in the constructor of the PE module. The required arguments to the module constructor is the SystemC module name, the assigned PE ID, a boolean controlling disabling/enabling of UI monitoring logging to screen, and pointers to the dependency controller, performance monitor, log file and parser objects. The modules selected for construction of the RTOS will depend on the protocol declarations done in the configuration file. At first the `module` declaration for this PE is fetched by calling the parser method `module_search`, with `"peID"` as parameter name and the ID of the PE as parameter value. When a module declaration, containing these

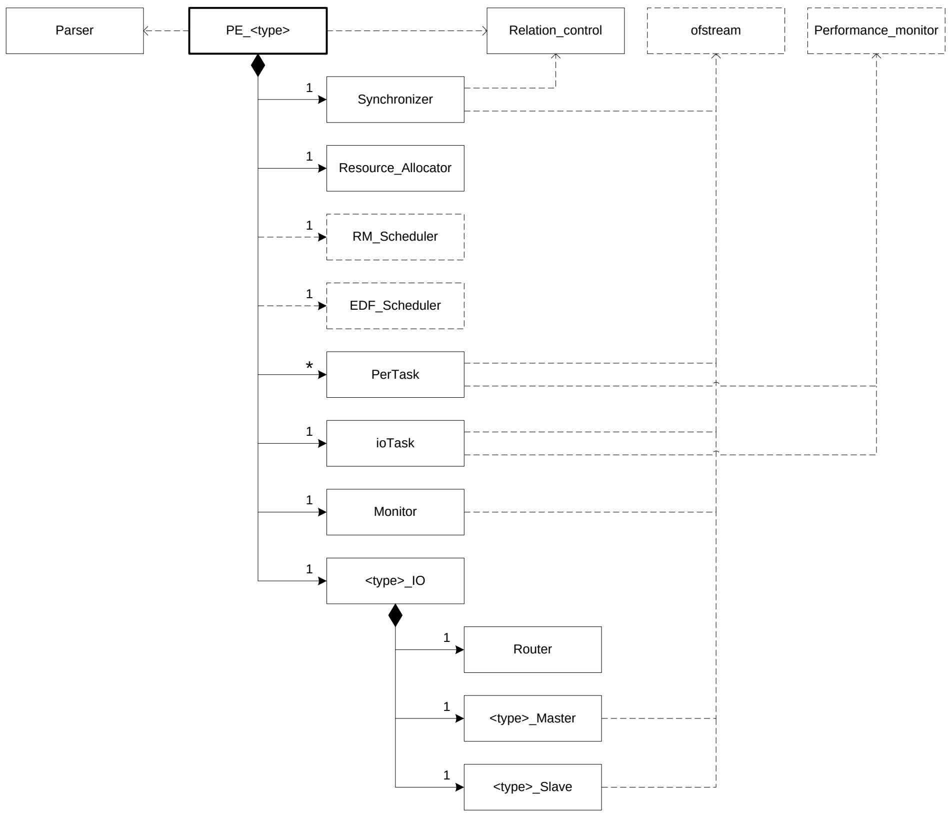


Figure 11.5: Simplified UML class diagram for the abstract PE module. `<type>` identifies if it is being a TL0 or TL1 PE module, where `<type>`=[TL0|TL1].

parameters, has been found, the method returns a pointer to a vector containing the module declarations. This vector is analyzed and the declarations for the synchronizer, resource allocator and scheduler is extracted and then used for selecting which RTOS modules to create and connect. Besides this, the message monitoring enable/disable flag is fetched from the module declaration and evaluated. Monitoring disabled causes the monitor message logging, to screen and log file, to disabled in all modules doing this[3].

If no module declaration exists for this PE or if there is an illegal or missing declarations, an error message will be asserted.

During module creation, the different constructor arguments are forwarded to modules as required. For an example, the pointer to the dependency controller is required by the synchronizer constructor.

Also created is the IO task and device module. Further, the assigned SoC communication address range for the PE is provided to the IO task. This is done by fetching the address range from the parser by calling the parser method `get_address`[4]

[3]These are the monitor, synchronizer, IO task, IO device and periodic task modules

[4]This is a macro method in the parser. The same information could also have been obtained from

with the PE ID as argument, followed by a call to IO task method, `set_address_range` with the returned address range as argument. If no address range has been assigned to the PE, an error message will be asserted.

Dynamic task creation

Periodic task objects are created dynamically, depending on the task declaration and partitioning, specified in the configuration file. This procedure follows, when the other modules have been created and connected. The task declaration list, available from the parser, is scanned and whenever a task has been assigned to current PE ID a new periodic task object is being created and connected. Getting an entry in the task declaration list is done by calling the parser method `get_task_from_list` with an index as argument. The method returns a pointer to struct, containing the task declaration (e.g. target PE ID, best-case/worse-case execution time, deadline etc.).

Each time a new periodic task object has been created, a pointer to the object is pushed onto the local task pointer vector, `local_task_list`. The pointers are being used, when configuring the tasks and when deleting the task objects in the module destructor. A pointer is also being provided to the dependency controller, by calling the method `push_back_task_ptr` with the pointer as argument. The dependency controller uses this pointer to access a task method for unblocking a blocked task (see also section 11.7, page 128). Finally, the encoded task ID[5] is stored in a local task ID vector, `taskID` and provided to the synchronizer module, by calling the method, `push_taskID` with the task ID as argument. The local task ID list is being used during task configuration, while the synchronizer uses the ID when accessing the global dependency database (see also section 11.1.3, page 90).

Inter-task dependency configuration

After a PE module has been constructed, any assigned tasks having preceding inter-task dependency must be configured to generated a `SOC_TRANSFER` message, when execution completes. This is required, since the default completion message is `FINISHED`. During inter-task dependency configuration, a task is being provided with information about the ID of all the non-local tasks (to which it has preceding dependencies to) as well as the address ranges of the PE's to which these have been mapped to. This information will be included in the `SOC_TRANSFER` message.

Inter-task dependency configuration is initiated by calling the PE module method, `configure_tasks` with a pointer to the parser module as argument. An algorithm scans the dependency database, available from the parser, and checks if any of the local tasks have preceding inter-task dependency. This is simply done

the module declaration, but requires some more computation steps

[5]see section 6.4.1, page 35

by scanning all rows using a fixed column ID[6], which maps to a local task ID, fetched from the local task ID list, `taskID`. Getting an entry from the dependency database is done by calling the parser method, `get_relation` with a row and column index as argument. It returns true or false, were true indicates a marked entry (dependency). Whenever an entry is marked, the algorithm checks if the row ID is associated with one of the other local task ID's from the task ID list, `taskID`. If so, the dependency is local (intra-dependency) and nothing is done. Otherwise an inter-task dependency exists and the task declaration information, associated with the non-local task ID, is found by scanning the task declaration list available from the parser module. An entry from the list is fetched by calling the parser method, `get_task_from_list` with an index as argument. The method returns a struct containing the task declaration. When the task declaration for the non-local task has been found, the PE ID, to which the task has been mapped to, is extracted and the associated address information is obtained by calling the parser method `get_address`, with the PE ID as argument. Afterward the pointer to the task to configure is found in the local task pointer list, `local_task_list` and the address information and the non-local task ID is forwarded to the task by calling the task method, `push_soc_comm_nfo`, with this information as argument. Finally, the subtask is configured to issue the `SOC_TRANSFER` message, by calling the method, `set_soc_transfer` with true as argument.

The algorithm completes execution, when all local tasks have been checked for inter-task dependency.

11.3 Parser

The parser module supports parsing of a very simple script language, used for describing PE module behavior, task declarations and dependencies etc. It accepts a configuration file as input, parses this and provides access to the different declared data, through a dedicated group of public methods. For a general description of the syntax form, and the way it is being used in this framework, please consult chapter 8, page 45. Currently seven declaration types are supported.

11.3.1 Parsing methodology

The simplicity of the syntax has also lead to a relative simple implementation of the module. The actual parsing methodology is *event based* in the sense that the configuration file is scanned char by char and processed "on-the-fly". That is, data declarations are stored in databases, as they are being detected during scanning. The database selection is relative to the type of declaration.

[6]See also section 5.1.1, page 24 for a description of the dependency database.

11.3.2 Error checking

The parser implements two types of error checking: (1) a lexical grammar and syntax check and (2) a declaration post check. The lexical grammar check is performed during the scanning, as a natural consequence of the event based parsing. If an unknown or illegal declaration is detected, the parsing terminates with an associated error message. The declaration post check follows a successful scanning; that is when all declarations have been successfully stored in the different databases. The post check consists of checking some of the declarations up against some predefined constrains. An example is the checking of SoC communication address overlap between two PE's. The different types of post check will be described later.

11.3.3 Parsing flow

The parsing flow is shown in figure 11.6. At first a parser object must be created. Provided arguments to the constructor are the expected number of arguments associated with a task (`arg1`) and resource requirement (`arg2`) declaration respectively. The figures are being used for reference, during the syntax check.

Parsing starts by calling the method, `start_parsing` with the configuration filename as argument. This starts the configuration file scanning followed by the declaration post check and processing. If any error occurs, the parsing terminates with an error message and `start_parsing` returns false. If parsing is successful it returns true, and afterward the different databases, containing the declaration data, can be accessed through different public method.

11.3.4 Configuration file scanning

Configuration file read approach

Reading the configuration file is done by fetching one char at a time. Fetching a char is done by calling the private method, `_nextch`, which returns the next char from the file. The char is stored in the private variable, `ch` always containing the current char and accessible in all private methods. For multi-char declarations like names and digits, a string will be constructed. Figure 11.7A and 11.7B are examples of code sequences, used for *name* (e.g. declaration-type mnemonic or task name) and *digit* string construction respectively. The foundation is based on pointer operations, since it is fast and simple. The string construction stops whenever a fetched char is not of a certain type (e.g. digit for digit string construction), or if the string length exceeds a certain limit. Afterward the string can be processed as required. For the digit string construction example, this consists of converting the string to an unsigned integer; also done using pointer operations. The examples illustrate the general approach used for string constructions in the parser. String construction and char fetching is done in the different private methods, associated with the configuration file scanning.

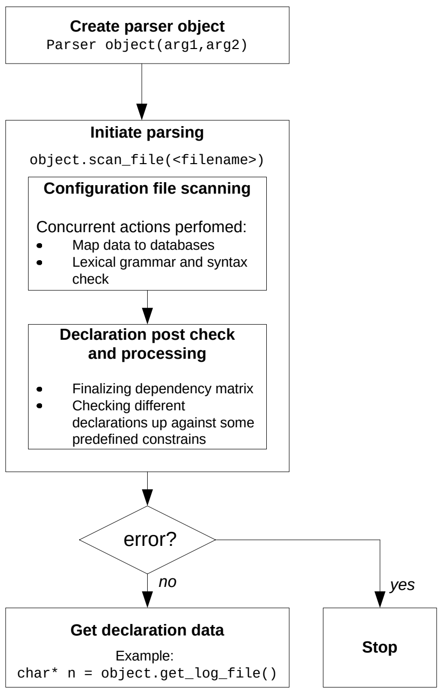

Create parser object
Parser object(arg1,arg2)
Initiate parsing
object.scan_file(<filename>)
Configuration file scanning
Concurrent actions perfomed:
Map data to databases
Lexical grammar and syntax check
Declaration post check and processing
Finalizing dependency matrix
Checking different declarations up against some predefined constrains
error?
no
yes
Get declaration data
Example:
char* n = object.get_log_file()
Stop

Figure 11.6: Parsing flow.

The scanning control algorithm

Figure 11.8 shows a *simplified* flowchart of the scanning control algorithm, implemented in `scan_file`. First action performed consists of opening the configuration file for reading, which is done by calling the private method, `_openfile` with a pointer to the filename as argument. Afterward starts the actual parsing, which continues until an error occurs (indicated by the `terminate` flag) or end-of-file (EOF) has been reached. The scanning can be divided into a *declaration-type mnemonic scan* and *declaration parameter scan*, executed in the mentioned order.

The *declaration-type mnemonic scan* serves to determine the type of declaration, by scanning and evaluating the declaration-type mnemonic. Remember from chapter 8, page 45 that a declaration always must start with a declaration-type mnemonic followed by the actual declaration. The declaration-type mnemonic scan is managed by the macro method, `_scan_symbol`. The method requires a symbol table as input, scans the configuration file and returns a pointer to a char, containing the symbol name, if the scanned symbol were found in the symbol table (otherwise it returns null). The symbol table to be provided is a `vector`, containing symbol structs (`sym_DB_type`). A symbol struct consists of a *symbol name* and a *declaration counter*, which is being incremented in `_scan_symbol` each time a symbol detection is declared. The counter can be evaluated later (declaration post check), to check how many times a certain symbol has been declared. The symbol table for the declaration-type mnemonic is `sym_declarationID`. It

(A) Name string construction

```
 char *p;                  // pointer to a char
 static char buffer[64];   // a buffer used for holding the string
 unsigned int N;           // a string length counter

 p = buffer;               // point at the start of the buffer
 N = 0;                    // initialize string length counter

// construct name string
 while((_letter() | _digit() | ch=='_') && N<63) {
   *p++ = ch;              // copy char copy into buffer
   N++;                    // increment string length counter
   ch = _nextch();         // and get next char from configuration file
 }

 *p++ = ‘\0’;              // terminate string with null
  p = buffer;              // and point at the start of the buffer


 // ...do something using the name string, accessed from p
```

(B) Unsigned integer string construction and conversion

```
 char *p;                    // pointer to a char
 static char buffer[64];     // a buffer used for holding the string
 unsigned int N;             // a string length counter

 _remove_leading_zeros();    // A macro removing leading zeros, if any

// construct unsigned integer string
 while(_digit() && N < 9) {
    *p++ = ch;               // copy char copy into buffer
    N++;                     // increment string length counter
    ch = _nextch();          // and get next char from configuration file
 }

 *p++ = ‘\0’;                // terminate string with null
  p = buffer;                // and point at the start of the buffer

// convert to unsigned integer
 if(N<9) {
   while(*p)
     value += ((unsigned int)(*p++)-48)*_pow(10,(N--)-1);
 }

// ...do something using the value
```

Figure 11.7: C++ code examples for name and digit string construction.

contains seven entries for `screen_dump`, `log_file`, `vcd_file`, `sub_task_map`, `dependency_map`, `ee_deadline` and `module` respectively and is created in the constructor.

The *declaration parameter scan* serves to scan the actual declaration data and store this in an associated database. For each declaration type, a dedicated method has been implemented, since the declaration syntax in general differs from declaration type to declaration type. Method selection is based on the declaration-type mnemonic (symbol name), returned from `_scan_symbol`. For an example, if the returned declaration-type mnemonic is `sub_task_map` the method, `_scan_task` is called, starting the task declaration parameter scanning. All methods return a status flag, indicating successful/unsuccessful scanning. False indicates successful scanning, while true indicates an error. Error messages, related to illegal syntax is generated inside the different methods.

Comments, space, tab and newline are removed at the different stages in scanning, by calling the private method, `_remove_separators`. Also, a collection of handy macro functions has been implemented. For an example, checking if the current fetched char (stored in `ch`) is a letter or digit can be done by calling the

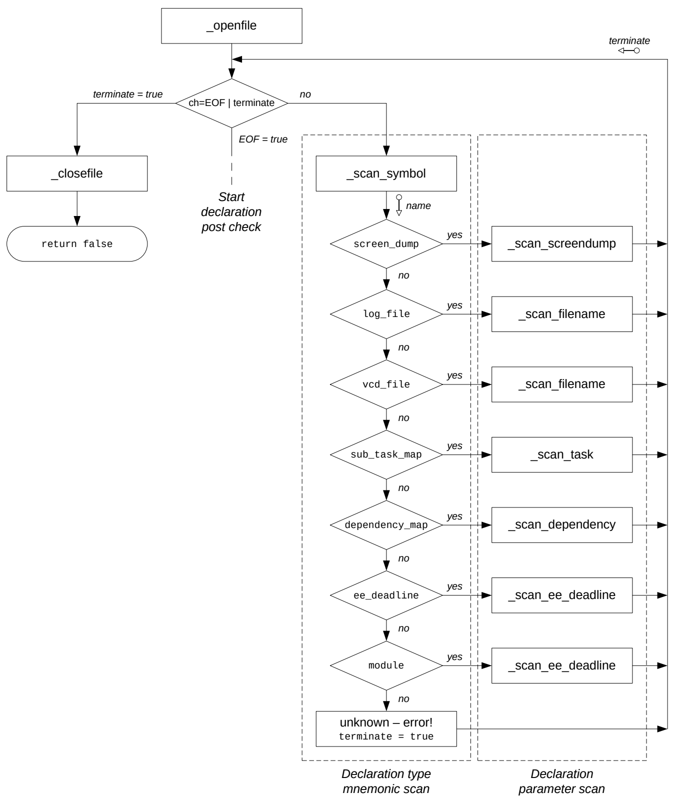


Figure 11.8: Simplified flowchart for configuration file scanning.

method _`letter` or _`digit`, returning a boolean, indicating yes or no.

11.3.5 Declaration post check and processing

Declaration post check and processing follows the configuration file scanning, if no grammar or syntax errors were found. The main operation performed is checking some of the declarations up against a set of predefined constrains. Besides error checking, the dependency matrix (defining the dependency database) is also finalized in this stage; that is expanded to include dependencies between subtasks as well. Figure 11.9 shows the simplified flowchart for the declaration post check and processing.

Declaration post check

The different types of post checks are implemented in the methods, seen on figure 11.9. They operate on the data available from the databases. All methods return a status flag, where true indicates an error. In case of error and warnings, the methods will assert a corresponding message. The behavior of the different methods are described briefly below.

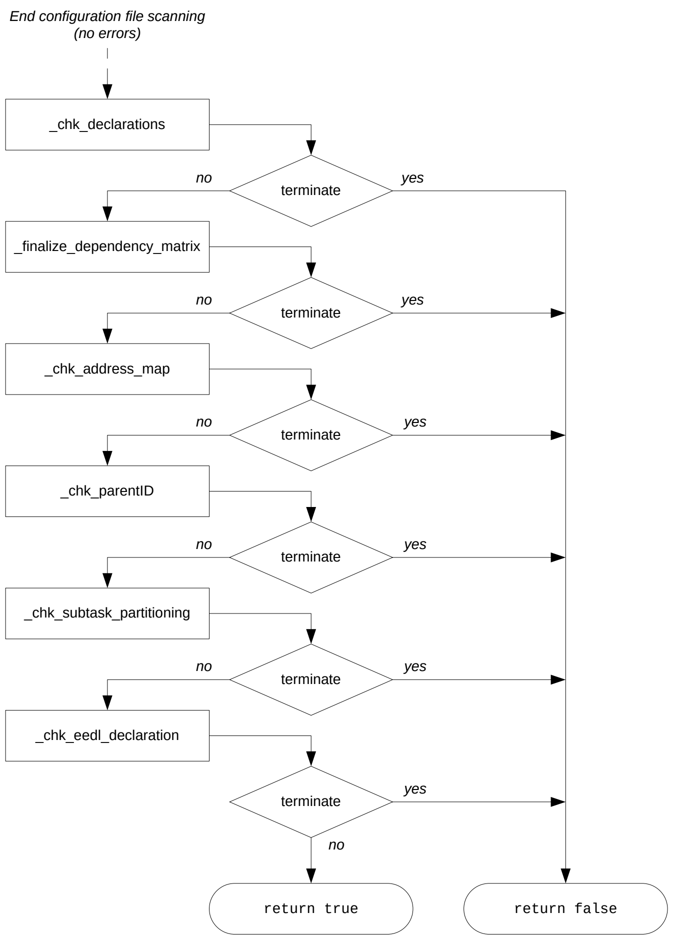


Figure 11.9: Simplified flowchart for the declaration post check and processing.

- `_chk_declarations`. Checks if all mandatory declarations has been declared in the configuration file (see also table 8.1, page 46). This is simply done by evaluating the associated declaration counter in the symbol list, `sym_declarationID` containing the declaration type mnemonics.
- `_chk_address_map`. Checks for illegal address declaration in the `module` declarations (e.g. address overlap between two PE's). A check will only be done for `module` declarations, containing `"peID"` and `"address"` declarations.
- `_chk_groupID`. Checks that none of the end-to-end tasks have a group ID equal to zero, since this is not allowed by the scheduler.
- `_chk_subtask_partitioning`. Checks for illegal subtask partitioning issuing read-response transfer. See also the set of rules, defined in section 6.3.1, page 33.

- `_chk_eedl_declaration`. Checks for missing or non-used end-to-end deadline declarations for end-to-end task consisting multiple and one subtask respectively.

Dependency matrix finalizing

The dependency database to be provided to the dependency controller must be refined to include dependencies between subtasks as well. However, the approach used for dependency declaration in the configuration file, is only to define a database expressing the dependencies between end-to-end task groups, since a group of subtasks, belonging to the same end-to-end task, always will be connected in a chain[7] [20]. The motivation has been to keep the dependency database declaration as simple as possible and mask away declarations, which can be assigned automatically. Thus the parser has to extend the declared dependency database also to include dependencies between subtasks.

Dependency database extension (or finalizing) is managed by the private method, `_finalize_dependency_matrix`. The algorithm implemented simply looks for all end-to-end tasks, consisting of multiple subtasks, and then creates a new database including these dependencies as well. However, addressing an entry in the database is a bit more complicated, since a task is identified by a group ID and a subtask ID. Dealing with this is done by creating a look up table (`mapping_nfo`), containing a row/column *index offset* for each group ID, indicating the row/column index associated with first subtask for a certain group ID. Thus when addressing an entry for a particular subtask, the *absolute* row/column index to use is found by fetching the index offset, from the look up table, using the group ID as argument, and then add the subtask ID.

Figure 11.10 shows an example of a finalized dependency database for three end-to-end task groups, T_1, T_2 and T_3 all having multiple subtasks. A dependency exists between T_1 and T_3. Figure 11.10A shows a section of the configuration file, containing task and dependency declarations while figure 11.10B shows the finalized database. For example, the absolute row/column index associated with $\tau_{3,2}$ is 7, when addressing the finalized database, since the index offset is 5[8] for group ID 3 and the subtask ID is 2.

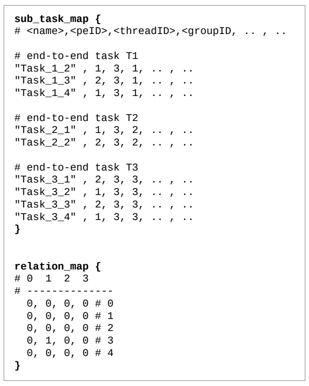
All information related to the dependency database is encapsulated in a struct (`relation_matrix_type`), containing parameters such as the original and expanded dependency database and the index offset look-up table.

11.3.6 Database description and access

Please refer to appendix A, page 161 for a brief description of the database types associated with the different declarations, as well as the method available for access.

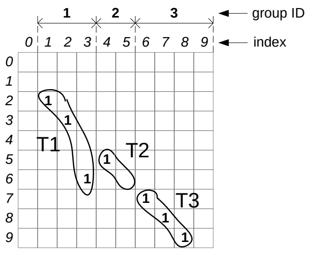
[7]see also section section6.3.1, page 33

[8]The index offset is not 6, since the first subtask ID always is 1



```
sub_task_map {
# <name>,<peID>,<threadID>,<groupID, .. , ..

# end-to-end task T1
"Task_1_2" , 1, 3, 1, .. , ..
"Task_1_3" , 2, 3, 1, .. , ..
"Task_1_4" , 1, 3, 1, .. , ..

# end-to-end task T2
"Task_2_1" , 1, 3, 2, .. , ..
"Task_2_2" , 2, 3, 2, .. , ..

# end-to-end task T3
"Task_3_1" , 2, 3, 3, .. , ..
"Task_3_2" , 1, 3, 3, .. , ..
"Task_3_3" , 2, 3, 3, .. , ..
"Task_3_4" , 1, 3, 3, .. , ..
}


relation_map {
# 0  1  2  3
# -------------
  0, 0, 0, 0 # 0
  0, 0, 0, 0 # 1
  0, 0, 0, 0 # 2
  0, 1, 0, 0 # 3
  0, 0, 0, 0 # 4
}
```

(A) Section of the configuration with task declarations and end-to-end dependency database.



(B) Finalized dependency database, now including dependencies between subtasks.

Figure 11.10: Example of a finalized dependency database

11.3.7 Maintenance

Adding new declaration types to the parser is very simple[9], due to the emphasis on modularity. See also figure 11.8, page 103 showing the scanning control algorithm. Following steps are required.

- Define new declaration-type mnemonic and add this to the declaration type mnemonic list, `sym_declarationID` in the constructor.
- Add new if-statement in `scan_file` for this declaration-type mnemonic.
- Define new database type and create a new declaration parameter scan method, managing the actual scanning of the declarations. Using the implemented macro method makes this easy.
- Define new methods for database access.

However, for most situations the `module` declaration type will probably be sufficient, since it allows clustering of declarations, each identified by a name and one or multiple values. See also how the `module` declaration is being used in this framework in section 8.2.1, page 46.

11.4 IO task-IO device communication link

This section presents the communication link between the IO task and IO device. Understanding this is essential to be able to add other types of IO tasks or IO device models to the abstract PE model.

[9]As long as the declaration type follows the same syntax rules, used in the current implemented declarations; that is a *declaration-type mnemonic* followed by the *declaration parameters*

11.4.1 The link

The communication link is based on the `sc_link_mp` model, from the SystemC master/slave library. The actual communication is based on high-level struct messages (`io_message_type`), similar to the approach used in the RTOS communication link. The different entries in the message struct are described in table 11.4.

Type	Name	Description
`unsigned int`	`comm`	Action identifier.
`unsigned int`	`type`	Action type (e.g. inter-processor communication transfer type).
`unsigned int`	`threadID`	A thread ID associated with the SoC communication.
`unsigned int`	`dataUnits`	Data transfer size associated with the SoC communication.
`deque<unsigned int>*`	`dataQ`	Pointer to a buffer containing data.
`deque<unsigned int>*`	`addrQ`	Pointer to a buffer containing addresses.
`char*`	`text`	A text describing the message. May be used for monitoring.

Table 11.4: IO task-device message struct, `io_message_type`.

11.4.2 The communication approach

The message communication between the IO task and IO device can be considered as high-level interrupt messages, where the actual inter-processor communication data are provided *indirectly* and not asserted onto the communication link. For an example, when the IO task is to start a new inter-processor communication event, it sends *only one* message to the IO device. This message contains all information required by the IO device to start the transfer; that is request type (read, write or response) identified by `type`, the data transfer size, identified by `dataUnits` etc. The data associated is provided *indirectly*, through pointers to deque objects, from where the IO device must fetch the addresses and/or data. One advantage of the usage of address and data deque objects is that they serve as buffers for burst transfer. It means the IO task eventually could push new data and addresses onto the deques, concurrent[10] with addresses and data being fetched (pop) by the IO device. See also the example in figure 11.11. In conjunction to this, `dataUnits` serves as a reference for the data transfer size. Thus, in case the buffers become empty, but not all data have been transmitted, the IO device could either just wait for data to become ready in the buffers (by evaluating, in each clock cycle, if a deque is empty, e.g. `addrQ->empty()`) or go into a sleep mode. Before going into sleep mode it should send a message to the IO task, notifying that it must wake-up the IO device, whenever new data are ready in the buffers. Sleep-mode means in this context that the IO device does not have to evaluate if a deque is empty, which otherwise could lead to performance degradation, simulation wise.

[10] In the same clock cycle

Wake-up is done by sending a new interrupt message to the IO device, similar to the initial SoC communication request message.

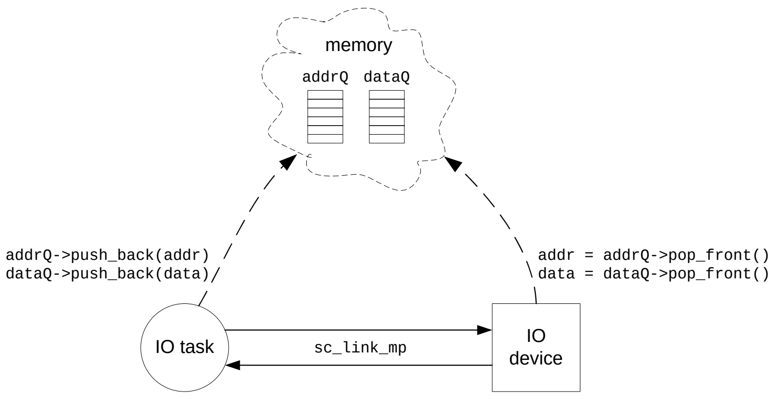


Figure 11.11: Illustration of the approach for indirect addressing. `addrQ` and `dataQ` illustrates the deque objects in memory, accessible from both the IO task and IO device.

The example is also applicable when the IO device is being the interrupt message initiator (when data is being received from the SoC communication interface). In this case, the address and data deque objects are located in the IO device.

The main motivation for the communication approach, described above, is the performance improvement obtained by providing data *indirectly*, compared to using the channel for *direct* data transfer. In case of *direct* data transfer, multiple messages must be send for a burst transfer, while only one message is required for the *indirect* approach (since pointers to the deque objects, where to fetch addresses and data, only have to be provided once).

11.5 IO task

The IO task models an IO device driver and implements the protocol, described in chapter 6, page 29. It serves as an inter-task dependency synchronization message encoder/decoder between the RTOS and the IO device. The IO task can handle multi-threaded SoC communication; a functionality implemented to support the multi-threaded OCP IO device model. It also supports preemption during fetching of response and write data, received from the SoC communication interface. However, IO task prioritizing has not been covered in this framework, meaning it always has the highest priority. Thus preemption will never occur.

The IO task interfaces to the IO device as well as the RTOS model (see figure 11.4, page 96), using the `sc_link_mp` model. It incorporates a master and slave port in the interface to the RTOS and IO device for bidirectional communication.

Figure 11.12 shows a simplified block diagram of the IO task, defined by the `ioTask` class. Oval figures indicate thread processes (`SC_THREAD` or `SC_SLAVE`) while rectangular figures indicate normal C++ methods. A name associated with a connection to process (e.g. `request_start`) indicates the name of an `sc_event` object, used for triggering the process execution.

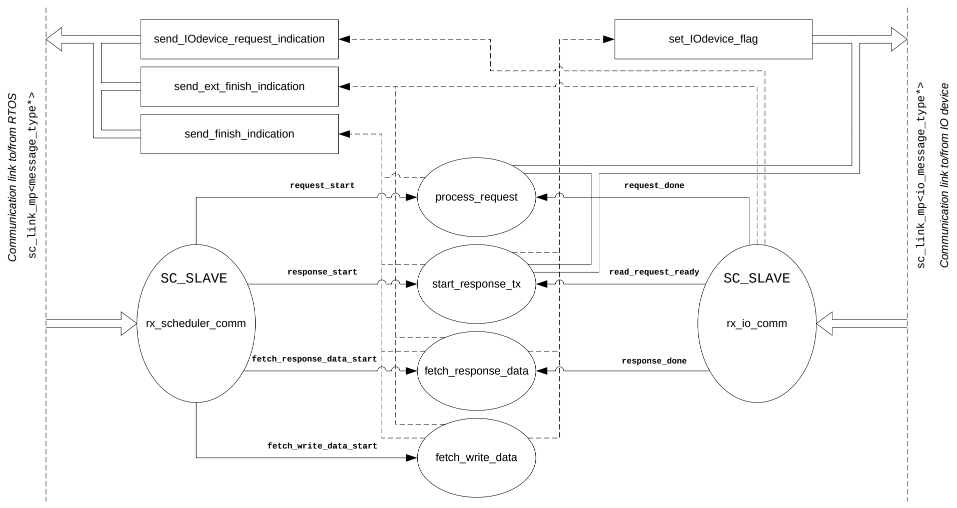


Figure 11.12: Simplified IO task block diagram.

The IO task has been implemented without any use of a state-machine. Instead the actual protocol is implemented in the four main thread processes, seen on figure 11.12. Other protocol types are easily implemented, simply by changing the behavior of the processes.

- `process_request` manages the issuing of write and read requests in conjunction to the methods, `prepare_wr_request` and `prepare_rd_request` (not shown in figure 11.12).
- `start_response_tx` manages the response transmission for a previously received read request.
- `fetch_response_data` manages the response data received for previously initiated read request.
- `fetch_write_data` manages the write data for a received write request.

The next sections present a more detailed behavior description of the different processes and also how they interact together.

11.5.1 RTOS interface slave port

The processes, `rx_scheduler_comm` contains the slave port interfacing to the RTOS. It decodes the message received from the scheduler and determines the type of action to perform. For RUN or RESUME messages, it triggers a process, according to the action type, identified by the type field in the message. Process triggering is done by asserting an associated notification event. For an example, when the IO task is to transmit response data, the message received will be RUN with type set to RESP. Thus it will assert the notification event response_start, causing the process, start_response_tx to execute and the response phase to begin. For PREEMPT messages, a corresponding preemption flag will be set (fetch_response_data_preempt or fetch_write_data_preempt) causing the associated process (fetch_response_data or fetch_write_data), to terminate. The flag will be cleared when the associated RESUME message is received.

The different types of supported messages and their meaning is described in table 11.5. Non-supported messages will cause the IO task to assert an error message.

comm	type	Description
RUN	WR	Start write request.
	RD	Start read request.
	RESP	Start response (to a read)
	RD_RESP_READY	Start fetching the response data received.
	WR_DATA_READY	Start fetching the write data received.
PREEMPT	RD_RESP_READY	Preempt response data fetching.
	WR_DATA_READY	Preempt write data fetching.
RESUME	RD_RESP_READY	Resume fetching the response data received.
	WR_DATA_READY	Resume fetching the write data received.

Table 11.5: Supported messages types, received from the scheduler. comm and type refers to the declaration in the message. See also table 11.1, page 88

For simplicity, notations notation like, RUN@WR might be used in the following sections. This notation means a message where comm and type entries are equal to RUN and WR respectively.

11.5.2 IO device interface slave port

rx_io_comm decodes the messages received from the IO device. An IO device message either indicates the completion of a previously initiated inter-processor communication event or that some data have been received from the SoC communication interface. These are represented by FINISHED or READY messages respectively. The different types of messages supported and their meaning is described in table 11.6. Non-supported messages will cause the IO task to assert an error message.

When a FINISHED message is received, a notification event is asserted to the thread process previously initiated the transfer (for an example request_done when a request has been completed). This notification is expected by the process, causing it to do a certain operation (See description for process_request and

comm	type	Description
READY	RESP	A read request has been received.
	RD_RESP_READY	Response data, associated with a previously initiated read has been received and now ready to be fetched.
	WR_DATA_READY	A write request has been received and data is now ready to be fetched.
FINISHED	WR	The initiated write request has completed.
	RD	The initiated read request has completed.
	RESP	The response phase (to a previously received read request) has completed.

Table 11.6: Supported messages types, received from the IO device. `comm` and `type` refers to the declaration in the message. See also table 11.4, page 107.

`start_response_tx` in section 11.5.3 and 11.5.4 respectively).

When a `READY` message is received, `rx_io_comm` captures the message information and store this in a database. The selected database and the information to store depends on the type being `RESP`, `RD_RESP_READY` or `WR_DATA_READY`. After a database has been updated a similar `READY` message is issued to the scheduler, indicating the IO task is ready to process the received IO device request.

The databases are used when the IO task processes the IO device requests. They hold information such as pointer to the address and data deque objects and a counter for keeping track of the amount of data received. This information is accessed and used in the different thread processes, executing the IO device requests. Table 11.7 shows the available database types (structs). The actual databases have been implemented as arrays of structs, to support threaded IO communication. Thus addressing a particular struct is done using the thread ID, associated with the IO device request (will be forwarded in the `READY` message issued to the scheduler).

11.5.3 Request transmission

The process, `process_request` manages the issuing of write and read request messages to the IO device. It is being triggered by the `request_start` event, when `rx_scheduler_comm` receives a `RUN@WR` or `RUN@RD` message from the scheduler, indicating a write or read request respectively. The process *does not* create messages but fetches the messages from a request queue, `requestQ` containing pointers to request messages. The actual messages are created and pushed onto the request queue in the methods `prepare_wr_request` and `prepare_rd_request` (not shown on figure 11.12). These methods are called before issuing the actual notification event. `prepare_wr_request` creates and prepares the address and data deque objects for write request, while `prepare_rd_request` prepares the address deque for read requests. For broadcasting of write data to multiple PE's, `prepare_wr_request` will creates multiple requests messages.

Database for **response data**. Keeps track of the response data deque pointer and a counter for monitoring the amount of responses received. The counter is updated when the IO task issues the read request, and decremented when response data is being fetched.

Database is updated when receiving a `READY@RESP` message from the IO device .
(only `respDataQ` is updated).

Struct name	:	`respDataQ_DB_type`
Database name	:	`respDataQ_DB`

Type	Name	Description
`unsigned int`	`respDataCounter`	A reference counter used for keeping track of the amount responses to received for a read request.
`deque<unsigned int>*`	`respDataQ`	Pointer to the deque, containing the response data.

Database for **write requests**. Keeps track of the address and data deque pointers and a counter for monitoring the amount of data received. The counter is initialized with the amount of data to receive, and decremented when write data is being fetched.

Database updated when receiving a `READY@WR_DATA_READY` message from the IO device.

Struct name	:	`writeDataQ_DB_type`
Database name	:	`writeDataQ_DB`

Type	Name	Description
`unsigned int`	`writeDataCounter`	A reference counter used for keeping track of the amount data received.
`deque<unsigned int>*`	`dataQ`	Pointer to the deque, containing the write data.
`deque<unsigned int>*`	`addrQ`	Pointer to the deque, containing the write addresses.

Database for **read requests**. Keeps track of the address deque pointer and a counter used for monitoring the number of responses transmitted. The counter is initialized with the amount of responses to transmit, and decremented whenever a response is transmitted

Database is updated when receiving a `READY@RD_RESP_READY` message from the IO device.

Struct name	:	`readRequestQ_DB_type`
Database name	:	`readRequestQ_DB`

Type	Name	Description
`unsigned int`	`requestCounter`	A reference counter used for keeping track of the amount responses to transmit.
`deque<unsigned int>*`	`addrQ`	Pointer to the deque, containing the read addresses.

Table 11.7: IO task database types.

After a request message has been issued to the IO device, `process_request` waits for a `request_done` event, from `rx_io_comm`. This event indicates that the request has been completed by the IO device and it may now accept a new request. If the request queue is not empty, the next request messages from the queue is processed in the same manner. `process_request` completes when the request queue becomes empty. Before the thread completes execution, it calls `send_finish_indication`, causing a `FINISHED` message to be issued to the scheduler, indicating the IO task execution has completed.

11.5.4 Response transmission

The process, `start_response_tx` manages the response to a previously received read request. It is being triggered by the `request_start` event, when the slave process, `rx_scheduler_comm` receives a `RUN@RESP`. This happens after the task, triggered by the read request, finishes execution. First operation performed is getting the address deque pointer and the data transfer size. This is done by accessing the database, `readRequestQ_DB` using the thread ID from the messages, received from the scheduler. This database entry has previously been updated in the slave process, `rx_io_comm` when it received the read request message from the IO device. In conjunction to this, the thread ID used for gaining access to the database is the same as the one associated with the read request.

Afterward, the process creates and prepares the response data deque. This is done simply by pushing the encoded ID of the task, triggering the response, onto the data deque. The number of responses created/pushed onto the data deque equals the number of addresses[11] fetched from the address deque. Also, the `requestCounter`, used for keeping track of the number of responses, is decremented correspondingly. It then sends a `RUN@RESP` message to the IO device, causing it to start the response phase for this particular thread. Afterward, it checks if all responses have been transmitted by evaluating if `requestCounter` has become zero. If not, it means that the initial read request was a non-single burst request, and that all requests have not been received yet. Thus `start_response_tx` must wait for the remaining request to come before asserting any further responses. The process then goes into sleep-mode and informs the IO device that it must sends a new `RUN@RESP` message (wake-up message) to the IO task whenever it receives the next pending read request for this particular thread. This is done by calling the method `set_IOdevice_flag`, which issues a `READY@RD_REQ_NOTIFY` message to the IO device. Afterward the process waits for an event to be asserted on `start_response_tx`, which happens when `rx_io_comm` receives a `READY@RESP` message from the IO device for this thread. The event causes the process to wake up and the response phase to be resumed. In conjunction to this, the process sends the `RUN@RESP` message to the IO device again. When the response phase completes, it calls `send_finish_indication`, causing a `FINISHED` message to be issued to the scheduler, indicating the IO task execution has completed. It also calls the method `set_IOdevice_flag` again, for notify-

[11]The read addresses are not being used for anything in this IO task

ing the IO device that it must send a message when it receives a new read request on this thread.

11.5.5 Write data processing

The process, `fetch_write_data` manages the data coming from a received write request. It is being triggered by the `fetch_write_data_start` event, when `rx_scheduler_comm` receives a `RUN@WR_DATA_READY` from the scheduler. The message can be considered as an echo of the `READY@WR_DATA_READY` message, previously initiated by `rx_io_comm`, when it received the write request from the IO device. From the message it uses the thread ID for gaining access to the `writeDataQ_DB` database entry, containing the data counter, `writeDataCounter` and the pointers to the address and data deque objects, associated with the write request. The data fetching is done, by popping an element off the address and data deques and decrements the data counter in each clock cycle. When the deques becomes empty, it checks if all data has been received by evaluating if the data counter is zero. If the data counter is nonzero, the IO task execution suspends, until new write data is ready again. This leaves room for other task to execute meantime. Before suspending, it notifies the IO device that a new `RUN@WR_DATA_READY` message must be send, when new data are received for this particular thread. This is done by calling `set_IOdevice_flag`, causing a `READY@WR_DATA_NOTIFY` message to be issued to the IO device. Afterward the process calls `send_finish_indication`, causing a `FINISHED` message to be issued to the scheduler, indicating the IO task execution has stopped/been suspended.

Resuming the write data fetching starts when `rx_scheduler_comm` receives a `RUN@WR_DATA_READY` for this thread again.

When the write request completes (i.e. data counter, `writeDataCounter` equals zero), any local tasks depending on the data can be released for execution. This is done by calling `send_ext_finish_indication` with the encoded ID of task[12], initiating the write request, as argument. The method issues a `FINISHED_EXT` message the synchronizer for this task. Before the process suspends and the IO task completes execution, it calls `set_IOdevice_flag` again followed by a call to `send_finish_indication`.

11.5.6 Response data processing

The process, `fetch_response_data` manages the fetching of response data, received for a previously initiated read request. It is being triggered by the `fetch_write_data_start` event, when `rx_scheduler_comm` receives a `RUN@RD_RESP_READY`. The behavior is similar to the process `fetch_write_data`, but it operates on the database, `respDataQ_DB`. See also section 11.5.5, describing `fetch_write_data`.

[12]extracted from the write address, by subtracting the PE base address

11.6 IO device

The IO device models the physical IO hardware port and manages the SoC communication protocol. The modular approach makes it easy to implement support for various types of SoC communication protocols, since the interface between the IO task and the IO device is well defined (See also section 11.4.2). In the current version of the framework, two IO device models have been implemented, supporting the OCP 2.0 protocol at TL1 and TL0. Common features for the models are:

- Configurable (signal-wise), relative to the OCP channel they connect to.
- Support for multi-threaded OCP interface. Further, the OCP slave supports out-of-order thread execution.
- Write data (OCP Slave) and response data (OCP master) buffers for each thread, with configurable size.

The next sections present the implementation of the OCP 2.0 TL1 and TL0 IO device models.

11.6.1 OCP TL1

The foundation of the OCP2.0 TL1 IO device model is based on the SystemC OCP Transaction Level Communication Channel, available from [2]. TL1 is also known as the transfer layer abstraction and provides cycle true simulation, but is faster than RTL simulation [18]. The communication is done through the *clocked* `OCP_TL1_Channel` object, supporting *single command operations* for initiating OCP specific transactions like request, response and data handshake. This channel is inherited from the generic `TL_Channel`, which actually supports transfer of any kind of data (also non-OCP compliant). Please consult [18] for more information.

The `TL1_IO` class defines the OCP 2.0 TL1 IO device model. It is composed of an IO task message router (`Router`), a master (`TL1_Master`) and slave (`TL1_Slave`) to support request initiating and receiving respectively. The router simply ensures correct routing of messages coming from the IO task (e.g. messages targeting the master is only being received by the master). This is done by evaluating the `type` declaration in the message.A simplified UML composition diagram is shown in figure 11.13.

Connecting the master and slave to an OCP channel is done after channel and module construction, by calling the methods `connect_OCP_Master` and `connect_OCP_Slave` respectively, with a pointer to the TL1 channels.

11.6.2 Supported OCP TL1 configurations

The OCP TL1 device model supports the OCP configuration, listed in table 11.8. The model automatically configures itself relative to the channel it connects to.

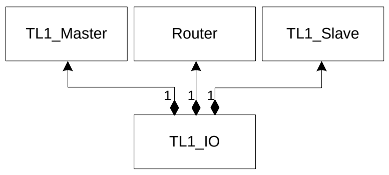


Figure 11.13: Simplified OCP2.0 TL1 IO device model UML composition diagram.

This is done in the `end_of_elaboration` method in the master and slave and happens after the module has been created and connected to the TL1 channel, but before the simulation starts. Channel parameters are fetched from the OCP channel by calling the `OCP_TL1_Channel` method, `GetParamCl` returning a `ParamCl` object, containing the actual parameters.

Configuring an OCP TL1 channel

The OCP channel can be configured using the *OCP configuration parameters*. The configuration is done after a channel has been created, by calling the `OCP_TL1_Channel` channel method, `setConfiguration` with a configuration parameter map object as argument. The configuration parameter map object is a STL C++ MAP (available from the `<map>` library) and must contain the different configuration parameter names and their associated types and values [18].

In this framework, the generation of the configuration parameter map is done by reading contents from a configuration parameter file, before a simulation starts. The filename must be provided as argument and the map will be generated in the top-level module. Table 11.9 shows an example a section of such a file.

11.6.3 OCP TL1 Master

Figure 11.15 shows a simplified block diagram of the OCP TL1 Master, defined by the `TL1_Master` class. It interfaces to the IO device task through `sc_link_mp` and the TL1 channel. Oval figures indicate thread processes (`SC_THREAD` or `SC_SLAVE`) while rectangular figures indicate normal C++ methods. A name associated with a connection to process (e.g. `m_RequestEvent`) indicates the name of an `sc_event` object, used for triggering the process execution. The dotted box shows response data buffers for the different threads.

The OCP Master does not use a state machine for protocol handling. The implementation is more high-level and uses dedicated clocked thread processes instead. The threads related this are:

- `requestThreadProcess` manages the issuing and protocol handling related to read and write requests.

Signal	Configuration Parameter	Comment
Request group		
MCmd		Always (1)
MAddr	`addr`	Required
MData	`mdata`	Required
MReqLast	`reqlast`	Optional
SCmdAccept	`cmdaccept`	Optional
Response		
MRespAccept	`respaccept`	Optional (2)
SData	`sdata`	Required
SResp	`resp`	Required (4)
Burst extension		
MBurstSeq	`burstseq`	Optional (5)
MBurstSingleReq	`burstsinglereq`	Optional
MBurstPrecise	`burstprecise`	Required if MBurstSeq
MBurstLength	`burstlength`	Required if MBurstSeq
Data handshake	`datahandshake`	Optional (7)
MDataThreadID	`threads>1`	Optional
MDataLast	`datalast`	Optional
SDataAccept	`dataaccept`	Optional (3)
Thread extension		
MThreadID	`threads>1`	
SThreadID	`threads>1`	
MThreadBusy	`mthreadbusy`	Optional (2)
SThreadBusy	`sthreadbusy`	Optional
	`sthreadbusy_exact`	Optional (6)
SDataThreadBusy	`sdatathreadbusy`	Optional (3)
	`sdatathreadbusy_exact`	Optional (6)

(1) WR and RD are required to be supported by the OCP channel. Other types currently not supported.
(2) Either MRespAccept or MThreadBusy must be used (in conjunction with responses). Disabling both parameters are not allowed.
(3) Either SDataAccept or SDataThreadBusy must be used when data handshaking is enabled.
(4) Write (WR) response enable is optional using `writeresp_enable` parameter.
(5) The master currently only support assertion of UNKN.
(6) Usage follows the protocol semantics, defined in [17], pp.49.
(7) If data handshake is disabled, all signals related to datahandshake becomes non-applicable.

Table 11.8: OCP configuration supported by the OCP2.0 TL1 IO device.

- `clearThreadBitProcess` monitors response data buffers, if a buffer becomes full for a particular thread. It clears the thread bit (MThreadBusy) whenever buffer space becomes available again. Only applicable if MThreadBusy is a part of channel.
- `responseThreadProcess` manages the protocol related to response data receiving and the buffering of response data. It also controls the issuing of notification messages to the IO task, when new response data become ready

```
addr i:1
addr_wdth i:16
addrspace i:0
burstlength i:1
burstlength_wdth i:16
burstprecise i:1
burstseq i:1
burstseq_unkn_enable i:1
burstsinglereq i:1
cmdaccept i:1
dataaccept i:1
datahandshake i:1
datalast i:1
 .
 .
```

Table 11.9: A section of an OCP configuration parameter file.

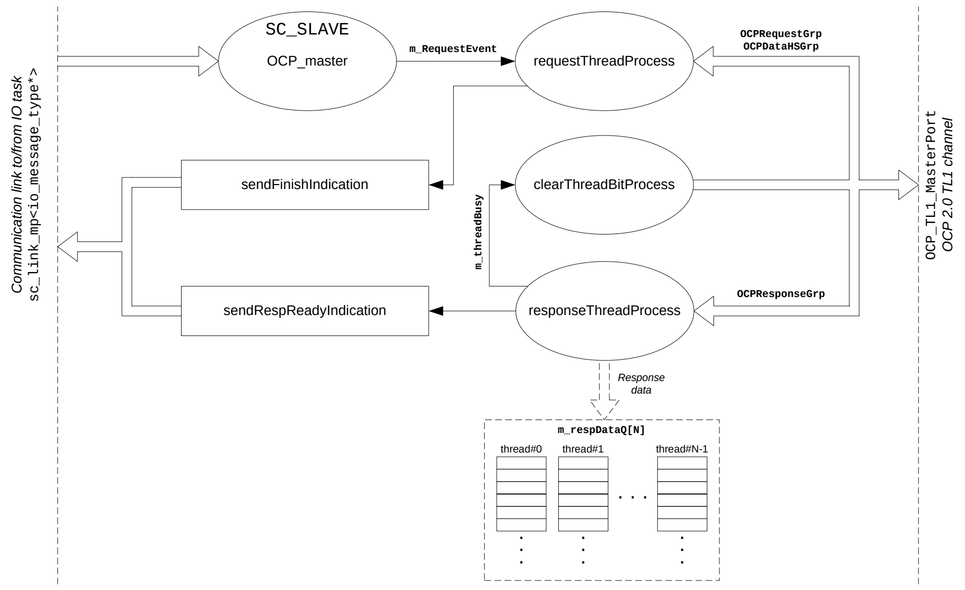


Figure 11.14: Simplified block digram for OCP2.0 TL1 Master.

in a buffer.

The next sections present a more detailed behavior description of the different processes and also how they interact together.

IO task interface slave port

The `SC_SLAVE` process, `OCP_slave` decodes the messages coming from the IO task and determine what action to perform. The message types supported are described in table 11.10. Non-supported messages will cause the TL1 master to assert an error message.

`comm`	`type`	Description
RUN	WR	Start a write request.
	RD	Start a read request.
SET_FLAG	RD_RESP_NOTIFY	Tells the master that it must issue a notification message, when read response data are being received on a particular thread.

Table 11.10: Supported IO task messages types. `comm` and `type` refers to the declaration in the message. See also table 11.4, page 107.

`RUN` messages causes a request to start by triggering the event, `m_RequestStart` while the `SET_FLAG` message causes the process to set an entry in the `m_respNotify` boolean table to true. `m_respNotify` serves as a response notification database, and accessed using the threadID. The purpose of the database will be described in section 11.6.3.

The master contains no buffers for request messages (should be managed by the IO task). Thus receiving a RUN message in the middle of a request phase will cause an error message to be asserted.

Request handling

A request phase starts when the process, `requestThreadProcess` gets triggered by the event, `m_RequestEvent`. Before starting the request, it checks if SThreadBusy must be used. If so, it checks if the thread bit is set for the thread associated with the request to initiate. If set, the slave cannot accept any new requests and the process waits until the thread bit becomes cleared, before asserting the request.

For write requests, the process selects an appropriate transaction method, based on the channel configuration. For an example, single request burst write will always be used, if data handshake and MBurstSingleReq is supported by the channel, and if the write request is associated with multiple data transfer.

The address and data used for the write request is fetched from the deque objects, created by the IO task. Accessing the objects are done through the pointers, `addrQ` and `dataQ` provided by the IO task request message (see also section 11.5.3, page 111). If the data/address deques becomes empty and not all data has been transmitted, the request/data handshake phase stalls until new data are available again. However, in the current implementation of the IO task, this scenario will never occur, since all data will be available when a request message is issued.

For a read request, the transaction method also depends on the channel configuration. If MBurstSingleReq is a part of channel, single request burst read will be used for a burst read requests. Read request addresses are fetched from the address deque, by gaining access to the object using the pointer, `addrQ`) provided by the IO task request message.

When a request phase completes, the process calls the method, `sendFinishIndication`, before completing. This method *reuses* the initial task request message created by the IO task, alter this to a `FINISHED` message and returns the pointer back to the IO task again, indicating that the request has been completed.

After the method, `sendFinishIndication` has completed executed the process suspends itself and waits for `OCP_master` to trigger a new request.

It must be emphasized that the current master implementation has following limitations:

- MCmd mnemonic supported are WR and RD.
- MBurstSeq will always be UNKN for burst requests.

However the implementation can easily be modified to support other types of MCmd and MBurstSeq.

Response handling

Receiving of response data are managed by the process, `responseThreadProcess`. When a valid response is being received (SResp = DVA) the response data are being pushed onto a response data deque, `m_respDataQ` selected relative to the response thread ID (SThreadID). In conjunction to this, `m_respDataQ` is implemented as an array of deques, with an array size equal to the number of threads supported by the channel (defined by the OCP configuration parameter, `threads`).

Afterward, the boolean response notification database, `m_respNotify` is checked to see if a response notification message must be issued to the IO task. This is done by gaining access to the database using the thread ID. If the entry is true, it means a notification must be send to the IO. Thus the process will call the method, `sendRespReadyIndication`, which creates and issues a `RD_RESP_READY` message to the IO task (see also table 11.6, page 111). This message also contains a pointer to the associated response data deque, containing the response data. After the method has been executed, the entry in `m_respNotify` is cleared, to ensures that no notification message will be send next time response data are received on this thread[13]. Thus only the response data will be pushed onto the response data deque.

If the size of the response data deque has reached the maximum buffer size, defined by `_MASTER_QUEUE_LIMIT`, the process will stall any responses coming afterward, for the particular thread. If MThreadBusy is a part of the channel, then this is done by asserting the associated thread bit. If MRespAccept is a part of channel

[13]This is not required since the IO task now has the pointer to the response data deque object and also know the data transfer size, associated with this thread.

instead, it will be de-asserted until buffer space becomes available again. When using MThreadBusy, the process will also trigger the event, `m_threadbusy`, causing the response data buffer size monitor, `clearThreadBitProcess` to executes. At each clock cycle, the process monitors the size of the buffers and clears a thread bit, when space becomes available in an associated previous full buffer. The process suspends itself whenever there is space left in all the buffers (i.e. when MThreadBusy equals zero).

11.6.4 OCP TL1 Slave

Figure 11.15 shows a simplified block diagram of the OCP TL1 Slave, defined by the `TL1_Slave` class. It interfaces to the IO device task through `sc_link_mp` and the TL1 channel. Oval figures indicate thread processes (`SC_THREAD` or `SC_SLAVE`) while rectangular figures indicate normal C++ methods. A name associated with a connection to process (e.g. `respQ_ready`) indicates the name of a `sc_event` object, used for triggering the process execution. The dotted boxes illustrates address and data buffers.

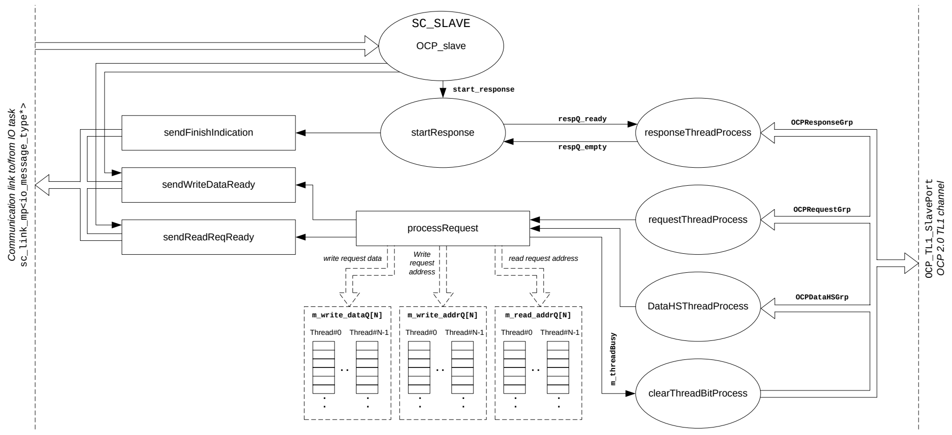


Figure 11.15: Simplified block digram for OCP2.0 TL1 Slave.

The OCP Slave does not use a state machine for protocol handling. The implementation is more high-level and uses dedicated clocked thread processes instead. The threads related this are:

- `startResponse` prepares response data to transmit for a previous received read request.
- `responseThreadProcess` transmit response data, prepared by `startResponse`, to the OCP channel and manages the protocol related to this.
- `requestThreadProcess` manages the received requests from the OCP channel.

- `dataHSThreadProcess` manages the protocol and all signals related to data handshake. Only applicable, if data handshake is a part of the channel (`datahandshake = 1`).
- `clearThreadBitProcess` monitors the write data buffers in case a buffer becomes full. It clears the thread bit (`SThreadBusy` or `SDataThreadBusy`) whenever buffer space becomes available again. Only applicable if SThreadBusy or SDataThreadBusy (data handshake) is a part of channel.

The next sections present a more detailed behavior description of the different processes and also how they interact together.

Data and address buffers

The slave incorporates buffers for each thread for buffering write addresses and data as well as read request addresses. Buffer size is controlled using the compiler statement, `_SLAVE_QUEUE_LIMIT`. The different buffer types are implemented as arrays of deque objects, where the array size equals the number of threads, supported by the OCP channel (defined by the OCP configuration parameter, `threads`). The deque objects, `m_write_addrQ` and `m_write_dataQ` are used for buffering addresses and data associated with a write request respectively, while `m_read_addrQ` is used for buffering addresses associated with a read request. Selecting a buffer for a particular thread is done using the thread ID.

Request notification database

The slave also implements a *request notification database* for write and read requests. The databases are linked to the address and data buffers and holds information about the burst lengths of pending requests available in a buffer for a particular thread. This information is being used when sending a request notification message to the IO task (the exact purpose will become clearer in the example following later).

The read and write request databases are implemented in `m_readReqReadyQ` and `m_writeReqReadyQ`, which are arrays of the struct type, `request_DB_type`, where the array size equals the number of threads supported by the OCP channel. The entries in the struct is described in table 11.11. Updating a database is done each time a request is being processed in the method, `processRequest`. This procedure is described in section 11.6.4 following later.

The following example tends to clarify the link between the buffers and the database as well as its usage with respect to notification message generation to the IO task: Assumed that three *different* burst write request has been buffered for thread#1[14]. These having a burst length of 21, 6 and 12 respectively and buffered

[14]This scenario would be possible if the slave were in a middle of a very long response phase, since request and response may happen concurrent. Thus the IO task cannot process the request before the response phase has completed

Type	Name	Description
`bool`	`notify`	Indicates if the slave must issue a notification message to the IO task when a request is received (required if true).
`unsigned int`	`counter`	Used to keep track of the number of requests/data received for a burst request.
`deque<unsigned int>`	`burstlength`	A FIFO buffer containing the burst length of pending requests currently stored in the associated buffer.
`deque<bool>`	`singlerequest`	A boolean, indicating if the request is a single request (true) or not (false). Only applicable for single request burst read.

Table 11.11: Request notification database struct, `request_DB_type`.

in the mentioned order. Here the `burstlength` FIFO buffer in the request notification data base for thread#1 (`m_writeReqReadyQ[1]`) will contain the burst length, also buffered in the mentioned order. This is illustrated in figure 11.16.

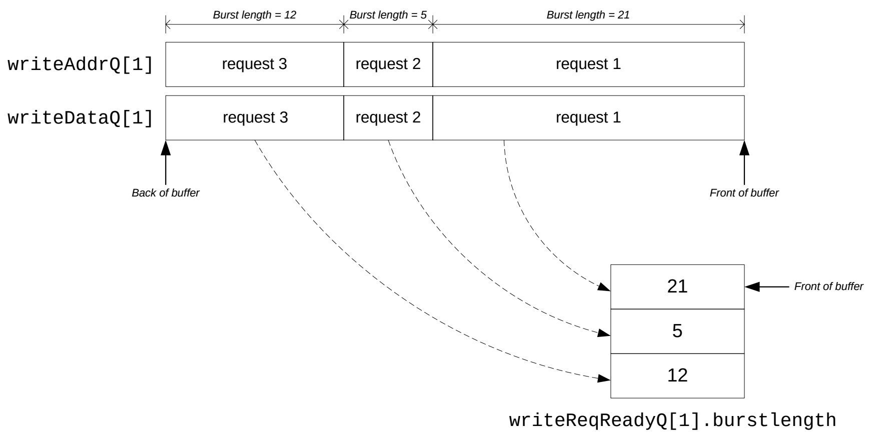


Figure 11.16: example

When the IO task sends a notification, that it is ready for receiving a write request again on thread#1, the burst length information available from the front of the `burstlength` FIFO will be fetched and included in the notification message issued to the IO task. Thus this message will be associated with the first burst write request, available from the buffer and having a burst length of 21. When this request has been processed, the next request, in front of the buffer is request 2, which is to be processed next etc.

IO task interface slave port

Messages from the IO task is decoded by the `SC_SLAVE` process, `OCP_slave`. Depending on the type of message, an action will be initiated by the process. The message types supported are described in table 11.12. Non-supported messages will cause the TL1 slave to assert an error message.

comm	type	Description
RUN	RESP	Start response phase to a previous received read request
SET_FLAG	WR_DATA_NOTIFY	Tells the slave that it must issue a notification message, when it receives a write request/write data on a particular thread
	RD_REQ_NOTIFY	Tells the slave that it must issue a notification message, when it receives a read request on a particular thread

Table 11.12: Supported IO task messages types. `comm` and `type` refers to the declaration in the message. See also table 11.4, page 107.

A `RUN` message causes the process to initiates a response transaction, by triggering the event, `start_response`. When a `SET_FLAG` messages is received, the IO task notifies the slave, that it must issue a message whenever there is a new request/data ready on a particular thread. At first, the process checks an associated request notification database (`m_readReqReadyQ` if message `type` = `RD_REQ_NOTIFY` or `m_writeReqReadyQ` if message `type` = `WR_DATA_NOTIFY`) to see if there is a pending request already available in the buffer. This is done by evaluating the `burstlength` FIFO buffer. If no pending request is available, the `notify` entry is set to true, causing a notification message to be issued, whenever a request is being received on this thread next time. However, if there is a pending request in the buffer, the bust length is popped from the `burstlength` FIFO and the method `sendWriteDataReady` or `sendReadReqReady` is called, depending on the `SET_FLAG` message `type` being `WR_DATA_NOTIFY` or `RD_REQ_NOTIFY` respectively.

The method, `sendWriteDataReady` will create and issue a `WR_DATA_READY` message to the IO task, indicating that a write request is ready, while the method `sendReadReqReady` will create and issue a `RESP` message to the IO task, indicating a read request is ready and a response is now expected. A `WR_DATA_READY` message also contains pointer to the associated write address and data buffers, while the `RESP` message contains a pointer to an associated read address buffer. In both scenarios the fetched burst length will be included (in `dataUnits`) as well, telling the IO task the data transfer size associated.

Request handling

Request receiving from the OCP channel is managed by the process, `requestThreadProcess`. This process operates on the `OCPRequestGrp` object, which contains all request group signals, also including MData, if data handshake is not a part of channel. When a request is received, the associated protocol handling will be done relative to the OCP channel configuration. Afterward the request is forwarded the method, `processRequest` which process the request.

For a write request, the method pushes the address and data onto the buffers associated with the request thread ID. Then the request notification database, `m_writeReqReadyQ` is updated, by decrementing the `counter` entry. This counter is used for keeping track of the amount of data received for a burst write. If the counter equals zero, when the database is being updated, it means that the request is new, since the data does not belonging to a previous burst write request. Thus the counter will be initialized to burst length-1. If the `notify` entry is false, the IO task is not ready to receive a write request ready notification, and the burst length for the new request will be pushed onto the `burstlength` FIFO instead. Otherwise the method calls `sendWriteDataReady` which issues a write request ready notification message to the IO task. If a write data buffer becomes full, and SThreadBusy or SDataThreadBusy is a part of the OCP channel, the method will trigger the event, `m_threadbusy`, causing the write data buffer size monitor, `clearThreadBitProcess` to executes. At each clock cycle, the process monitors the size of the buffers and clears a thread bit, when space becomes available in an associated previous full buffer. The process suspends itself whenever there is space left in all the buffers (i.e. when SThreadBusy or SDataThreadBusy equal zero).

Processing of the read requests and updating the request notification database (`m_readReqReadyQ`) is done in a similar manner as for write requests. However, read request addresses will be pushed onto the associated read address request buffer in `m_read_addrQ`, selected relative to the request thread ID. Also, for a single request burst read, the method will prepare the address sequence and push this onto the read address buffer.

It must be emphasized that the current slave implementation has following limitations:

- MCmd mnemonic supported are WR and RD.
- Assertion of SThreadBusy *or* transaction stalling by not asserting SCmdAccept only relates to full write data buffers.

Data handshake

Managing write request, when data handshake is a part of the OCP channel (`datahandshake==1`), is done in a different way, since the write data, MData will be transmitted together with the data handshake signals, contained in the `OCPDataHSGrp` object. In this configuration, all data handshake signals are managed by the process, `DataHSThreadProcess`. The motivation for separating the request and data handshake management is because the OCP protocol allows for time-wise separation of the request and data handshake phase. It means that a write request may be received at one point in time, but the actual data handshake phase may come several clock cycles later. The approach of separating request and data handshake handling also allows for support of out-of-order thread execution, since a request on another thread eventually could be processed, in the time interval in-between a request-handshake phase.

Managing the linking between a request and data handshake is done using a request buffer, `reqQ`. The buffer is an array of deque objects holding `OCPRequestGrp` objects, where array size equals the number of threads supported by the OCP channel. When a write request is received and the data handshake has not started yet, the process, `requestThreadProcess` pushes the received request object onto the buffer associated with the request thread ID (MThreadID). When `DataHSThreadProcess` receives the data handshake, it fetches the request object from the front of the request queue (selected relative to the thread ID, MDataThreadID), fills in the received data and forwards the request object to the method, `processRequest` which then process the request. For single burst write request, `requestThreadProcess` pushes N copies of the request object onto the request queue, where N equals the burst length.

Response handling

A response phase starts when a notification event is asserted on `start_response` by the `OCP_slave` process. This causes the process, `startResponse` to execute, which serves to prepare the response data, before starting the actual response. The process fetches the response data from the response data deque object (through the pointer, `dataQ` provided in the message from the IO task), converts this into `OCPResponseGrp` object and pushes this onto a deque object, `respQ` serving as a FIFO buffer for responses ready to transmit. For each response data, an object will be created and pushed onto the buffer. When all response data has been converted, the process initiates the response phase by asserting a notification event on `respQ_ready`. This causes `responseThreadProcess` to execute, which manages the assertion of response data onto the OCP channel as well as the protocol related to this. At each clock cycle the process fetches a new response group object from FIFO buffer, `respQ` and assert this onto the OCP channel. This continue until the FIFO buffer becomes empty and the response phase ends. Before `responseThreadProcess` suspends execution, it asserts a notification event on `respQ_empty`, informing `startResponse` that the response phase has ended. If there are no pending responses left to transmit, `startResponse` suspends as well. Otherwise is waits for response data to become ready in the buffer again and the procedure described above is repeated, until all responses has been transmitted.

11.6.5 OCP TL0

The OCP 2.0 TL0 model implements support for cycle true simulations at RTL level. However, the model does currently not support propagation delay emulation, with respect to delayed signal assertion. The implementation follows the *exact* same approach used for the implementation of the TL1 model, except that the OCP TL1 channel model has been replaced with dedicated `sc_in` and `sc_out` ports for the different OCP input output signals. Please refer to section 11.6.3, page 116 and 11.6.4, page 121 for an implementation description of the TL1 master and slave respectively.

Further, the TL0 slave uses the OCP signal mnemonic and signal group objects, `OCPRequestGrp` and `OCPResponseGrp`, available from the OCP transaction level library, from the header file `ocp_globals.h`. This is being done as a convenient way to encapsulate all information related to a request or response into a single object. For an example, when a request is being received, all request information from the TL0 interface will be stored in an `OCPRequestGrp` object and forwarded to `processRequest` for processing.

The OCP TL0 model is defined by the `TL0_IO` class, where the TL0 master and slave are defined by the classes, `TL0_Master` and `TL0_Slave` respectively.

11.6.6 Supported OCP TL0 configuration

The TL0 Model can be configured relative to the channel it connects to. However, compared with the TL1 model this must be done during building the framework. Thus a new channel configuration requires rebuilding of the model. The different configuration parameters are implemented as compiler statements and specified in the header file, `TL0_OCP_configuration.h`. Table 11.8 lists the configurations supported by the TL0 model.

Signal	Configuration Parameter	Comment
Request group		
MCmd		Always (1)
MAddr		Required
MData		Required
MReqLast	`_reqlast`	Optional
SCmdAccept	`_cmdaccept`	Optional (2)
Response		
MRespAccept	`_respaccept`	Optional (3)
SData		Required
SResp		Required
Burst extension		
MBurstSeq		Required
MBurstSingleReq		Required
MBurstPrecise		Required
MBurstLength		Required
Data handshake	`_datahandshake`	Optional (5)
MDataThreadID		Required
MDataLast	`_datalast`	Optional
SDataAccept	`_dataaccept`	Optional (4)
Thread extension		
MThreadID		Required
SThreadID		Required
MThreadBusy	`_mthreadbusy`	Optional (3)
SThreadBusy	`_sthreadbusy`	Optional (2)
SDataThreadBusy	`_sdatathreadbusy`	Optional (4)

(1) WR and RD are required to be supported by the OCP channel. Other types currently not supported by the IO device.
(2) Either SCmdAccept or SThreadBusy must be used. Disabling/enabling of both parameters are not allowed. When using SThreadBusy, the model follows the `sthreadbusy_exact` semantic.
(3) Either MRespAccept or MThreadBusy must be used (in conjunction with responses). Disabling/enabling both parameters are not allowed. When using MThreadBusy, the model follows the `mthreadbusy_exact` semantic.
(4) Either SDataAccept or SDataThreadBusy must be used when data handshaking is enabled. Disabling/enabling both parameters are not allowed. When using SDataThreadBusy, the model follows the `sdatathreadbusy_exact` semantic.
(5) If data handshake is disabled, all signals related to data handshake becomes non-applicable.

Table 11.13: OCP configuration supported by the OCP2.0 TL0 IO device.

11.7 Dependency controller

The dependency controller module manages the task dependency database, describing the dependencies between the tasks assigned to a simulation. The database is global in the sense, that it is being shared between the synchronizers in the different PE's in a controlled and well defined manner. Accessing the database from the synchronizers is done through a pointer to the dependency controller object, provided during PE module construction (see also section 11.2.1, page 96).

The dependency controller also manages the unblocking of tasks, when a task graph/application completes. In conjunction to this, the dependency controller keeps a database containing pointers to all the task objects. A pointer is used for gaining access to the unblocking method in a task object.

Figure 11.17 is a block diagram showing the connection between the dependency controller and the synchronizer and task objects. The dotted line indicates a pointer to the object.

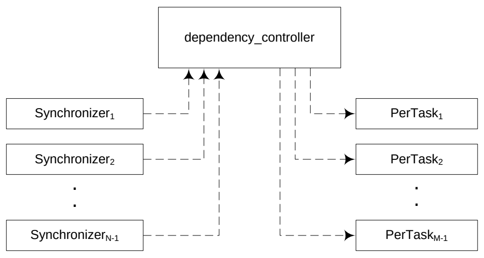


Figure 11.17: Block diagram showing the connection between the dependency controller and the synchronizer and task objects.

The dependency controller is defined by the class, `dependency_controller`. It has four public methods:

- `initialize_database.` Initializes the dependency database.
- `push_task_ptr.` Called when a pointer to a task object is provided to the dependency controller (done during PE module construction and task object creation).
- `finish.` Called by the synchronizer, when a task finishes and the database has to be updated.
- `mask.` Called by the synchronizer, when a check is done to see if all dependencies for a particular task has been resolved.

The implementation of the methods are described in the next sections.

11.7.1 initialize_database

The method, `initialize_database` serves to initialize the global dependency data base. It is being called at the top-level module, after the parsing has completed but before the simulation starts.

When calling the method, a pointer to the parser object is provided as argument, since the method needs to fetch the dependency database information from the configuration file.

First operation performed is copying the *mapping information look-up table* from the parser (`mapping_nfo`), which describes how an (end-to-end) task ID, defined by a group ID and subtask ID, maps to the extended dependency database[15]. This information will be used, whenever a synchronizer addresses an entry in the global dependency database, since addressing is always based on the end-to-end task ID. The mapping information is stored in the private unsigned integer matrix, `mapping_nfo`.

Afterward starts the task graph separation scanning algorithm. This algorithm scans the extended dependency matrix available from the parser, and extract all independent tasks graphs, which execute in parallel. It is being done by analyzing all dependencies. Separation of task graphs are required in conjunction to the management of task graph execution completion and task unblocking. For each task graph, the algorithm creates an unique dependency database object only containing the dependencies and other relevant information for this task graph. A dependency database object is a struct of the type `rm_type`. Table 11.14 describes the different entries in this database struct.

Type	**Name**	**Description**
`unsigned int` (NxN matrix)	`matrix`	Holds the dependency database.
`unsigned int` (1x2 array)	`boundary`	Holds an upper and lower row/column index, defining a boundary in the dependency data base, containing the task graph dependencies. `boundary[0]` holds the lower row/column boundary index, `boundary[1]` holds the upper row/column boundary index. Entries outside boundaries are don't cares since they are not a part of this task graph.
`bool` (1xN array)	`task_used`	A look-up table identifying the tasks belonging to the task graphs. The index maps directly to a row/column indexes used in the dependency database, `matrix`. An index marked as true indicates that this particular task belongs to the task graph.

Table 11.14: Task graph dependency data base struct, `rm_type`.

During the task graphs extraction scanning, the algorithm also checks for illegal task graphs, containing feedback edges. If such a task graph is detected, an error message will be asserted.

When the algorithm completes, the vector objects, `rm_list` and `rm_list_img` will contain pointers to the different task graph dependency databases created. Thus, it is the different dependency database objects together which makes up the global dependency database. The synchronizers operate on the database objects associated with `rm_list`, while the database objects associated with `rm_list_img` are original copies used for reference, when a task graph completes execution and the dependencies are to be restored.

[15]Please consult section 11.3.5, page 105 for a description of the extended relation database and the mapping information.

The example in figure 11.18, page 133 tends to illustrates outcome from the task graphs extraction. In the example two independent task graphs are considered, only composed of end-to-end tasks with one subtask. The top of the figure shows the task graphs, while the dependency database, as declared in the configuration, is shown below. When the task graph extraction algorithm finishes, two dependency data base objects will have been created; one for task graph 1 and another for task graph 2 respectively. The contents of the databases are shown in the bottom of the figure. The marked area in the `matrix` indicates the region containing the dependencies, as defined by the `boundary`. Notice in `matrix`, that the last task in a branch has been marked with 2. This is a special mark, inserted by the algorithm, used for end-of-branch indication. The mark is being used in the algorithm, detecting when a task graph has completed execution (implemented in the `finish` method described later).

11.7.2 push_task_ptr

Whenever a new task object is being created, the dependency controller must be provided with a pointer to the object. The pointer is be used for gaining access to the unblocking method in the task object, if the task has dependencies and whenever the associated task graph completes.

Calling `push_task_ptr` is be done during the dynamic task creation process, which is a part of PE module construction. See also section 11.2.1, page 98. When the method is called the provided pointer is pushed onto the vector object, `task_list_img` holding the different task pointers.

11.7.3 finish

The `finish` method is called from a synchronizer, whenever it receives a `FINISHED` message associated with a local task or external task (coming from the IO task). The operations performed by the method are:

- Clear dependencies in the dependency data base for local tasks.
- Checking if task graph execution has completed.

When calling the method, two arguments must be provided: the encoded ID[16] of the finished task and a pointer to a vector object, holding the ID's of the local tasks assigned to the particular PE in which the synchronizer is located[17].

Clear dependencies in the dependency data base for local tasks

First operation performed is resolving/clearing the dependencies to the local tasks. At first, the data base associated with the finished task ID is found from the `rm_list`

[16]see section 6.4.1, page 35

[17]the local task ID list is located in the synchronizer and is initialized during the dynamic task assignment phase. See also section 11.2.1, page 98.

by analyzing the `task_used` look up table in the different data base objects. If an entry associated with the finished task ID is true, then the correct data base has been found, since true indicates that the finish task belongs to the task graph encapsulated in the database. Afterward follows the clearing of the selected entries, based on the finished task ID and the local task ID's available from the task list, is cleared. The finished task ID is associated with column addressing while a local task ID is associated with row addressing. In conjunction to this, the actual row/column indexes are found using the mapping information, stored in `mapping_nfo`.

Checking if task graph execution has completed

After the database has been updated a check is performed to see if the particular task graph has completed execution. This is simply done by evaluating if all entries in `matrix`, within the defined boundary, has been cleared. If so, task graph execution has completed and a new execution cycle may start. First the dependencies are restored again, using information from the corresponding original reference database object, available from `rm_list_img`. Afterward follows the task unblocking. Which task to unblock is found by determine the ID's of the task belonging to the task graph. This is done simply by scanning `task_used`. Whenever an entry is true, the associated index is converted back to a task ID (using `mapping_nfo` look-up table) and a search in the task pointer list (`task_list_img` is done, until the associated task object is found. Identifying a task object is done by calling the task object method, `GetTaskID` which returns the encoded task ID. When there is a task ID match, the correct task object has been found and the method, `unblock` is called, causing the task to get unblocked. The procedure described above is repeated until all the tasks have been unblocked.

11.7.4 mask

The `mask` method is called whenever a synchronizer needs to checks if all dependencies has been resolved for a task. The method returns false if all dependencies have been resolved. Otherwise true. Argument provided to the method is the encoded task ID. First operation performed is finding the mapped index to use when addressing the dependency database. This is done based on the task ID and using the `mapping_nfo` look up table. Next step consists of finding the correct database object in `rm_list`, associated with the task ID. This is done by a look-up in `task_used`, using the mapped index, until true is found in a database object (indicates the task ID belongs to the particular task graph). When the correct data base object has been found, all column entries in `matrix` within the defined boundaries are added together, using a fixed row index, identified by the mapped ID. If the result is zero, then all dependencies have been resolved.

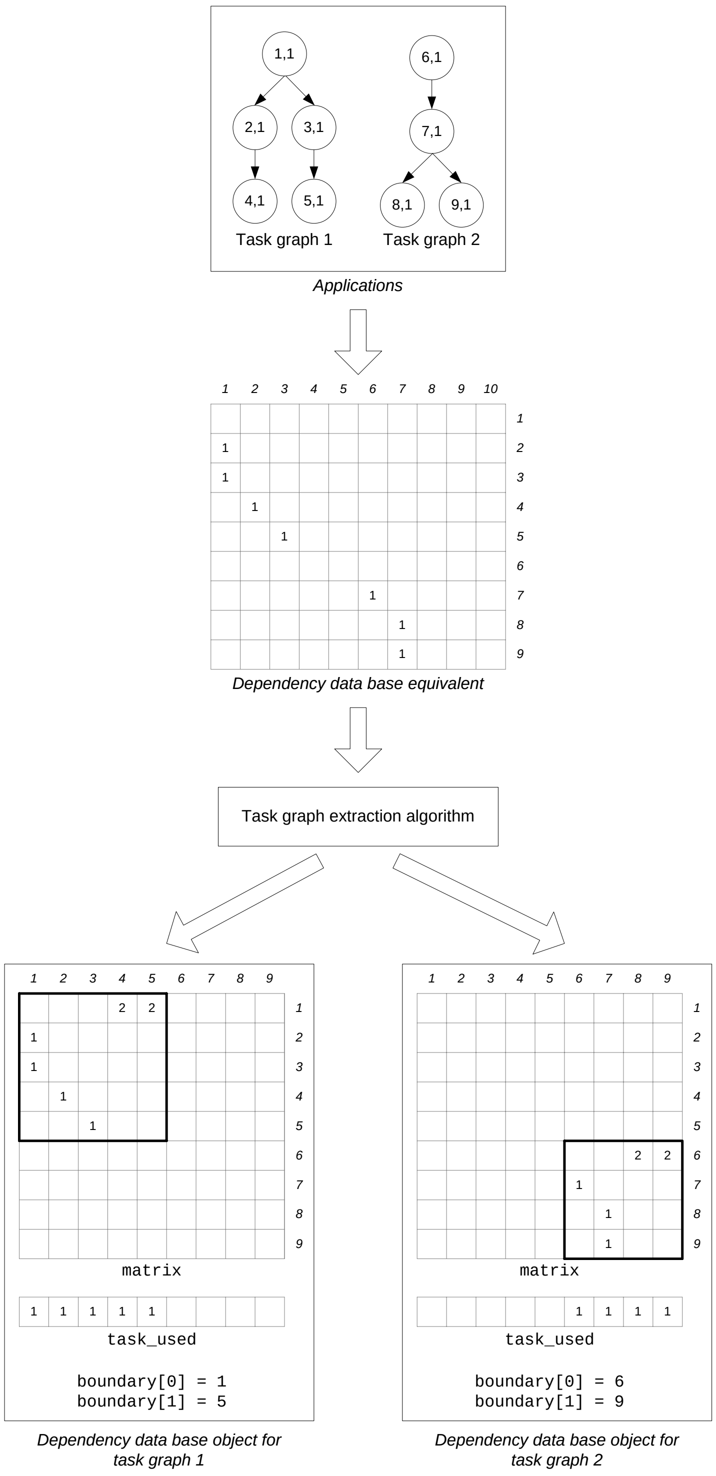


Figure 11.18: Example illustrating the task graph extraction.

11.8 Performance monitor

The performance monitor module monitors the end-to-end deadline for task groups with multiple as well as the performance of the different PE's assigned to the framework. PE performance covers utilization and IO task usage. The different types of monitoring is based on *reporting method calls* to performance monitor. For an example, in conjunction to the end-to-end deadline monitoring, a task calls a dedicated method in the performance monitor when execution starts end finishes.

Calling the reporting methods, form the different modules doing reporting, is done through a pointer to the performance monitor object, provided during PE module construction. Modules calling reporting method are the periodic tasks and IO task modules. Reporting done by the periodic task relates to end-to-end deadline and PE utilization monitoring, while IO task reporting relates to IO task usage monitoring.

Figure 11.19 is a block diagram showing the connection between the performance monitor and the IO task and periodic task objects. The dotted lines indicate pointer to the object.

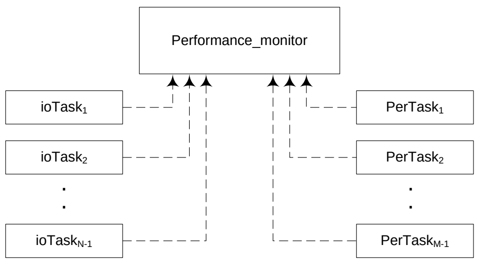


Figure 11.19: Block diagram showing the connection between the performance monitor and the IO task and periodic task objects.

The performance monitor is not a required module and may be left out when building the framework. In fact, doing so will reduce the simulation time. The performance monitor is defined by the `Performance_monitor` class.

11.8.1 Initialization

After a performance monitor module object has been created, the different internal databases must be initialized. This must be done after parsing but before the simulation starts. Initialization is done by calling the method, `initialize` with a pointer to the parser object and the number of PE's as arguments. From the parser object, it gets the number of subtasks for each end-to-end task as well as the end-to-end deadline. This information is stored in the Nx2 unsigned integer database, `subtask_DB` and will be used when initializing the associated end-to-end deadline counter, for an end-to-end task group. Row is associated with the end-to-end task group ID, while column 0 and 1 holds the no.of subtasks and the deadline respectively. If no end-to-end deadlines are declared at all (meaning that there are not any end-to-end tasks with multiple subtasks) the end-to-end deadline monitoring will be disabled.

The no.of PE's is used for initializing the PE utilization and IO task usage databases, `cpu_DB` and `iotask_DB` each being an array of the struct type, `cpu_DB_type` and `iotask_DB_type` respectively and with an array size equal to the no.of PE's. Thus gaining access to a database, for a particular PE, is accomplished by using the PE ID. Table 11.15 and 11.16 describes the entries in the structs.

Type	**Name**	**Description**
`bool`	`idle`	A flag identifying if the PE is in idle (true if so)
`double`	`count`	A counter used for keeping track of the no.of clock cycles a PE has been in used. Only incremented when not in idle (i.e. `idle`=false).

Table 11.15: PE utilization data base struct, `cpu_DB_type`.

Type	**Name**	**Description**
`unsigned int`	`type`	Identifies the current state of the IO task; e.g. being in idle (0), doing a write request (1) etc. The value of `type` maps directly to the different `type` valid for the IO task `RUN` message. See also table 11.5, page 110.
`double` (1x6 array)	`count`	A counter used for keeping track of the no of clock cycles used in the different states. When the counter is updated, the value of `state` is used for index selection.

Table 11.16: IO task data base struct, `iotask_DB_type`.

11.8.2 End-to-end deadline reporting methods

The methods, `subtaskRun` and `subtaskFinished` is called by a periodic task object when execution starts and completes respectively. Provided arguments to the methods are the group ID and subtask ID. The method, `subtaskRun` has only a meaning when called by the first subtask in an end-to-end task. When this happens a *deadline counter object* for the current task group is created and initialized and pushed onto the vector, `eedl_DB` holding deadline counter objects for activate end-to-end tasks. A counter object is a struct of the type, `eedl_DB_type`. The different struct entries is described in table 11.17.

During deadline counter object initialization, the number of subtask and the end-to-end deadline is fetched from `subtask_DB` and stored in `pendingSubTasks` a `eedl` respectively.

Type	Name	Description
`unsigned int`	`groupID`	The end-to-end task group ID associated with the deadline counter object.
`unsigned int`	`pendingSubTasks`	Identifying the number of subtask left in the group. Is decremented each time a subtask in the group finishes execution.
`unsigned int`	`nextSubTask`	Identifying the ID of the next expected subtask to finish. Used for error monitoring purpose only; that is if next subtask finishing execution has a different subtask ID than the one identified by `nextSubTask`, an error message will be asserted.
`unsigned int`	`eedl`	The end-to-end deadline counter. Is decremented in each clock cycle.

Table 11.17: End-to-end deadline counter struct, `eedl_DB_type`.

When `subtaskFinished` is called, the counter object associated with the group ID, provided as argument, is found in the vector, `eedl_DB` and `pendingSubTasks` is decremented while `nextSubTask` is incremented. If `pendingSubTasks` is zero after being decremented, it means that all subtask has been completed and the end-to-end task has finished execution. Thus the deadline counter object is erased from `eedl_DB` afterward.

11.8.3 PE utilization reporting methods

The methods, `cpu_busy` and `cpu_idle` are used when notifying the performance monitor that a PE is busy or in idle respectively. The methods are called from a periodic task object, when task execution starts and completes. Required argument to the methods are the PE ID. Calling `cpu_busy` causing the `idle` flag to be set to false in the associated data base entry in `cpu_DB`, while calling `cpu_idle` causing the flag to be set to true. Addressing the data base entry in `cpu_DB` is done using the provided PE ID.

When `idle` is false the utilization counter, `count` will be incremented each clock cycle.

11.8.4 IO task reporting method

An IO task reports to the performance monitor by calling the method, `iotask_state`. Whenever the IO task is launched for execution or finishes, it calls the method with the PE ID and the *type* of execution as argument (identified by the `type` entry in the message received from the scheduler. See also table 11.5, page 110). Using the PE ID, the method accesses a database entry in `iotask_DB` and set `type` equal to the execution type.

When incrementing the IO task usage counter, in each clock cycle, the value of `type` will determines which entry to increment in `count`.

11.8.5 Data base updating

Updating the counters in the different databases are managed by the `SC_THREAD` process, `update_monitor_DB` which executes at the rising edge of the clock. For the end-to-end deadline counter objects, located in `eedl_DB`, the `eedl` entry is decremented. If `eedl` has reached zero, it means that the end-to-end deadline has been missed for the particular task group, and a notification will be asserted to screen/log file. For the CPU utilization and IO usage databases, all the database entries, associated with the different PE's, are accesses and the counters are being incremented, if enabled.

11.8.6 Monitoring summary methods

A PE utilization and IO task usage summary is obtained by calling the methods, `pe_utilization_summary` and `iotask_usage_summary` respectively.

`pe_utilization_summary` will print the percentage utilization of the different PE's,defined as the no.of clock cycles a PE has been in use, relative to the total no. of simulation clock cycles.

`iotask_usage_summary` will print the percentage usage of the IO task, defined as the no.of clock cycles the IO task has been executed, relative to the total no. of clock cycles the associated PE has been in use. Also printed is the percentage usage for write requests, read request, write data receiving etc.

Chapter 12

Implementation: SoC communication platform model

This chapter presents the implementation specific details for the different modules, forming the SoC communication platform model. It is *highly* recommended to use the source code for reference, when reading this chapter. The source code can be found on the enclosed CD-ROM and may be used as reference. Please consult the README file for a directory contents description.

12.1 IO port

The IO ports modules currently available supports OCP2.0 at TL0 and TL1. These modules are the same being used for the IO device modules in the extended abstract PE model. Please consult section 11.6, page 115 for an implementation description.

12.2 Intermediate adapter

The intermediate adapter manages the interfacing between the IO port and the SoC communication layer model and is defined by the class, `SoC_comm_inter_adapt`. Whenever a transport message is received, it initiates a message to the IO port causing it to start a transaction. Similar, the intermediate adapter will issue a transport message to the SoC communication layer model, whenever a new SoC communication event (request/response) is received by the IO port. In conjunction to this, it uses a `sc_link_mp` master and slave port in the interface to the SoC communication layer model and IO port.

Figure 12.1 shows a simplified block diagram of the intermediate adapter. Oval figures indicate thread processes (`SC_THREAD` or `SC_SLAVE`) while rectangular figures indicate normal C++ methods. A name associated with a connection to a process (e.g. `request_start`) indicates the name of an `sc_event` object, used for triggering the process execution.

The next sections describe the implementation of the different methods and processes and how the interact together.

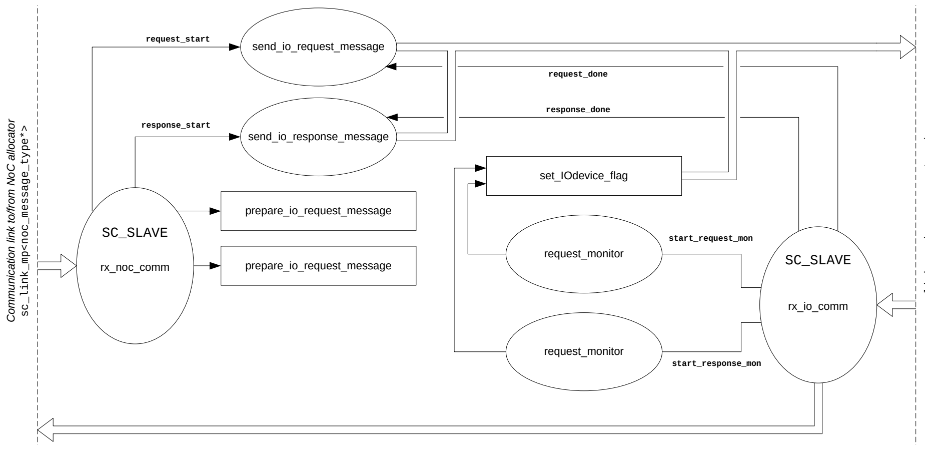


Figure 12.1: Simplified block diagram of the intermediate adapter.

12.2.1 Internal databases

The intermediate adapter incorporates databases for the management the request and response data received from the IO port. The request database is contained in `request_mon_DB` and is an array of the struct type, `request_mon_DB_type`. The response database is contained in `response_mon_DB` and is an array of the struct type, `response_mon_DB_type`. The array size for both databases equal the number of threads supported by the SoC communication interface (initialized in the class constructor). Thus there exist a database for each thread, since request/response data for the different threads must be treaded independently (also to support out-of-order execution).

The request and response databases are maintained by the clocked `SC_THREAD` processes, `request_monitor` and `response_monitor` respectively. The exact procedure for this will be explained later.

Table 12.1 and 12.2 describes the entries contained in the structs, `request_mon_DB_type` and `response_mon_DB_type`.

12.2.2 SoC communication layer interface slave port

The `SC_SLAVE` process, `rx_noc_comm` contains the slave port interfacing to the SoC communication layer. It decodes the transport message coming from the SoC allocator and determines the type of transfer to initiate. The supported message types and their meaning is described in table 12.3.

Type	Name	Description
`bool`	`done`	A flag used in the `request_monitor` process, for controlling when all data has been received for a request. The flag will be set to true when the request phase completes.
`unsigned int`	`type`	Holds the type of request (WR or RD)
`unsigned int`	`dataUnits`	Data transfer size. This is the same as the burst length.
`unsigned int`	`counter`	A counter used for monitoring the amount of data received for a requests. Incremented each time new data are fetched from the IO port data/address buffer.
`deque<unsigned int>*`	`addrQ`	Pointer to a deque object, used for buffering the addresses fetched from the IO port request address buffer.
`deque<unsigned int>*`	`dataQ`	Pointer to a deque object, used for buffering the data fetched from the IO port request data buffer. Only applicable for write requests.
`deque<unsigned int>*`	`ioDev_addrQ`	Pointer to deque object in the IO port, where to fetch request addresses from.
`deque<unsigned int>*`	`ioDev_dataQ`	Pointer to deque object in the IO port, where to fetch request data from.

Table 12.1: Request database struct, `request_mon_DB_type`.

12.2.3 Request transmission

A request transmission will be initiated when the process, `rx_noc_comm` receives a `RUN` transport message with the `type` entry being `WR` or `RD`, indicating write or read request respectively. If the transport message is associated with a read request, the corresponding response database entry from `response_mon_DB` is updated (selected using `threadID`). Updating consists of fetching the entries, `from` and `dataUnits` from the transport message and store this in the response database. See also table 12.2. This information will be used, when the response for the particular thread is received by the IO port. Afterward the process issues a call to the method, `prepare_io_request_message` which creates an IO port message (`io_message_type`), fetches the SoC communication related information from the transport message (transaction type, thread ID, data units and data/address deque pointers), copy this into the message object and finally pushes a pointer to the message onto the queue, `requestQ` (deque object) holding pending IO port request messages. Next, a notification on `request_start` is asserted, causing the process, `send_io_request_message` to fetch the message pointer from `requestQ` and forwards this to the IO port. The process now waits until a notification is asserted on `request_done`, indicating the request phase has been

Type	Name	Description
`bool`	`done`	A flag used in the `response_monitor` process, for controlling when all responses has been received for a previously initiated read request. The flag will be set to true when the response phase completes.
`unsigned int`	`from`	Holds the IO adapter ID from where the read request came from. Used for identifying the target for the transport message to issue, when the response phase starts.
`unsigned int`	`dataUnits`	The burst length of the read request. Used as a reference for identifying when all responses has been received.
`unsigned int`	`counter`	A counter used for monitoring the amount of response data received. Incremented each time new response data are fetched from the IO port response data buffer.
`deque<unsigned int>*`	`dataQ`	Pointer to a deque object, used for buffering the response data fetched from the IO port response data buffer.
`deque<unsigned int>*`	`ioDev_dataQ`	Pointer to a deque object in the IO port, where to fetch response data from.

Table 12.2: Response database struct, `response_mon_DB_type`.

`comm`	`type`	Description
`RUN`	`WR`	Start a write request.
	`RD`	Start a read request.
	`RESP`	Start a response phase (to a previously received read request).

Table 12.3: Supported transport messages types, received from the IO device. `comm` and `type` refers to the declaration in the transport message. See also table 9.1, page 57.

completed by the IO port. It then checks if there are any pending request messages buffered in the queue. If so, the next message pointer is fetched, and the same procedure described above is repeated, until the queue becomes empty. Then the process suspends itself.

12.2.4 Response transmission

A response transmission will be initiated when the process, `rx_noc_comm` receives a `RUN` transport message with the `type` entry being `RESP`. The procedure is similar to the request transmission handling, but using the method, `prepare_io_response_message` for IO port message preparation and the process, `send_io_response_message` for managing the message pointer forwarding to the IO port. In conjunction to this, the queue, `responseQ` is used for holding pointers to pending response messages, while `response_start` and `response_done` is used for notifying the process, `send_io_response_message` when a response message is ready and when a response phase has been completed by the IO port respectively.

12.3 IO port interface slave port

The `SC_SLAVE` process, `rx_io_comm` receives the messages coming from the IO port. Whenever a message is received, it is being decoded and an associated action will be initiated. An IO port message either indicates that some data have been received from the SoC communication interface (a request or response) or that a previous initiated request or response phase has been completed. The different types of supported IO port messages and their meaning is listed in table 12.4[1].

comm	**type**	**Description**
READY	RESP	A read request has been received.
	RD_RESP_READY	Response data, associated with a previously initiated read has been received and now ready to be fetched.
	WR_DATA_READY	A write request has been received and data is now ready to be fetched.
FINISH	WR	The initiated write request has completed.
	RD	The initiated read request has completed.
	RESP	The response phase (to a previously received read request) has completed.

Table 12.4: Supported messages types, received from the IO port. `comm` and `type` refers to the declaration in the message. See also table 11.4, page 107.

When receiving a `FINISH` message for a previously initiated request or response, the process asserts a notification on `request_done` or `response_done`. See also the previous sections describing the request and response transmission.

12.4 Request receiving

When the process, `rx_io_comm` receives request message from the IO port, it performs the following actions in the described order:

Creating and initializing a new transport message The thread ID (`threadID`) and burst length (`dataUnits`) is fetched from the IO port message and stored in the associated entries in the transport message. If the request being a read or write, the `type` entry in the transport message will be set to `RD`

[1]This table is similar to table 11.6, page 111, but repeated here for convenience.

or `WR` respectively. Also initialized in the transport message is the *position indication* entries. `from` and `now` is set equal the current node/IO adapter ID, while `to` is found by calling the method, `get_target_nodeID` (not shown in figure 12.1) with the first provided request address as argument. From the look up table, `node_address_matrix` containing the address space for the different IO adapters, it finds the ID of the target IO adapter and returns this.

Create deque objects for request address/data buffering New data and address deque objects are created, for buffering the addresses and data from the request. Pointers to these objects are included in the transport message (in the `addrQ` and `dataQ` entries), since the destination IO adapter must be able to fetch the data from the buffers when it receives the transport message and starts the request. If the request being a read, only an address deque object will be created.

Update the request database This is required by the process, `request_monitor` which manages the request phase and fetching of data from the IO port. Selecting the correct database entry is done using the request thread ID. Updating consists of storing the request type and data transfer size associated. Also stored are the pointers to newly created address/data buffers as well as the pointers to the address/data buffers provided by the IO port (from where to fetch the request address/data). Finally is the entry, `done` set to false, indicating a pending request on this thread, and the request counter entry, `counter` is initialized to 1 (since the first data from the request has been pre-fetched and stored in the request buffers as well). See also table 12.1, page 141.

Issues the transport message to the SoC communication layer model This is done after the request database entry has been updated.

Start request monitoring and data fetching After issuing the transport message, a notification is asserted on `start_request_mon`. This causes the clocked `SC_THREAD` process, `request_mon` to executes. At each clock cycles it checks all database entries and do the following actions for threads associated with an ongoing request (identified by `done==0`):

- fetches a new address/data from the IO task request buffers and pushes this onto the address/data buffers to be used by the target IO adapter. If the IO device request buffers are empty and the request phase is incomplete (indicating an interrupted burst request), then no operation is done.
- Increments the request counter entry, `counter` if an address/data was fetched from the IO port buffer.

When all requests have been received for a particular thread (that is when `counter==dataUnits`) the entry, `done` will be set to true, telling

`request_monitor` that there are no longer any ongoing request on this thread. Also, a `SET_FLAG` message is issued to the IO port, notifying that it must issue a `READY` message when a new request phase starts on this thread. The notification message is issued by calling the method,

`set_IOdevice_flag` with the thread ID (equals to the database entry addressing index) and a notification identifier (`RD_REQ_NOTIFY` if read request or `WR_DATA_NOTIFY` if write request) as arguments.

The `request_monitor` process suspends itself if there are no ongoing requests on any threads.

12.5 Response receiving

The actions performed when the process, `rx_io_comm` receives a response message from the IO port is similar to what is being done when receiving a request message. See also previous section. The only difference is that the response database, `response_mon_DB` and the clocked `SC_THREAD` processed, `response_mon` is being used for the response management. In conjunction to this, the event, `start_response_mon` will be used for triggering the execution of `response_mon`.

12.6 SoC allocator - 1D/2D mesh NoC topology model

The SoC allocator module defined by the class, `SoC_comm_alloc_mesh` models a 1D/2D mesh NoC topology with packet switched traffic. The routing is based on a minimal path algorithm. Selection of a minimal path will be done dynamically, relative to avoid link contentions whenever possible. The allocator can also be used for modeling a 1D mesh, by setting the mesh `span` parameter to 1, during module construction (see also the next section).

It must be emphasized that the current model considering links as being the *only* shared communication resources. From a modeling perspective, it means routers are assumed to have infinite buffers and zero latency.

The SoC allocator consists of a single process and method:

- `allocate`. A `SC_SLAVE` process managing the transport messages and implementing the actual algorithm, modeling the topology. It receives and transmit transport message pointers.
- `send_release_message`. A method called from `allocate` whenever a link, having a having pending reservation, is released.

The next sections explain the implementation approach used for modeling a 1D/2D mesh with minimal path routing. The descriptions relate to the code found in `allocate`.

12.6.1 Initialization - defining a mesh grid

The layout of a mesh is configurable, using parameter, $span$ provided to class constructor during object creation. This parameter defines the number of nodes per row/column, yielding a symmetrical mesh. Thus if $span = 3$ the corresponding modeled symmetrical mesh consists of 3x3 nodes. If the number of nodes connecting SoC allocator is smaller or larger than $span^2$, the modeled mesh becomes asymmetrical. A special case exists, with $span = 1$, which always will model a 1D mesh. Figure 12.2 illustrates different kinds of mesh grid configuration, relative to the value of span and the no. of network adapters connected (equivalent to the IO adapters in figure 9.1, page 54).

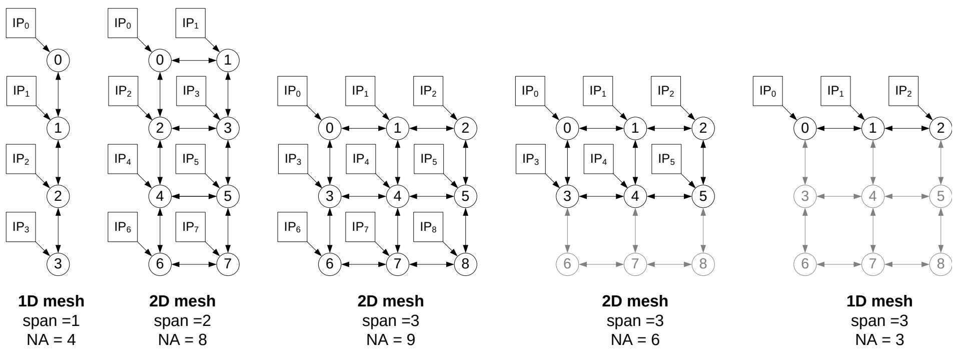


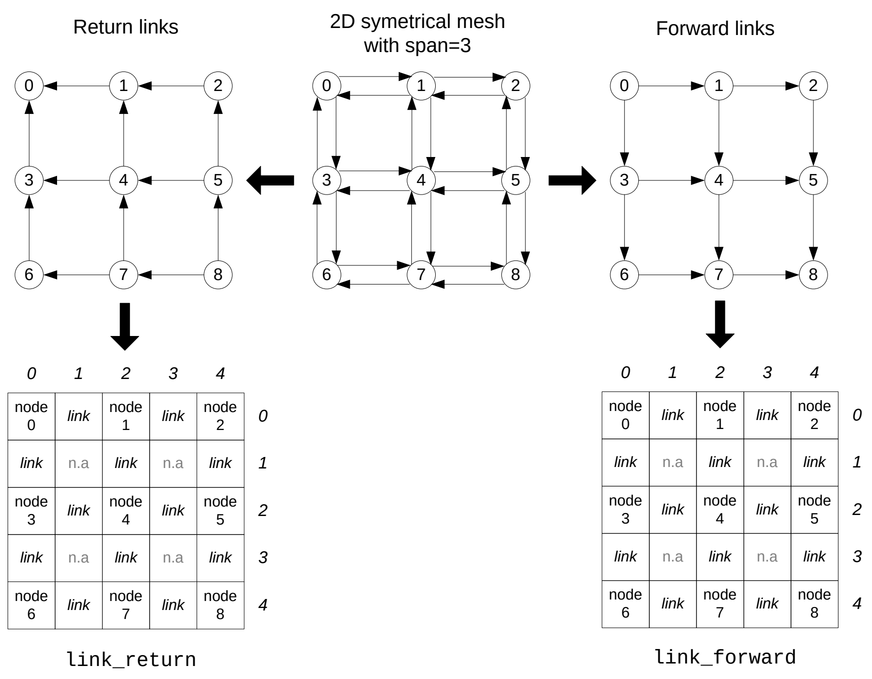
Figure 12.2: Examples of mesh grid configuration. NA defines the number of network adapters.

12.6.2 The mesh database

To manage the usage of the different network communication resources (in this model only consisting of links), two mesh databases, `link_return` and `link_forward`, are being used. Each database is implemented as an NxN unsigned integer matrix. `link_return` and `link_forward` is used in association with return and forward links respectively. The value of N is defined as $N = (2 \cdot span) - 1$ where $span$ equals the number of router nodes in a row/column, yielding a symmetrical mesh. Figure 12.3 shows an example how the routers nodes and links maps to the matrices. The example assumes a 2D mesh with span = 3.

As it can be seen, each entry in a mesh database corresponds to a certain position in the mesh, being a node or link. The mapping between the mesh and the database is straightforward. For an example, node 2 maps to entry $(0, 4)$ while the link between node 2 and 5 maps to entry $(1, 4)$.

The value of an entry in a mesh database is used for identifying the state of a particular link. Following scenarios exists:



0	1	2	3	4	
node 0	*link*	node 1	*link*	node 2	0
link	n.a	*link*	n.a	*link*	1
node 3	*link*	node 4	*link*	node 5	2
link	n.a	*link*	n.a	*link*	3
node 6	*link*	node 7	*link*	node 8	4

`link_return`

0	1	2	3	4	
node 0	*link*	node 1	*link*	node 2	0
link	n.a	*link*	n.a	*link*	1
node 3	*link*	node 4	*link*	node 5	2
link	n.a	*link*	n.a	*link*	3
node 6	*link*	node 7	*link*	node 8	4

`link_forward`

Figure 12.3: An example showing how a 2D mesh maps to `link_return` and `link_forward`.

Entry value	State
0	The link is free/not in use
1	The link is occupied but there is no pending reservation in the NoC scheduler.
>1	The link is occupied and there is a pending reservation on this link in the SoC scheduler. The number of reservations equals: entry value - 1.

For an example, if the value in entry $(1, 4)$ in `link_forward` is non-zero it means that the forward link between node 2 and 5 is in use.

Whenever a link is being reserved, the associated entry is incremented. When it is being released again, the entry is decremented.

12.6.3 The basic minimal path algorithm

The routing algorithm implemented by the SoC allocator aims to find the minimal path through the mesh. It is based on the pseudo code algorithm shown in figure 12.4.

Initial input to the algorithm is a current position, (i, j) and a destination position, (m, n) in a 2D mesh. During each iteration in the while-loop, the current position, (i, j) moves closer and closer to the destination, (m, n) until the position eventually is the same, after which the while loop terminates. The algorithm is very simple and tends to reduce the horizontal and vertical delta distances during each iteration. Whenever a horizontal or vertical delta distance equals zero (e.g. $i = m$ or $j = n$) the position is not changed any further. Following the movement

```
# (i,j) identifies the current position
# (m,n) identifies the destination position

while i<>m AND j<>n  do
  # select horizontal minimal path
  if i<>m then
    if i<m then
      i = i + 1
    else
      i = i - 1
    end if
  end if

  # select vertical minimal path
  if j<>n then
    if j<n then
      j = j + 1
    else
      j = j - 1
    end if
  end if

end while
```

Figure 12.4: Pseudo code for the basic minimal path algorithm for a 2D mesh

of current position, (i, j) from start to finish, it can be observed that this always will equal the minimal path in a 2D mesh.

The actual implementation of the algorithm is not fully identical to the pseudo code in figure 12.4. However, the computation steps for finding the next horizontal or vertical routing path in the mesh is similar.

12.6.4 Mapping a node position to a mesh database entry

When the allocator receives a transport message, the next link to use will be based on the routing information entries, `now` and `to` available from the message. `now` indicates the current node position of the message (node ID), while `to` indicates the destination node ID.

From the routing information entries, it is not possible directly to see the locations in the mesh since they do no contain any information about the topology. Thus to be able to select which link to use next, the entries must be mapped to the mesh, as it is being described by the mesh databases. The mapping will result in a position defined by a row and column index, (i, j). Finding the mapped row and column for a node ID is simply done by performing an *integer division* and *modulo division* between the node ID and the span and multiply by two, respectively. The equations are also shown below:

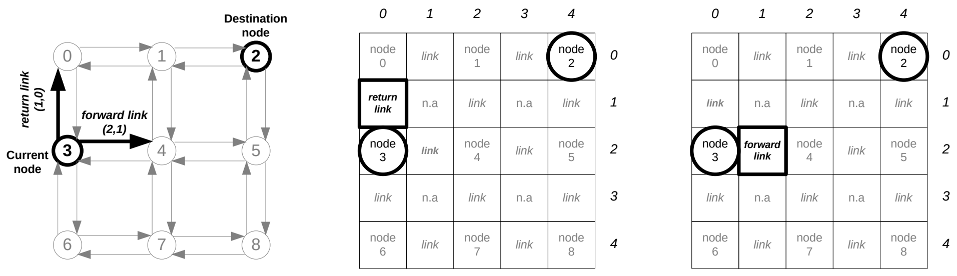
Row	$i = 2 \cdot (nodeID/span)$
Column	$j = 2 \cdot (nodeID\%span)$

Example: If the current node ID is 3 and the destination node ID is 2 and the span is 3, the corresponding mapping yields: $nodeID = 3 \mapsto (2, 0)$ and $nodeID = 2 \mapsto (0, 4)$. See eventually also figure 12.3, page 147.

12.6.5 Approach to link selection and reservation

After the position ID mapping, the next link to use if found from the mapping index, relative to the selecting a minimal routing path, as defined by the algorithm in figure 12.4. Depending on the relative placement of the current and destination node, it may be possible to use either a vertical or horizontal link. The following illustrates this:

Example: From the previous example, it was found that the current and destination node ID of 3 and 2 would map to $(2, 0)$ and $(0, 4)$ respectively. Here the minimal path algorithm finds the next possible link to use to be either the *vertical return link* identified by $(1, 0)$ or the *horizontal forward link* identified by $(2, 1)$. This is also illustrated in figure 12.5 showing the links in the mesh and the corresponding entries in the mesh databases.



(A) Possible links to use

(B) Corresponding entries in the `link_return` mesh database

(C) Corresponding entries in the `link_forward` mesh database

Figure 12.5: The next possible next links to use. Figure A shows the links in the 2D mesh while figure B shows the corresponding entries

Whenever it is possible to use either a vertical or horizontal link, the implemented algorithm always tries to reserve a vertical link first. If the link is being occupied (the associated entry in the selected mesh database in non-zero) and it is possible to use a horizontal link, it will reserve this instead; also if it is being occupied. If it is only possible to use a certain link (vertical or horizontal), a reservation will be put on this link, no matter if it is being occupied or not.

In conjunction to link reservation, this is being done by incrementing the associated entry in the selected mesh database. Also, mesh database selection depends on the delta distance to move in the mesh being negative or positive, index-wise.

Example: Considering the example from before (see also figure 12.5), it is now assumed that a reservation of the horizontal forward link, identified by $(2, 1)$ is to be done. Here the selected mesh database would be `link_forward` since the delta distance to move is positive. After selection, the entry in $(2, 1)$ in `link_forward` is incremented.

12.6.6 Transport message management

In a transport message, the entry, `resourceID` is used for identifying the shared resource used or to be used by the message (see also table 9.1, page 57). For this SoC allocator, it refers to a particular link in the mesh and is implemented as a 1x3 matrix, encoded as shown in table 12.5.

Entry	Description
0	Identifies the link type, where 0=reverse and 1=forward.
1	The horizontal link position in the mesh database.
2	The vertical link position in the mesh database.

Table 12.5: Encoding of `resourceID`.

When the process, `allocate` receives a transport message not coming from an intermediate adapter, is means that a previous link, identified `resourceID` has been released. Based on the entries in `resourceID`, a mesh database is selected and the associated value in the database entry is decremented. If there is a pending reservation on this link, a call is made to the method, `send_release_message`. This method issues a `RELEASE` message to the scheduler, containing the resource ID information, causing the pending transport message to be released.

Afterward a check is done to see if the transport message has reached the destination node, by evaluating if the position indicators, `now` and `to` are equal. If so, it means that the destination adapter has completed the inter-processor communication event and the message may now be deleted. If the transport messages has not reached the destination node yet, the next link to use is found using the approach described in the previous sections. In conjunction to this the entry, `resourceID` is updated to reflect which link to use, while current node ID position,`to` is updated to reflect the next node ID position of the message. Also, the entry, `CSL` identifying for how long time the link is to be occupied, is initialized a value equal to the data transfer size (identified by the entry, `dataUnits`. If there was no link contention, the transport message is finally changed to a `GRANT` message and forwarded to the SoC allocator. Otherwise it is changed to a `REFUSE` message and forwarded to the SoC scheduler.

12.7 SoC allocator - single shared bus model

The SoC allocator module, defined by the class, `SoC_comm_alloc_bus` models a single shared bus, with the bus being granted on a first-come-first-served basis. The model is shown in figure 12.6.

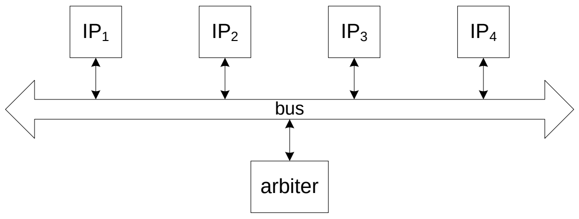


Figure 12.6: Single shared bus model.

The implementation is similar to the 1D/2D Mesh model, except that there is only one link available. Otherwise the approach for doing link reservation and transport message management is exactly the same.

The model consists of a process and a method:

- `allocate`. A `SC_SLAVE` process managing the transport messages and the bus database. It receives and transmit transport message pointers.
- `send_release_message`. A method called from `allocated` whenever the bus is released and there is a pending reservation on the link.

The bus database is defined by the unsigned int variable, `bus`.

12.8 SoC resource usage buffer

The SoC resource usage buffer is used for buffering transport messages while using a shared communication resource. It is defined by the class, `SoC_comm_res_buff` and consists of the following processes:

- `rx_noc_message`. A `SC_SLAVE` process receiving transport message pointers from the SoC allocator. Whenever it receives a `GRANT` transport message, the pointer is pushed onto the resource usage buffer, `buffer` (a deque object, holding pointers to transport messages).

  It also checks if `now == to`. If this condition is met, it means that message is to reach the destination IO adapter, since the current resource connects to the destination node. Thus the destination IO adapter may start the SoC communication event simultaneously with the resource being granted. This is accomplished by creating a copy of the transport message, change it to a `RUN` message and forwards this to the destination intermediate adapter.
- `update_buffer`. A clocked `SC_THREAD` process for maintaining the buffer. In each clock cycle the `CSL` entry in the different buffered messages is decremented and whenever `CSL` has reached zero, the transport message pointer is removed from the queue and forwarded back to the SoC allocator again.

12.9 SoC scheduler

The SoC scheduler is used for buffering transport messages waiting for a shared communication resource to become free. Scheduling is done according to the first-come-first-served principle. The SoC scheduler is defined by the class, `SoC_comm_scheduler` and consists of a process and a method:

- `rx_noc_message`. A `SC_SLAVE` process receiving transport message pointers from the SoC allocator. A `REFUSE` transport message, causes the transport message pointer to be pushed back onto the queue, `Q` (a deque object, holding pointers to transport messages). A `RELEASE` message initiates a call to the method, `check_queue`.

- `check_queue`. Scans the transport message pointer queue, `Q` until finding the first pending message waiting for this resource (identified by the entry, `resourceID`). When the message has been found it is being changed to a `GRANT` transport message, forwarded to the SoC usage buffer and the pointer is removed from the queue, `Q`.

Chapter 13

Conclusion

The aim with this project has been to integrate an abstract SystemC based RTOS model for MPSoC [7], together with a OCP2.0 based communication platform for low-level inter-processor communication. The motivation for this is to expand the ability to mix and integrate different SoC models, operating at different abstraction levels, into a common SoC communication platform having the same interface to all models. Having the ability to select from a variety of SoC models, representing IP cores a different abstraction levels, will serve as a powerful tool for SoC designers, since it becomes possible to customize a simulation framework, relative to the design space exploration experiments to perform (e.g. for SoC communication platform analysis). Thus IP core abstraction level refinement can be done as desired.

As an extension to the project, a SystemC based abstract SoC communication platform model has been developed as well.

The original abstract RTOS based PE[7] model has been extended to support low-level SoC communication by implementing an IO device and IO task module on top of it. The IO device models a physical hardware port while the IO task models a device driver application, used for controlling the hardware port and managing data/address encoding, related to inter-task dependency. Two IO device modules has been implemented, supporting OCP2.0 at TL0 and TL1 respectively. They have a multi-threaded interface ans support out-of-order thread execution. Further, they are configurable (signal-wise) relative to the channel they connect to.

In conjunction to the management of inter-task dependencies, a new *task dependency module* has been added on top of the original synchronizer module. The module serves as a common synchronization database, since all synchronizers connect to this module. It also fixes the lack of support for periodically task graph execution, which was a serious issue in the original synchronizer module, limiting the simulation time to a single task graph period.

Further, the abstract RTOS based PE model has been extended to support end-to-end tasks [20], which is being used in conjunction to the implementation of read/response based inter-processor communication.

A new *performance monitor module* has been implemented as well. This module monitors the performance of the different PE's, with respect to utilization and time spend on inter-processor communication etc., thus providing useful information to the SoC designer. It also serves to monitor the deadline for end-to-end tasks, consisting of multiple subtasks.

Finally, the performance of the model has been improved 10..15% by changing the `sc_mp_link` module communication approach from message object passing to *pointers* to message object passing.

To provide the SoC designer with an easy and flexible method for configuring an abstract PE based MPSoC simulation framework, a dedicated *configuration file parser module* has been implemented as well. The parser accepts a configuration file as input, describing different parameters for a simulation (task assignment/partitioning etc.), using a simple script language as showed in the example in figure 8.8, page 51. From the configuration file declarations, a simulation framework can be dynamically created/configured, without having to rebuild the system-level model again. Further, the simplicity and modularity of the parser makes it very easy to implement new script commands and data types, if needed.

All new modules have been implemented with emphasis on modularity, to ensure backward compatibility with any previous simulation frameworks, based on the abstract RTOS model. This approach also makes it easy to implement new hardware port and IO device driver models, since the interface to the modules is well-defined.

In conjunction to the extended abstract PE model, a SystemC based abstract SoC communication platform model has been developed as well. It supports modeling of different topologies, ranging from a simple bus to a 1D/2D mesh based NoC, using minimal path routing and packet switched traffic. It favorers from being able to support transmission of real data, while still having an abstract description of the underlying communication topology. Further, the modular implementation approach makes it easy to implement new topology models as well as other SoC communication protocols. Another interesting feature arising from the modularity, is the ability to integrate/mix communication protocols operating at different abstraction levels (e.g. OCP2.0 TL0 and TL1) or for that matters different SoC communication protocols (e.g. OCP and Wishbone). The current model supports OCP2.0 TL0 and TL1 SoC communication protocols.

With the new extensions added to the abstract PE model, the power and flexibility of the model has increased to even new heights. Especially the introduction of the configuration file parser, has moved the model up to a level, where it has become useful in practice, since a simulation framework no longer needs to be rebuild each time design parameters such as task partitioning or scheduling policies are changed. The usefulness of the extended abstract PE model as well as the new SoC communication model was clearly demonstrated using a series of different design space exploration experiments, ranging from analysis of a simple architecture, consisting of a bus and two abstract PE's, to a more complex performance analysis of an architecture, consisting of nine abstract PE's connected in a 2D mesh. Thus it can be concluded that the objectives for the project are successfully met.

Chapter 14

Future work

This chapter highlight some of the suggested future work, related to the RTOS and SoC communication frameworks.

14.1 RTOS framework

- **New method for doing task dependency declaration** The current approach using a dependency matrix works fine, but for many tasks ($>$100) the method becomes cumbersome, also increasing the chances for making errors. A new approach could be define the precedence or succeeding dependencies for each task in the task declaration.

- **Support for multiple transaction types (write and/or read) for a task having multiple inter dependencies** In the current framework the same transaction type will be used for all preceding edges associated with inter dependency.

- **Better summary reporting after simulation completes** That is also including information such as no.of missed deadline, average execution time etc. for the different task graphs. See also section 10.1.4, page 67 presenting an example of the current summary reporting.

- **Support for transmission of real data between tasks** This is required for doing MPARM behavior emulation and relates to the implementation of the task model.

- **Support for conditional task graph execution/branching** Emulation of non-static task graphs and features such as semaphore and mutexes. Relates to the implementation of the synchronizer, task model and IO task.

14.2 SoC communication platform framework

- **Extend the SoC allocator for mesh based NoC modeling** The current model assumes a router node has infinite buffer and zero latency. This is too unrealistic/optimistic and should modeled as a shared communication resource as well.
- **Better real-time and summary reporting** The current approach, as for an example, shown in figure 10.13, page 77 is far from optimal.

14.3 Simulation presentation in general

As stated above, it is common for both frameworks, that the simulation result presentation is non-optimal. Getting a quick overview of the system performance and potential bottlenecks, related to parameters such as bad scheduling and task partitioning, becomes difficult when doing real complex system-level modeling (e.g. nine PE's and five applications) and using very long simulation times. Implementing a well organized summary reporting method is therefore essential, if the frameworks are to become easy to use for very complex design space exploration experiments.

Bibliography

[1] Cofluent studio. http://www.cofluentdesign.com.

[2] Ocp international partnership. http://www.ocpip.org.

[3] Software arm. http://www.g141.com/projects/swarm.

[4] Specc. http://www.specc.org.

[5] Systemc based soc communication modeling for the ocp protocol. http://www.ocpip.org. white paper.

[6] *Network-Centric System-Level Model for Multiprocessor SoC Simulation*. 2004.

[7] *A SystemC-Based Abstract Real-Time Operating System Model for Multiprocessor System-on-Chip*. Morgan Kaufmann Publishers, October 2004.

[8] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. Rtos modeling for system level design. In *Proceedings of the conference on Design, Automation and Test in Europe*, page 10130. IEEE Computer Society, 2003.

[9] Knud Martin Hansen. Simulation framework for wireless sensor networks. Master’s thesis, Technical University of Denmark, IMM, 2004.

[10] Fabiano Hessel, Vitor M.da Rosa, Igor M.Reis, Ricardo Planner, Cï¿½ar A.M.Marcon, and Altamiro A.Susin. Abstract rtos modeling for embedded systems. In *IEEE International Workshop on Rapid System Prototyping*, 2004.

[11] J.Sun and J.Liu. Synchronization protocols in distributed real-time systems. In *The 16th International Conference on Distributed Computer Systems*, pages 48–45, May 1996.

[12] Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing on-chip communication in a mpsoc environment. In *Proceedings of the conference on Design, automation and test in Europe*, page 20752. IEEE Computer Society, 2004.

[13] Jan Madsen, Kashif Virk, and M.Gonzalez. Abstract rtos modelling for multiprocessor system-on-chip. In *International Symposium on System-on-Chip, pp.147-150*, 2003.

[14] Shankar Mahadevan, Federico Angiolini, Michael Storgaard, Rasmus Grndahl Olsen, Luca Benini, Jens Spars, and Jan Madsen. A network traffic generator model for fast network-on-chip simulation. In *International Symposium on System-on-Chip*, 2005.

[15] Grant Martin. Design-to-volume: Cadence nanometer design strategy. http://lina.ee.ntu.edu.tw/course/HW_SW_Codesign_2004/slides/Part-IV-SystemC-03092004.pdf.

[16] R. Le Moigne, O. Pasquier, and J-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *Proceedings of the conference on Design, automation and test in Europe*, page 30082. IEEE Computer Society, 2004.

[17] OCP-IP Association, 5440 SW Westgate Drive, Suite 217 Portland, OR 97221. *Open Core Specification Protocol 2.0*, revision 1.1.1 edition, 2003.

[18] OCP-IP Association, 5440 SW Westgate Drive, Suite 217 Portland, OR 97221. *A SystemC OCP Transaction Level Communication Channel*, revision 2.0.3 edition, 2004.

[19] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. In *IEEE Transactions on computers, vol.39, no. 9*, September 1990.

[20] Jun Sun. *Fixed-Priority End-to-End Scheduling In Distributed Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[21] SystemC. *SystemC Version 2.0.1 Master-Slave Communication Library*, 2002.

[22] SystemC. *SystemC Version 2.0.1 User's Guide*, 2002.

[23] Kashif Virk and Jan Madsen. A system-level multiprocessor system-on-chip modeling framework. In *International Symposium on System-on-Chip*, 2004.

Appendix A

Parser database descriptions

1. **Declaration type mnemonic:** `screen_dump`

Database type description
A boolean variable, `screen_dump`.

Access methods

Name	Description
`get_screen_dump`	Returns the value of `screen_dump`

2. **Declaration type mnemonic:** `log_file`

Database type description:
Array of chars, `log_file[]` holding the log filename.
Access methods

Name	Description
`get_log_file`	Returns a pointer to `log_file`

3. **Declaration type mnemonic:** `vcd_file`

Database type description
Array of chars, `vcd_file[]` holding the VCD filename.

Access methods

Name	Description
`get_log_file`	Returns a pointer to `vcd_file`

4. **Declaration type mnemonic:** `sub_task_map`

Database type description
A vector, `task_list` holding pointers to struct objects (`task_type`), encapsulating the subtask task declarations (`vector<task_type*>`). The struct, `task_type` contains the following entries.

Type	Name	Descripition
`char` (array)	`name`	The task name
`unsigned int` (array)	`arg`	The different task parameters
`vector<res_type>`	`resource`	A vector holding structs, containing the different resource requirement. `res_type` is the struct and consists of an unsigned integer array, `arg` holding the resource requirement parameters

Access methods

Name	Description
`task_list_size`	Returns the size of `task_list`.
`get_task_from_list`	Returns the pointer (`task_type*`) to task declaration struct object no. N, where N is the argument provided to the method.
`Get_subtaskID`	Returns the ID of the subtask, associated task declaration struct object no. N, where N is the argument provided to the method.

5. Declaration type mnemonic: `dependency_database`

Database type description
A struct object, `rm` of the type `dependency_matrix_type`, containing the end-to-end dependency database as well as the extended/finalized database, with subtask dependencies. Also includes the size of each database and the *index offset* look-up table, associated with the finalized database.

Type	Name	Description
`unsigned int` (matrix)	`parent`	The end-to-end dependency database, as specified in the configuration file.
`Unsigned int` (array)	`parent_size`	Row and column size of the end-to-end dependency database.
`unsigned int` (matrix)	`subtask`	The finalized dependency database, with subtask dependencies.
`unsigned int` (array)	`subtask_size`	Row and column size of the finalized database
`unsigned int` (matrix)	`mapping_nfo`	A look-up table, holding the index offset for each end-to-end task as well as the no. of subtask in each end-to-end task. `[k][0]` = no.of subtask belonging to parent ID, `k`. `[k][1]` = index offset for parent ID, `k`.

Access methods

Name	Description
`dependency_matrix_size`	Returns the size of the finalized dependency database, `subtask`.
`get_dependency`	Returns an entry from the finalized dependency database, `subtask`. The row and column index is provided as argument to the method.
`get_mapping_nfo`	Returns a value from the `mapping_nfo` look-up table. Argurmnts to be provided are the parent ID, k and the data selection value (0 or 1) `(k,0)` returns no.of subtask belonging to parent ID, `k`. `(k,1)` returns the index offset for parent ID, `k`.
`get_dependency_matrix`	Returns a pointer to the struct object, `rm` containing all information related to the dependency database (See table above).

6. Declaration type mnemonic: `ee_deadline`

Database type description
A vector, `dl_list` holding pointers to struct objects (`ee_deadline_type`), encapsulating the parent ID and associated end-to-end deadline (`vector<ee_deadline_type*>`). A struct, `ee_deadline_type` consists of an unsigned integer array, `arg` with two entries, were arg[0] holds the parent ID and arg[1] holds the end to end deadline.

Access methods

Name	Description
`get_ee_deadline`	Returns the end-to-end deadline associated with parent ID, provided as argument. Target variable for the returned end-to-end deadline must be provided as argument as well. The function returns true if an end-to-end deadline exists for the parent ID. Otherwise false.

7. Declaration type mnemonic: `module`

Database type
A vector, `module_list` holding pointers to *other* vector objects, holding struct objects (`module_type`), encapsulating the different module declarations. A declaration consists of a name and one or multiple values. The struct, `module_type` contains the following entries.

Type	Name	Descripition
`char` (array)	`name`	The parameter name
`unsigned int` (array)	`arg`	The different values associated with name

For each module declaration in the configuration file there will be generated a vector object, and a pointer to this is stored in `module_list` (`vector<vector<module_type*>*>`).

Access methods

Name	Description
`module_search`	Search for a module in `module_list,` containing a certain declaration name and value (e.g. "peID", 3) provided as argument to the method. Returns a pointer to the *first* detected module having the declaration. The pointer points to the vector, holding the module declarations (`vector<module_type*>`). If no modules have this declaration the method returns null.
`get_module`	Returns a pointer to the `k`'th module declaration, where `k` is provided as argument to the method. The pointer points to the vector, holding the module declarations (`vector<module_type*>`). If k is out of bound, the method returns null.
`get_address`	A macro method returning the SoC communication address declaration for a PE, where the ID for this PE is provided as argument. The macro will only search through module declarations, containing a `"peID"` and `"address"` name declaration. Target variables for the returned address (low, high limit) must be provided as argument as well. When a module is found the method returns true. Otherwise false.

Appendix B

sc_link_mp communication benchmarking

The experiment presented in this appendix describes a simulation performance benchmarking test procedure, used for investigating different `sc_link_mp` communication approaches. The source code can be found on the enclosed CD-ROM in `/ARTS_Model/builds/test`. Two approaches are considered:

- Message objects passing (`sc_link_mp<message_type>`)
- Message pointer object passing (`sc_link_mp<message_type*>`)

Test configuration

The test configuration consists of a module having master and slave port (`sc_outmaster` and `sc_inslave`). Two modules are instantiated and connected in a back-to-back configuration. At each clock cycle, a master will transmit 100 messages to slave in the other module, which fetches a value from the message. Before starting a new transmission sequence a message is created. After the end transmission sequence, the message is deleted again.

The implementation is the same for both communication scenarios. The only difference is the data type asserted onto the `sc_link_mp`. Table B.1A and B.1B shows the C++ source code using pointer and message passing respectively.

Running benchmarking

Each benchmark program was executed in a sequential manner for 100 times, using a fixed simulation time and clock period of to 20000ns and 1ns respectively. Simulation time was logged onto a file and processed in MS Office Excel afterward. Platform used for simulations was Fedora Core 2 running on a Asus

```
void test_ptr::slave()
{
  unsigned int type;
  message_type* temp = in;
  type = temp->type;
}


void test_ptr::master()
{
  message_type* temp;
  while(true) {
    wait();
    temp = new message_type;
    for(unsigned int i=0;i<100;i++) {
      temp->type = i;
      out = temp;
    }
    delete temp;
  }
}
```

(A) Message object passing
`sc_link_mp<message_type>`

```
void test_msg::slave()
{
  unsigned int type;
  message_type temp = in;
  type = temp.type;
}


void test_msg::master()
{

  message_type* temp;
  while(true) {
    wait();
    temp = new message_type;
    for(unsigned int i=0;i<100;i++) {
      temp->type = i;
      out = *temp;
    }
    delete temp;
  }
}
```

(P) Message pointer passing
`sc_link_mp<message_type*>`

Table B.1: C++ source code for benchmarking.

L3800 laptop@2.2GHz Pentium mobile. Figure B.1 shows an error bar plot of the different simulation times obtained for the two benchmark programs. The X-axis is the simulation time.

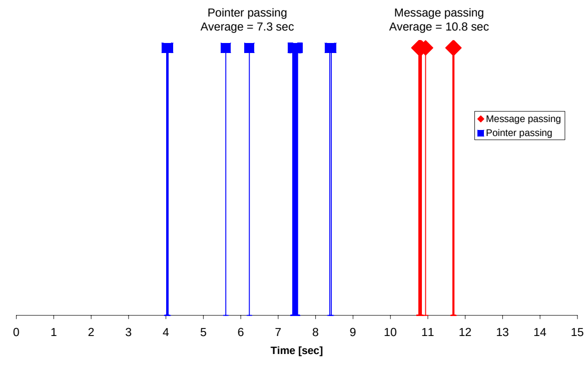


Figure B.1: Simulation times using message object passing and pointer parsing.

Conclusion

As it can be seen, the average simulation time by passing message objects and message pointers is 10.8 sec. and 7.3 sec respectively. Thus, by passing pointers to message objects instead of message objects gives an performance improvement of approx. 1.47. The reason for this is because message object construction/destruction is avoided. However it is to believe that difference in the program code may yield different benchmark figures.

Appendix C

OCP channel configuration for examples

parameter	vaule
`addr_wdth`	32
`burstlength_wdth`	32
`cmdaccept`	1
`dataaccept`	1
`datahandshake`	1
`datalast`	1
`data_wdth`	32
`mthreadbusy`	0
`reqlast`	1
`respaccept`	1
`sdatathreadbusy`	0
`sthreadbusy`	0
`threads`	10

Table C.1: OCP2.0 TL0 channel configuration (declared in `TL0_OCP_configuration.h`).

parameter	vaule
addr	1
addr_wdth	16
addrspace	0
burstlength	1
burstlength_wdth	16
burstprecise	1
burstseq	1
burstseq_unkn_enable	1
burstsinglereq	1
cmdaccept	1
dataaccept	1
datahandshake	1
datalast	1
data_wdth	32
mdata	1
mthreadbusy	1
mthreadbusy_exact	1
rdlwrc_enable	0
readex_enable	0
read_enable	1
reqlast	0
mreset	0
sreset	0
reqdata_together	1
resp	1
respaccept	0
sdata	1
sthreadbusy	0
sdatathreadbusy	0
sthreadbusy_exact	0
threads	10
write_enable	1
writenonpost_enable	0
writeresp_enable	0

Table C.2: OCP2.0 TL1 channel configuration (declared in the parameter file, ocp.)

Appendix D

Simulation logfile for example 1

```
  0 s     PE#0: task(1,1) (ready) -> scheduler
  0 s           synchronizer: READY from task(1,1) received.
              all dependencies resolved
  0 s     PE#0: scheduler (run) -> task(1,1)
  0 s     PE#0: task(2,1) (ready) -> scheduler
  0 s           synchronizer: task(2,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#0: task(4,1) (ready) -> scheduler
  0 s           synchronizer: task(4,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#0: task(6,1) (ready) -> scheduler
  0 s           synchronizer: task(6,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#0: task(8,1) (ready) -> scheduler
  0 s           synchronizer: task(8,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#0: task(10,1) (ready) -> scheduler
  0 s           synchronizer: task(10,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#0: task(12,1) (ready) -> scheduler
  0 s           synchronizer: task(12,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#0: task(14,1) (ready) -> scheduler
  0 s           synchronizer: task(14,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#0: task(16,1) (ready) -> scheduler
  0 s           synchronizer: task(16,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#1: task(3,1) (ready) -> scheduler
  0 s           synchronizer: task(3,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#1: task(5,1) (ready) -> scheduler
  0 s           synchronizer: task(5,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#1: task(7,1) (ready) -> scheduler
  0 s           synchronizer: task(7,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#1: task(9,1) (ready) -> scheduler
  0 s           synchronizer: task(9,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#1: task(11,1) (ready) -> scheduler
  0 s           synchronizer: task(11,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#1: task(13,1) (ready) -> scheduler
  0 s           synchronizer: task(13,1) has a dependency.
              pushed on the Pending Tasks Queue
  0 s     PE#1: task(15,1) (ready) -> scheduler
  0 s           synchronizer: task(15,1) has a dependency.
              pushed on the Pending Tasks Queue
 45 ns    PE#0: task(1,1) (request to NoC: task(3,1),addr=0x1011,dataUnits=10)-> adaptor
 45 ns    PE#0: scheduler (start NoC write request) -> task(IO)

 45 ns    |OCP| PE0_TL0.IOdevice.master: sent BURST request.
          | M | Data handshake: yes
          | A | MCmd: WR, MAddr: 0x1011, MThreadID: 0x1
          | S | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1
```

```
    46 ns     |OCP| soc_comm.tl0_io_a.slave: receiving BURST request.
              | S | Data handshake: yes
              | L | MCmd: WR, MAddr: 0x1011, MThreadID: 0x1
              | A | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


    46 ns     |OCP| soc_comm.tl0_io_b.master: sent BURST request.
              | M | Data handshake: yes
              | A | MCmd: WR, MAddr: 0x1011, MThreadID: 0x1
              | S | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


    47 ns     |OCP| PE1_TL0.IOdevice.slave: receiving BURST request.
              | S | Data handshake: yes
              | L | MCmd: WR, MAddr: 0x1011, MThreadID: 0x1
              | A | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1

    47 ns     PE#1: task(IO) (write data ready) -> scheduler
    47 ns           synchronizer: READY from IO task
    47 ns     PE#1: scheduler (fetch data from SLAVE) -> task(IO)

    55 ns     |OCP| PE0_TL0.IOdevice.master: Request completed

    55 ns     PE#0: task(IO) (IO task finished) -> scheduler
    55 ns           synconizer: releases task(2,1)
    55 ns     PE#0: scheduler (run) -> task(2,1)

    56 ns     |OCP| soc_comm.tl0_io_b.master: Request completed
    57 ns     PE#1: task(1,1) (external task finished) -> scheduler
    57 ns           synconizer: releases task(3,1)

    57 ns     PE#1: task(IO) (IO task finished) -> scheduler
    57 ns     PE#1: scheduler (run) -> task(3,1)

    74 ns     PE#0: task(2,1) (finished) -> scheduler
    74 ns           synconizer: releases task(4,1)
    74 ns     PE#0: scheduler (run) -> task(4,1)

    76 ns     PE#1: task(3,1) (finished) -> scheduler
    76 ns           synconizer: releases task(5,1)
    76 ns     PE#1: scheduler (run) -> task(5,1)

  1571 ns     PE#0: task(4,1) (finished) -> scheduler
  1571 ns           synconizer: releases task(6,1)
  1571 ns     PE#0: scheduler (run) -> task(6,1)

  1573 ns     PE#1: task(5,1) (finished) -> scheduler
  1573 ns           synconizer: releases task(7,1)
  1573 ns     PE#1: scheduler (run) -> task(7,1)

  2138 ns     PE#0: task(6,1) (finished) -> scheduler
  2140 ns     PE#1: task(7,1) (request to NoC: task(8,1),addr=0x71,dataUnits=10)-> adaptor
  2140 ns     PE#1: scheduler (start NoC write request) -> task(IO)

  2140 ns     |OCP| PE1_TL0.IOdevice.master: sent BURST request.
              | M | Data handshake: yes
              | A | MCmd: WR, MAddr: 0x71, MThreadID: 0x1
              | S | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


  2141 ns     |OCP| soc_comm.tl0_io_b.slave: receiving BURST request.
              | S | Data handshake: yes
              | L | MCmd: WR, MAddr: 0x71, MThreadID: 0x1
              | A | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


  2141 ns     |OCP| soc_comm.tl0_io_a.master: sent BURST request.
              | M | Data handshake: yes
              | A | MCmd: WR, MAddr: 0x71, MThreadID: 0x1
              | S | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


  2142 ns     |OCP| PE0_TL0.IOdevice.slave: receiving BURST request.
              | S | Data handshake: yes
              | L | MCmd: WR, MAddr: 0x71, MThreadID: 0x1
              | A | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1

  2142 ns     PE#0: task(IO) (write data ready) -> scheduler
  2142 ns           synchronizer: READY from IO task
  2142 ns     PE#0: scheduler (fetch data from SLAVE) -> task(IO)

  2150 ns     |OCP| PE1_TL0.IOdevice.master: Request completed

  2150 ns     PE#1: task(IO) (IO task finished) -> scheduler
```

```
 2151 ns     |OCP| soc_comm.tl0_io_a.master: Request completed
 2152 ns     PE#0: task(7,1) (external task finished) -> scheduler
 2152 ns           synconizer: releases task(8,1)

 2152 ns     PE#0: task(IO) (IO task finished) -> scheduler
 2152 ns     PE#0: scheduler (run) -> task(8,1)
 4721 ns     PE#0: task(8,1) (request to NoC: task(9,1),addr=0x1081,dataUnits=10)-> adaptor
 4721 ns     PE#0: scheduler (start NoC write request) -> task(IO)

 4721 ns     |OCP| PE0_TL0.IOdevice.master: sent BURST request.
             | M | Data handshake: yes
             | A | MCmd: WR, MAddr: 0x1081, MThreadID: 0x1
             | S | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


 4722 ns     |OCP| soc_comm.tl0_io_a.slave: receiving BURST request.
             | S | Data handshake: yes
             | L | MCmd: WR, MAddr: 0x1081, MThreadID: 0x1
             | A | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


 4722 ns     |OCP| soc_comm.tl0_io_b.master: sent BURST request.
             | M | Data handshake: yes
             | A | MCmd: WR, MAddr: 0x1081, MThreadID: 0x1
             | S | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1


 4723 ns     |OCP| PE1_TL0.IOdevice.slave: receiving BURST request.
             | S | Data handshake: yes
             | L | MCmd: WR, MAddr: 0x1081, MThreadID: 0x1
             | A | MBurstSeq: UNKN, MBurstLength: 10, MBurstSingleReq: 1

 4723 ns     PE#1: task(IO) (write data ready) -> scheduler
 4723 ns           synchronizer: READY from IO task
 4723 ns     PE#1: scheduler (fetch data from SLAVE) -> task(IO)

 4731 ns     |OCP| PE0_TL0.IOdevice.master: Request completed

 4731 ns     PE#0: task(IO) (IO task finished) -> scheduler
 4731 ns           synconizer: releases task(10,1)
 4731 ns     PE#0: scheduler (run) -> task(10,1)

 4732 ns     |OCP| soc_comm.tl0_io_b.master: Request completed
 4733 ns     PE#1: task(8,1) (external task finished) -> scheduler
 4733 ns           synconizer: releases task(9,1)

 4733 ns     PE#1: task(IO) (IO task finished) -> scheduler
 4733 ns     PE#1: scheduler (run) -> task(9,1)

 4837 ns     PE#0: task(10,1) (finished) -> scheduler
 4837 ns           synconizer: releases task(12,1)
 4837 ns     PE#0: scheduler (run) -> task(12,1)

 4839 ns     PE#1: task(9,1) (finished) -> scheduler
 4839 ns           synconizer: releases task(11,1)
 4839 ns     PE#1: scheduler (run) -> task(11,1)

 5701 ns     PE#0: task(12,1) (finished) -> scheduler
 5701 ns           synconizer: releases task(14,1)
 5701 ns     PE#0: scheduler (run) -> task(14,1)

 5703 ns     PE#1: task(11,1) (finished) -> scheduler
 5703 ns           synconizer: releases task(13,1)
 5703 ns     PE#1: scheduler (run) -> task(13,1)

11696 ns     PE#0: task(14,1) (finished) -> scheduler
11696 ns           synconizer: releases task(16,1)
11696 ns     PE#0: scheduler (run) -> task(16,1)

11698 ns     PE#1: task(13,1) (finished) -> scheduler
11698 ns           synconizer: releases task(15,1)
11698 ns     PE#1: scheduler (run) -> task(15,1)

22773 ns     PE#0: task(16,1) (finished) -> scheduler

22775 ns     PE#1: task(15,1) (finished) -> scheduler

22775 ns     Task graph 0 completed. Now preparing for a new cycle:
             Restoring relation matrix
             Unblocking task (1,1)
             Unblocking task (2,1)
             Unblocking task (3,1)
             Unblocking task (4,1)
```

```
               Unblocking task (5,1)
               Unblocking task (6,1)
               Unblocking task (7,1)
               Unblocking task (8,1)
               Unblocking task (9,1)
               Unblocking task (10,1)
               Unblocking task (11,1)
               Unblocking task (12,1)
               Unblocking task (13,1)
               Unblocking task (14,1)
               Unblocking task (15,1)
               Unblocking task (16,1)
TASK ADDRESS MAP
   PE#0 :
   Task(  1, 1) :
   Task(  2, 1) :
   Task(  4, 1) :
   Task(  6, 1) :
   Task(  8, 1) : 0x71
   Task( 10, 1) :
   Task( 12, 1) :
   Task( 14, 1) :
   Task( 16, 1) :

   PE#1 :
   Task(  3, 1) : 0x1011
   Task(  5, 1) :
   Task(  7, 1) :
   Task(  9, 1) : 0x1081
   Task( 11, 1) :
   Task( 13, 1) :
   Task( 15, 1) :


PE UTILIZATION
   PE#0 : 75.7967% (22739@30000)
   PE#1 : 67.0833% (20125@30000)

IO TASK USAGE
   PE#0 : 0.131932% (30@22739)
   Write data TX    : 0.0879546% (20@22739)
   Write data RX    : 0.0439773% (10@22739)

   PE#1 : 0.149068% (30@20125)
   Write data TX    : 0.0496894% (10@20125)
   Write data RX    : 0.0993789% (20@20125)
```