SAE Aerospace
An SAE International Group

AEROSPACE STANDARD

SAE AS5506A

Issued 2009-12

Superseding AS5506

ARINC 653 annex

RATIONALE

The SAE AS5506A standard is the core language document for version 2 of the Architecture Analysis and Design Language (AADL). This second revision has special features for defining virtual processors and virtual buses that provide stronger abstractions for defining layered and partitioned architectures. ARINC 653 is a common architectural style used in the aviation. The modeling of ARINC 653 architectures provides opportunities for multiple forms of analysis and for code generation. This annex defines a common approach to describe ARINC 653 architectures for model integration, quantitative analysis and as input to ARINC 653 code generators.

This Architecture Analysis & Design Language (AADL) ARINC 653 annex document was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division.

MAJOR CHANGES

There is the list of major changes:

- Define common terms
- New properties for specifying module schedule
- New properties for specifying Health-Monitoring concerns
- Reduce annex-specific properties, use more core properties

SAE Technical Standards Board Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be reaffirmed, revised, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2008 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: **Tel: 877-606-7323 (inside USA and Canada)**
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: custsvc@sae.org

SAE WEB ADDRESS: http://www.sae.org

TABLE OF CONTENTS

A.1 Rationale ... 3
A.2 ARINC 653 partition management (ARINC 653 Module) ... 4
A.3 ARINC 653 partitions modeling ... 5
A.4 Multi processors architectures ... 7
A.5 ARINC 653 processes modeling ... 8
A.6 ARINC 653 inter-partition communication modeling ... 10
A.7 ARINC 653 intra-partition communication modeling ... 12
A.8 ARINC 653 buffers modeling ... 12
A.9 ARINC 653 blackboards modeling ... 13
A.10 ARINC 653 events modeling ... 14
A.11 ARINC 653 semaphores modeling ... 15
A.12 ARINC 653 memory requirements modeling ... 15
A.13 ARINC Health Monitor (HM) modeling ... 17
A.14 ARINC 653 modes/states modeling ... 18
A.15 ARINC 653 application-specific Hardware & Device Drivers ... 18
A.16 Summary of modeling rules ... 20
A.17 ARINC 653 Property Set ... 22

APPENDIX B INFORMATIVE SECTION ... 26

B.1 System validation using the ARINC 653 annex ... 26
B.2 Example with one module ... 27
B.3 Example with two modules ... 34

ARINC 653 Annex

A.1 Rationale

(1) This annex has been defined to support the modeling, analysis and automated integration of ARINC 653 and derived or similar partitioned architectures. It provides AADL architectural style guidelines and AADL defined ARINC 653 oriented properties to define a common approach to use AADL standardized components to express ARINC 653 architectures. Without the annex defined framework, modelers would need to define their own AADL ARINC 653 oriented properties and select their own approach to representing ARINC 653 with AADL components.

(2) This annex was made to help the system designers in the modeling of partitioned architectures, especially ARINC 653 compliant systems. AADL models can be checked and verified using various tools. Consequently, the modeling of partitioned architectures will help system designers to verify and validate their models.This document does not provide any guidance for the verification of AADL models except to provide a means for common specification.

(3) By providing a common framework for partitioned system expression, distributed development and common analysis tools are supported. Code generators to auto integrate ARINC 653 systems based on AADL ARINC 653 annex compliant models are supported. The code generation annex of the AADL provides sufficient information to map most of AADL/ARINC 653 modeling patterns (processes, inter and intra communication channels, shared data) into C or Ada code. However, configuration of ARINC 653 operating systems is not detailed in the code generation annex.

(4) The ARINC 653 standard contains several parts, defining required and extended services. This document provides the mapping between the AADL and the required services of the ARINC 653 standard.

(5) The current document provides a mapping for services defined in the ARINC 653 PART1 standard. Mapping of other services are beyond the scope of this annex.

(6) The avionics-specific terms used in the annex are defined below:

a. **Integrated Modular Avionics**: A shared set of flexible, reusable, and interoperable hardware and software resources that, when integrated, form a platform that provides services, designed and verified to a defined set of safety and performance requirements, to host applications performing aircraft functions.

b. **Module**: A component or collection of components that may be accepted by themselves or in the context of an IMA system. A module may also comprise other modules. A module may be software, hardware, or a combination of hardware and software, which provides resources to the IMA system hosted application.

c. **Application**: Software and/or application-specific hardware with a defined set of interfaces that when integrated with a platform(s) performs a function.

d. **Application software**: The part of an application implemented through software. It may be allocated to one or more partitions.

e. **Partition**: An allocation of resources whose properties are guaranteed and protected by the platform from adverse interaction or influences from outside the partition.

f. **Application-specific hardware**: Hardware dedicated to one application.

g. **Cabinet**: A physical package containing one or more IMA components or modules, that provides partial protection from environmental effects (shielding) and may enable installation and removal of those component(s) or module(s) from the aircraft without physically altering other aircraft ystems or equipment.

h. **Core Software**: The operating system and support software that manage platform resources to provide an environment in which an application can execute.

i. **Component**: A self-contained hardware or software part, database, or combination thereof that may be configuration controlled.

A.2 ARINC 653 partition management (ARINC 653 Module)

(7) In ARINC 653, partitions are managed by the core software with a dedicated kernel that ensures time and space isolation. It schedules partitions using a static timeline scheduling algorithm repeated at a given rate, the major time frame. Each partition has at least one time frame to execute its tasks (called processes in the ARINC 653 standard).

(8) The ARINC 653 core software and its associated physical processor core are modeled with AADL using the `processor` component. This approach is consistent with the AADL concept of a processor to include the operating evironment. The `processor` component is used to specifiy partition management properties which express its requirements.

a. The AADL `processor` component models the ARINC 653 core software. ARINC 653 modules contain partitions. In AADL, these partitions are modeled with AADL `virtual processor` components that are either contained in or bound to an AADL `processor` component. These `virtual processors` subcomponents model partitions runtimes.

b. ARINC 653 module time slots are modeled with the `ARINC653::Module_Schedule` property attached to an AADL `processor` component. The property is a list that defines window schedules. List elements contains the following attributes:

i. `Duration`: The time slot duration.

ii. `Partition`: reference to the partition (`virtual processor`) associated with this slot.

iii. `Periodic_Periodic_Start`: specifies if all periodic tasks should start at the beginning of the partition execution.

c. The `ARINC653::Module_Major_Frame` property is associated with an AADL `processor` component and specifies the major time frame of an ARINC 653 module.

d. The Process_Swap_Execution_Time property from the core language specifies the time needed to switch from one partition to another. The value represents the time required to clean a partition and activate another (cache flush, memory segments change).

e. The ARINC653::Module_Version and ARINC653::Module_Identifier properties aim at adding a version to the module (potentially similar to the one used in the system configuration) as well as a textual description or other comments.

f. The ARINC 653 health monitoring properties usable at the module level are covered in the health monitoring section (A.13).

ARINC 653 entity	AADL entity	Properties
Module	Processor	• ARINC653::Module_Major_Frame • ARINC653::Module_Schedule • ARINC653::HM_Error_ID_Levels • ARINC653::HM_Error_ID_Actions • ARINC653::Module_Version • ARINC653::Module_Identifier • Process_Swap_Execution_Time

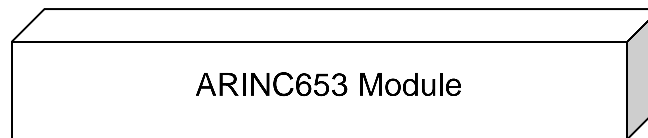


Figure 1 – Graphic representation of an ARINC 653 module (without partition runtime)

A.3 ARINC 653 partitions modeling

(9) An ARINC 653 partition conceptually consists of a separated address space and a specific runtime. This runtime manages resources within the partition and schedules ARINC 653 processes (that correspond to AADL thread components) that are executed in its address space. An ARINC 653 partition hosts the application software to be executed.

(10) Each ARINC 653 partition is represented by an AADL process component component bound to a virtual processor and one or several memory components. In AADL, process components are used to indicate address space protection for threads. The AADL process component and its association to a memory component model the partition address space. The AADL virtual processor component models the partition specific runtime environment provided by the core software.

a. The memory requirements of the ARINC 653 partition are specified by adding the `Data_Size` and `Code_Size` properties. They are added to the AADL `process` component that models the partition. In addition, `process` components are bound to the physical memory using the AADL `Actual_Memory_Binding` property.

b. The `virtual processor` component describes the partition-level scheduler and runtime requirements using AADL properties. Within each AADL `virtual processor` component, the property `Scheduling_Protocol` defines the scheduling policy used inside each ARINC 653 partition.

c. The `Actual_Processor_Binding` AADL property associates the components (`virtual processor` and `process`) that represent a partition. Each AADL `process` component (that contains the application software components) must be bound to an AADL `virtual processor` (that specified the partition runtime provided by the core software).

d. Partitions space isolation is specified by associating an AADL `process` component with one or several AADL `memory` components with the `Actual_Memory_Binding` property.

e. AADL `virtual processor` components (partition runtime) must be contained in or bound to AADL `processor` components (core software). In case one partition (AADL `virtual processor`) can be associated with several core modules (AADL `processor`), users must associate them using the property `Actual_Processor_Binding`.

f. The `Activate_Entrypoint_Source_Text` property specifies the name of a subprogram used to initialized the partition.

g. The Development Assurance Level (DAL) of the application software executed within a partition the `ARINC653::DAL` on its associated `virtual processor` component. It represents the DAL is the application software executed by the partition, not the DAL of the core software. Thus, all the software components contained in a partition should have a DAL value greater or equal to the one of the partition.

h. The `Thread_Swap_Execution_Time` specifies the potential cost of process switching inside a partition. When partition-level scheduler switches from one process to another, there is a potential switching time that should be taken in account for analysis. This property was designed to specify this overhead time so that system designers can specify scheduler overhead for each partition.

i. The `ARINC653::Error_Handling` property specifies the ARINC 653 process (AADL `thread` component) used to recover error raised at the partition level.

j. The `ARINC653::Partition_Id` defines an identifier that potentially corresponds to the one used by the underlying Operating System. The `ARINC653::Partition_Name` defines a partition name with a string that potentially uses natural language.

k. The `ARINC653::System_Partition` property indicates if the partition is an ARINC 653 partition. A system partition is allowed to perform some specific operations (such as processing input/output)

l. The ARINC 653 health monitoring properties usable at the ARINC 653 partition level are covered in the health monitoring section (A.13).

ARINC 653 entity	AADL entities	Properties
Partition	`Virtual Processor`	• Scheduling_Protocol • Activate_Entrypoint_Source_Text • ARINC653::DAL • ARINC653::Partition_Name • ARINC653::Partition_Identifier • ARINC653::System_Partition • ARINC653::Error_Handling • ARINC653::HM_Error_ID_Actions • Thread_Swap_Execution_Time
	`Process`	• Data_Size • Code_Size

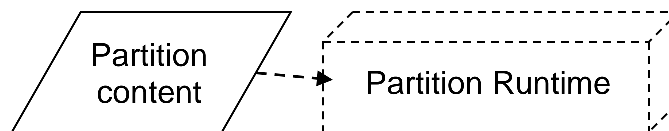


Figure 2 - Graphic representation of a partition and its association to partition runtime

A.4 Multi processors architectures

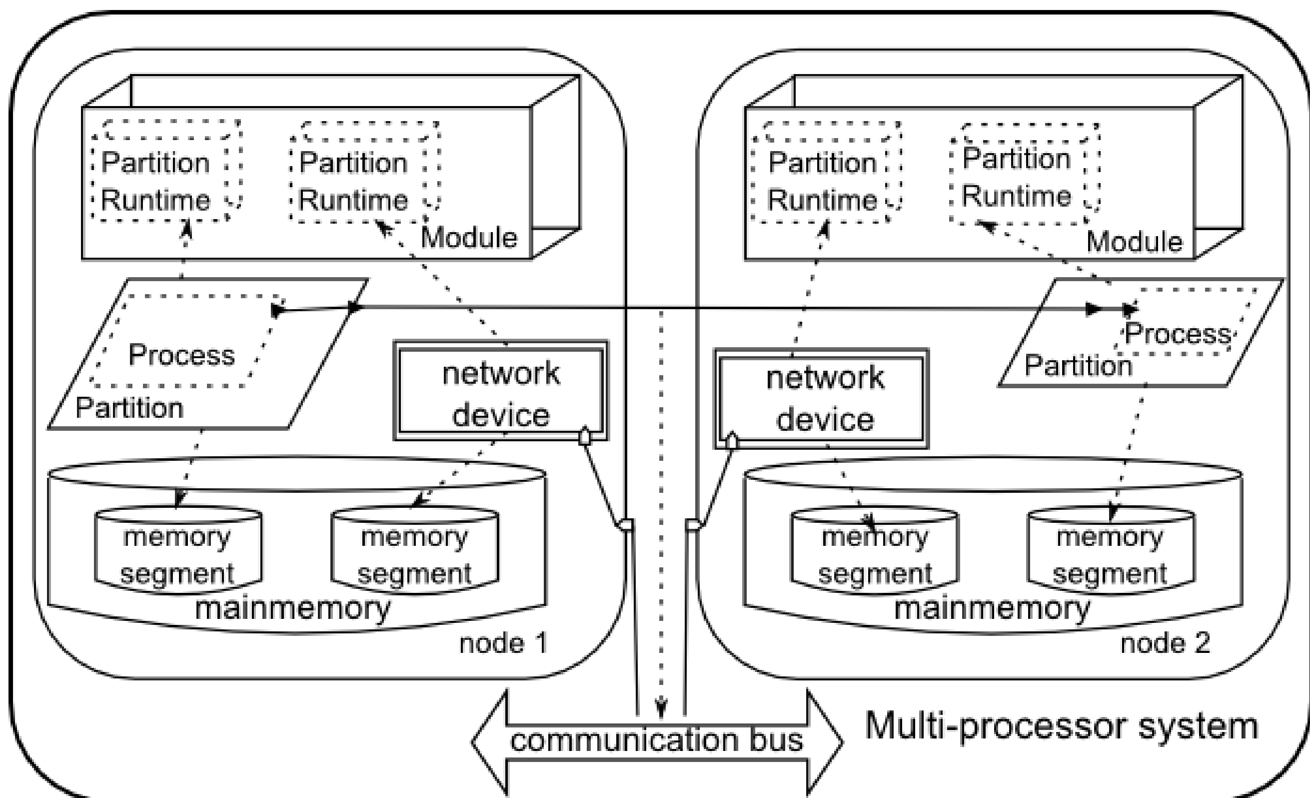
(11) Multi-processor partitioned systems in ARINC 653 are represented by multiple physical processors, each with an ARINC 653 module defined to schedule the partitions on that processor. The AADL system component is a hierarchical component to integrate software and hardware components. It is used to model the multi-processor system with its partitions.

(12) Each node of the multi-processor is specified within an AADL `system` component. This top system component contains AADL `processor`, `virtual processor` and `process` with at least: one AADL `processor` (the ARINC module), one AADL `virtual processor` (the partition-level scheduler), one AADL `process` (the partition address space), one AADL `memory`

component to bind it to, and one AADL `thread` within the AADL `process` component (the ARINC 653 process executed in a partition).

(13) A higher level AADL `system` component is used to contain multiple system components reflecting a multiprocessor system.

(14) Communication between ARINC 653 modules is modeled with event data ports and data ports. Inter-partitions communications through different processors. See sections A.6 for modeling of inter-partitions communication channels)



A.5 ARINC 653 processes modeling

(15) ARINC 653 processes execute the application software that consists in code in their partition's address space. Each ARINC 653 process is associated with a set of requirements: entry point, stack size, period, priority, time capacity and deadline type. List of relevant properties properties are listed in the table below.

(16) ARINC 653 processes are mapped to AADL with the AADL `thread` component.

a. The ARINC 653 base priority concept is mapped to the AADL `Priority` property from the standard AADL property set. The meaning of priority is dependent on the scheduling algorithm used.

b. The ARINC 653 stack size concept is mapped in AADL using the `Stack_Size` property from the standard property set.

c. The ARINC 653 entrypoint concept is mapped in AADL using the `Initialize_Entrypoint` property from the standard property set.

d. The ARINC 653 period concept is mapped in AADL using the `Period` property from the standard property set.

e. The ARINC 653 deadline concept is mapped in AADL using the `Deadline` property from the standard property set.

f. The ARINC 653 time capacity concept is specified with AADL using the `ARINC653::Time_Capacity` property from the AADL standard property set

g. The ARINC 653 deadline type concept (soft or hard) is specified with the AADL enumeration `Deadline_Type` associated to an AADL thread. The value can be either `soft` or `hard`.

h. The ARINC 653 health monitoring properties usable at the ARINC 653 process level are covered in the health monitoring section (A.13).

i. The core AADL property `Dispatch_Protocol` specifies the type of process (periodic, aperiodic, sporadic).

ARINC 653 entity	AADL entity	Properties
Process	`Thread`	• ARINC653::HM_Error_ID_Actions • Code_Size • Data_Size • Heap_Size • Stack_Size • Initialize_Entrypoint • Dispatch_Protocol • Compute_Execution_Time • Deadline • Period • Priority • Deadline_Type • Time_Capacity

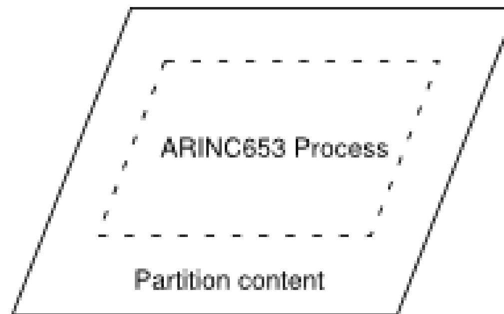


Figure 3 - Graphic representation of an ARINC 653 process inside a partition

A.6 ARINC 653 inter-partition communication modeling

(17) In ARINC 653, inter-partition communication represents data exchange across partitions (one partition sends data to other partitions). The ARINC 653 standard defines two kinds of inter-partition communication: queuing ports and sampling ports. ARINC 653 queuing ports store and queue each instance of data so that receivers can read each queued data element. ARINC 653 sampling ports keep the most recent value (latest data instance replaces prior values). Both ARINC 653 queuing and sampling ports have timing requirements (refresh of data, queuing policy, etc.).

(18) ARINC 653 inter-partition communications are specified using AADL `ports` connected between AADL `process` components.

(19) Identifiers for ARINC 653 inter-partitions may be defined in AADL in one of two ways:

a. Identifiers of the ARINC 653 sampling and queuing ports are derived from the name of the AADL `features` inside the AADL `process` component.

b. Identifiers are explicitly defined by the system designer using the AADL `Source_Name` property. In that case, the `Source_Name` property is added on AADL `data ports` or `event data ports`.

(20) ARINC 653 sampling ports are specified using AADL `data ports` inside an AADL `process` component, originating at an AADL `thread` component (ARINC 653 process).

a. The size of ARINC 653 sampling ports (AADL `data ports)` is deduced from the size the AADL `data` component associated with the port. It corresponds to the maximum message size option of an ARINC 653 sampling port.

b. The refresh period of ARINC 653 sampling ports is specified with the `ARINC653::Sampling_Refresh_Period` property. This property is only relevant on AADL `in data port` since the refresh period is only used on the receiver side.

(21) The modeling of ARINC 653 queuing ports is made with the declaration of `event data ports` in an AADL process component,

a. Size of ARINC 653 queuing ports (AADL `event data ports`) is deduced from the size of its associated `data` component and the port's queue size. The size of the queue (number of elements that can be stored) is specified with the property `Queue_Size`. It corresponds to the size used when the port is created.

b. The timeout of ARINC 653 queuing ports is modeled using the `ARINC653::Timeout` property. It is used for sending data (a sender process sending the data may be blocked until there is enough space to store its message) and receiving (a receiver process may be blocked until there is some data to read in the queue) data.

c. The modeling of queuing discipline of ARINC 653 queuing ports is achieved with the `ARINC653::Queueing_Discipline` property. Supplied values for this property are `FIFO` and `By_Priority`.

(22) In ARINC 653, queuing and sampling ports are connected through a channel. A channel connects one source port to at least one destination port (section 2.3.5.1 of the ARINC 653 standard). AADL can connect one source port to several destination ports (section 9.2.2 of the AADL standard). Thus, AADL connections fit with the semantics of ARINC 653 channels. In consequence, the modeling of ARINC 653 channels is specified by connecting one AADL [`event`] `data port` to at least another AADL `data port` or `event data port`.

ARINC 653 entity	AADL entity	Properties
Queuing ports	Connection of `event data ports` between `process` components	• Queue_Size • ARINC653::Queueing_Discipline • ARINC653::Timeout • Source_Name
Sampling ports	Connection of `data ports` between `process` components	• ARINC653::Sampling_Refresh_Period • Source_Name

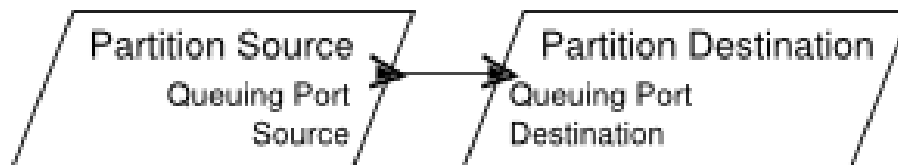


Figure 4 - Graphic representation of an ARINC 653 queuing port connected through two partitions

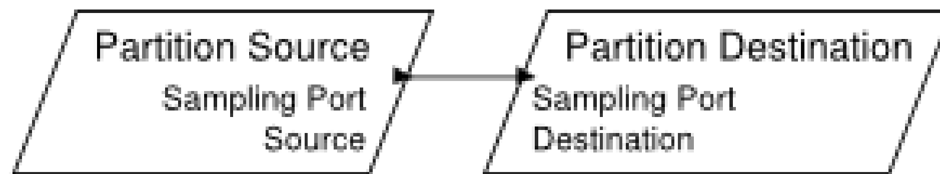


Figure 5 - Graphic representation of an ARINC 653 sampling port connected through two partitions

A.7 ARINC 653 intra-partition communication modeling

(23) In the ARINC 653 standard, processes contained within to the same partition can exchange data using intra-partition communications. The ARINC 653 standard defines four kinds of intra-partition communications: buffer, blackboard, event and semaphore.

(24) ARINC 653 intra-partition communication is specified in AADL models using connections and data accesses between AADL `thread` components. Communication is considered as intra-partition communication as long as it is contained within the AADL `process` and involves only `thread` components located in the same `process`.

(25) ARINC 653 intra-partition channels identifiers may be defined in AADL in one of two ways:

a. The identifier can be deduced from the name of its corresponding AADL entity (shared `data` component, `[event] data port`).

b. System designers can specify the name of the intra-partition channel with the `Source_Name` property. This property is added to AADL ports or data and designate the name of the corresponding ARINC 653 communication channel. By using the the same value in the `Source_Name` property on the same AADL resources, system designers are able to model ARINC 653 communication channels with several readers and writers.

A.8 ARINC 653 buffers modeling

(26) ARINC 653 buffers provide a mechanism to exchange data across ARINC 653 processes located in the same partition. Values are put in a queue and the receiver receives each queued value.

(27) ARINC 653 buffers are modeled in AADL with `event data ports` in `thread` components

a. The size of the queue is deduced by the size of the `data` type and the value of the `Queue_Size` property associated with the AADL port.

b. The AADL `ARINC653::Queueing_Discipline` property indicates what kind of queuing protocol is used. This property is associated with an AADL `event data port`. Supplied values for this property are `FIFO` and `By_Priority`.

ARINC 653 entity	AADL entity	Properties
Buffers	Connection of `event data ports` between threads components located in the same process.	• Queue_Size • ARINC653::Queueing_Discipline • Source_Name

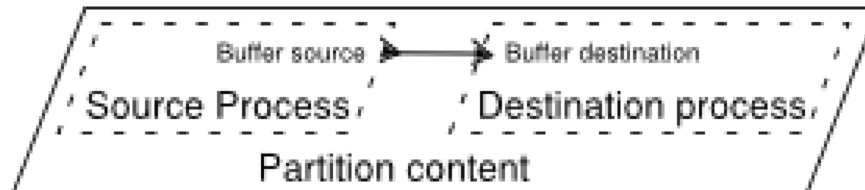


Figure 6 - Graphic representation of an ARINC 653 buffer communication channel

A.9 ARINC 653 blackboards modeling

(28) ARINC 653 blackboards enable data sharing across ARINC 653 processes located in the same ARINC 653 partition. Unlike ARINC 653 buffers, ARINC 653 blackboards do not keep each instance of the data and only store the most recent received value. System designers can specify a timeout when an ARINC 653 process reads a data on a blackboard: readers can wait data until this specified timeout.

(29) ARINC 653 blackboards can be specified with AADL using `data ports` or shared `data` components. The following paragraphs detail both modeling patterns.

(30) ARINC 653 blackboards can be specified with AADL using a shared `data` component across several AADL `thread` components.

a. When a shared data is used to model an ARINC 653 blackboard, the size of the ARINC 653 blackboard is deduced from the size of the shared AADL `data` component.

b. The AADL `ARINC653::Timeout` property is associated with an AADL `data access` to specify the timeout used to read the ARINC 653 blackboard.

c. Modeling ARINC 653 blackboard usage by an ARINC 653 process is achieved by an AADL `data access` feature in an AADL `thread` component.

(31) ARINC 653 blackboards can also be specified with AADL using `data ports`. These AADL `data ports` are connected between AADL `thread` components located in the same partition.

a. When using AADL `data port` to represent ARINC 653 blackboard, the size of the specified ARINC 653 blackboard is deduced from the size of the `data` type associated with the AADL `data port`.

b. The AADL `ARINC653::Timeout` property is associated with the `data port` to model the timeout used to read a blackboard.

c. Modeling ARINC 653 blackboard sharing across several threads is achieved by connecting `data ports` between AADL `thread` components.

ARINC 653 entity	AADL entity	Properties
Blackboard	• `Data` component shared between `thread` components located in the same `process`. • `Data ports` connected between `thread` components located in the same `process`.	• ARINC653::Timeout • Source_Name

Figure 7 - Graphic representation of an ARINC 653 blackboard communication channel

A.10 ARINC 653 events modeling

(32) ARINC 653 events are used to synchronize ARINC 653 processes (AADL thread `components`) located in the same ARINC 653 partition (AADL `process` component). They support control flow between processes by notifying occurrences of conditions to awaiting ARINC 653 processes.

(33) ARINC 653 events are mapped using AADL `event ports` between processes.

a. The `ARINC653::Timeout` property is used on `in event ports` to model timeout used by an ARINC 653 process (AADL `thread` component) when it is waiting for the event.

ARINC 653 entity	AADL entity	Properties
Events	Connection of `event ports` between `thread` components located in the same process.	• ARINC653::Timeout • Source_Name

Figure 8 - Graphic representation of an ARINC 653 event communication channel

A.11 ARINC 653 semaphores modeling

(34) ARINC 653 semaphores synchronize ARINC 653 processes located in the same partition and provide controlled access to resources. Unlike ARINC 653 events, ARINC 653 semaphores can have a queuing discipline that defines queuing mechanisms for waiting processes.

(35) In ARINC 653, semaphores are used to protect resources from concurrent access. In AADL, resources are modeled using the `data` component. The use of shared data is achieved using `data access` features. The use of a semaphore is modeled using the property `Concurrency_Control_Protocol` with the value set to `Protected_Access`.

a. AADL shared `data` components that use ARINC 653 semaphores to avoid concurrent access must be contained in a process component in order to be shared with several threads. The access to the data is achieved with AADL `data access` features.

b. The AADL `ARINC653::Queueing_Discipline` property indicates what kind of queuing protocol is used to dispatch tasks waiting on the semaphore. This property is associated with an AADL `data access`. Supplied values for this property are `FIFO` and `By_Priority`.

ARINC 653 entity	AADL entity	Properties
Semaphore	Data inside `process` and `data access` for each `thread` that needs it.	• ARINC653::Timeout • ARINC653::Queueing_Discipline • Source_Name

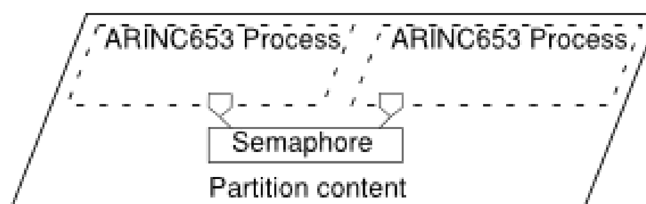


Figure 9 - Graphic representation of an ARINC 653 semaphore usage across two ARINC 653 processes

A.12 ARINC 653 memory requirements modeling

(36) System designers may want to model partitions isolation in different memory segments. To do that, `memory` components that represent physical memory contain several `memory` subcomponents (add other `memory` components as subcomponents). In this hierarchy of AADL `memory` components, the root `memory` component models the hardware memory whereas the AADL `memory` subcomponents model logical separation of the memory (memory segments where partitions store code and data).

(37) Requirements of `memory` component that models memory segments are specified with AADL using the `Base_Address` property (which correspond to the base address of the segment in the ARINC 653 architecture) and the `Memory_Size` property (which correspond to the size of the word in this memory component).

(38) Each AADL `process` component (part of the modeling of ARINC 653 partitions) is associated (bound) with an AADL `memory` component using the `Actual_Memory_Binding` property from the standard property set. It specifies the deployment of ARINC 653 partitions (AADL `process` component) on physical memory (AADL `memory` component).

(39) The ARINC 653 standard uses space isolation across partitions. Consequently, the binding rule between partitions and memory implies that an AADL `memory` component is bounded to exactly one partition (AADL `process`).

(40) ARINC 653 defines three memory access types: read, write and execute. ARINC 653 memory accesses and permissions are specified with AADL by associating the `Memory_Protocol` property with an AADL `memory` component.

(41) ARINC 653 requires that memory content is explicitly specified (if a memory contains code, data or both). This requirement is specified by adding the `ARINC653::Memory_Kind` property on an AADL `memory` component.

ARINC 653 entity	AADL entity	Properties
Memory Requirements	Describe memory requirements for `process` components and specify the allocation of partitions (AADL `process` component) on hardware memory (AADL `memory` component).	• Actual_Memory_Binding • Memory_Size • Base_Address • Memory_Protocol • ARINC653::Memory_Kind

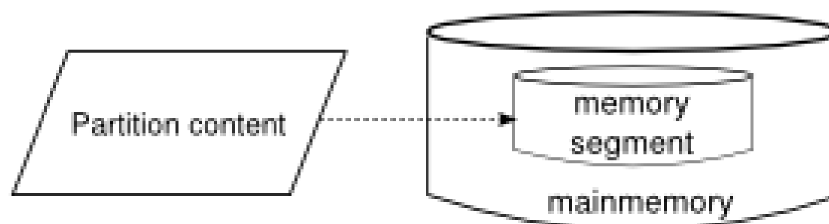


Figure 10 - Graphic representation of memory segments inside the main memory and process binding to memory segments

A.13 ARINC Health Monitor (HM) modeling

(42) In ARINC 653, errors can be managed at different levels: module level, partition level and process level. These levels correspond to the following AADL components: `processor`, `virtual processor` and `thread`. Moreover, this is the responsibility of the system integrator to indicate which error is handled at each level. The actual ARINC 653 enumerates faults that can be raised in each level of an ARINC 653 architecture.

(43) The AADL property `ARINC653::HM_ID_Levels` specifies the list of all potential errors that can be triggered within an ARINC 653 modules (AADL `processor`). This list contains information for these errors using an AADL record type (HM_Error_Level_Type) that specifies

a. the error identifier (using the `ErrorIdentifier` field)

b. the error description (using the `Description` field)

c. the containment level of the error (`Module_Level`, `Partition_Level` or `Processor_Level` – as defined by the `Error_Level_Type` AADL type.)

d. the ARINC 653 related error code (`ErrorCode`) specified by an `ARINC653::Supported_Error_Code` value.

(44) The AADL property `ARINC653::HM_Error_ID_Actions` specifies health-monitoring actions at each level. This property is added on AADL `processor` (ARINC 653 module level), AADL `virtual processor` (ARINC 653 partition) or AADL `thread` (ARINC 653 process). The `ARINC653::HM_Error_ID_Actions` specifies a list that contains informations for error handling:

a. the error identifier (using the `ErrorIdentifier` field) related to the action. This value is a reference to an existing identifier used by the `ARINC653::HM_ID_Levels` property

b. the action description (using the `Description` field) that details the action using natural language.

c. the undertaken action performed (using the `Action` field) to recover the error. This can be a textual description or a reference to the name of the procedure provided by the underlying ARINC 653 runtime.

(45) The ARINC 653 standard states that list of errors and actions are defined by the operating system provider. In consequence, users of the ARINC 653 annex can redefine the list of possible errors in the property set as in the property set values. To do so, users can extend the list of errors specified by the AADL type `ARINC653::Supported_Error_Code`.

(46) In addition, system designers may want to model errors detection and recovery mechanisms to analyze error impact or fault propagation. Modeling of faults or errors and their impact is beyond the scope of this annex. However, errors and recovery procedures can be specified using the Error Modeling Annex of the AADL. This annex provides a suitable semantics to model errors and their propagation in a layered architecture.

A.14 ARINC 653 modes/states modeling

(47) The ARINC 653 standard describes four operational modes for partitions (COLD START, WARM START, NORMAL and IDLE) as well as several states for the system (Module Init, Module Function, Partition Init, etc.). This list of states might be extended on a user and implementation basis.

(48) ARINC 653 modes and states could be mapped to AADL `modes`. To do so, the `State_Information` property provides a way to establish a connection between this two concepts. The `State_Information` is added to an AADL model and contains

a. An identifier (field `Identifier`) that references the related ARINC 653 mode identifier

b. A description (field `Description`) that describes the ARINC 653/AADL mode using natural language.

A.15 ARINC 653 application-specific Hardware & Device Drivers

(49) In ARINC 653, application-specific hardware is controlled by device drivers that reside in core software. The device driver can be located either at the kernel or partition level. Therefore, they can be located within main module functionalities or isolated in a distinct partition that has its own resources and services. In ARINC 653, such a partition is called system partition, because it provides dedicated services and functions to communicate with the hardware although it is contrained by time and space partitioning. Consequently, system designers and operating system providers can have different implementation strategies to implement device drivers in ARINC 653 systems.

(50) These implementation strategies impact system functionalities and performances. Our modeling patterns are designed for the specification of these two different implementation strategies so that system designers can precisely describe their system and analyze impact of their implementations strategies.

(51) Application-specific hardware (network interface, sensor, etc.) is specified with the AADL `device` hardware component. On the other hand, the device driver (software that controls the device) is described by adding an AADL `abstract implementation` component and associated with the AADL `device` component using `Device_Driver` property.

(52) The `abstract` component associated with the AADL `device` component contains all the necessary components to control the application-specific hardware. It is composed by AADL components (`thread`, `data`, etc.) with their properties and requirements (timing, memory, etc.). Modeling of device drivers internals describe the underlying operating system and thus, allow system designer to analyze their impact on the overall system (in terms of performance, latency, etc.)

(53) AADL `device` components must be associated with an AADL `processor` (an ARINC 653 module) or an AADL `virtual processor` (an ARINC 653 partition).

a. If the AADL `device` component is bound to an AADL `processor`, it means that the device driver resides in the ARINC 653 module.

b. If the `device` component is bound to an AADL `virtual processor`, it means that the device driver resides in a system partition and uses time and space isolation mechanisms of the partition. In this case, the partition is considered as a system partition.

These two binding mechanisms illustrate the different implementation strategies for device drivers in ARINC 653 systems and thus, ease the analysis of their impact on the overall architecture.

ARINC 653 entity	AADL entity	Properties
Device drivers	AADL `device` component. The associated device driver is described using the `Device_Driver` property of the standard property set. This `device` component is bound to AADL `processor` or AADL `virtual processor` components.	• Device_Driver • Actual_Processor_Binding

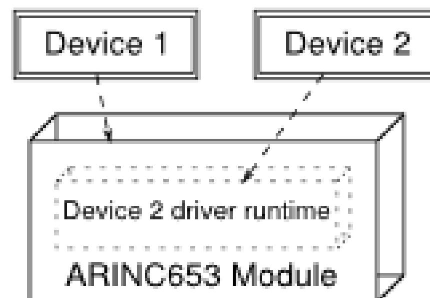


Figure 11 - Graphic representation of device modeling in ARINC 653 architectures with both different binding strategies

A.16 Summary of modeling rules

ARINC 653 entity	AADL entity	Properties
Module	`Processor`	• ARINC653::Module_Major_Frame • ARINC653::Module_Schedule • ARINC653::HM_Error_ID_Levels • ARINC653::HM_Error_ID_Actions • ARINC653::Module_Version • ARINC653::Module_Identifier • Process_Swap_Execution_Time • Scheduling_Protocol
Partition	`Virtual Processor`	• Scheduling_Protocol • ARINC653::DAL • ARINC653::Partition_Name • ARINC653::Partition_Identifier • ARINC653::System_Partition • ARINC653::Error_Handling • ARINC653::HM_Error_ID_Actions • Activate_Entrypoint_Source_Text • Thread_Swap_Execution_Time
	`Process`	• Data_Size • Code_Size
Process	`Thread`	• ARINC653::HM_Error_ID_Actions • Code_Size • Data_Size • Heap_Size • Stack_Size • Initialize_Entrypoint • Compute_Execution_Time • Deadline • Period • Priority • Proprity_Type • Time_Capacity • Dispatch_Protocol
Queuing ports	Connection of `event data ports` between `process` components	• Queue_Size • ARINC653::Queueing_Discipline • ARINC653::Timeout • Source_Name
Sampling	Connection of `data ports`	• ARINC653::Sampling_Refresh_Period

ports	between process components	• Source_Name
Buffers	Connection of event data ports between threads components located in the same process.	• Queue_Size • ARINC653::Queueing_Discipline • Source_Name
Blackboards	• Data component shared between thread components located in the same process. • Data ports connected between thread components located in the same process.	• ARINC653::Timeout • Source_Name
Semaphore	Data contained in a process component and shared between several thread components.	• ARINC653::Timeout • Source_Name • ARINC653::Queueing_Discipline
Events	Connection of event ports between thread components located in the same process.	• ARINC653::Timeout • Source_Name
Memory Requirements	Describe memory requirements for ARINC 653 process (AADL thread components) and specify the allocation of ARINC653 partitions (AADL process component) on hardware memory (AADL memory component).	• Actual_Memory_Binding • Memory_Size • Base_Address • Memory_Protocol • ARINC653::Memory_Kind
Device drivers	AADL device component. The associated device driver is described using the Device_Driver property of the standard property set. This device component is bound to AADL processor or AADL virtual processor components.	• Device_Driver • Actual_Processor_Binding
Mode	AADL mode. The mode can then represent an ARINC 653 mode.	• ARINC653::State_Information

A.17 ARINC 653 Property Set

property set ARINC653 **is**

```
Module_Major_Frame : Time
  applies to ( processor );
```

The `Module_Major_Frame` property specifies the major frame for the ARINC 653 module (AADL `processor` component).

```
Sampling_Refresh_Period : Time
  applies to ( data port );
```

The `Sampling_Refresh_Period` property indicates data arrival rate for an `in data port`. It corresponds to the concept of refresh time of ARINC 653 sampling port.

```
Supported_Error_Code: type enumeration
              (Module_Config, Module_Init, Module_Scheduling,
               Partition_Scheduling, Partition_Config,
               Partition_Handler, Partition_Init, Deadline_Miss,
               Application_Error, Numeric_Error,
               Illegal_Request, Stack_Overflow,
               Memory_Violation, Hardware_Fault, Power_Fail);
```

The `Supported_Error_Code` enumeration corresponds to the possible Error code that can be raised at the different levels of an ARINC 653 architecture (module, partition, process). The list of possible values is implementation dependent and can be modified by the system designer.

```
Supported_Memory_Kind : type enumeration
   (memory_data, memory_code);
```

The `Supported_Memory_Kind` enumeration describes possible content of an AADL `memory` component.

```
Memory_Kind : list of (Supported_Memory_Kind)
  applies to ( memory );
```

The `Memory_Kind` property describes the content of an AADL `memory` component.

```
Timeout : Time
  applies to (event data port,data port,event port,access connection);
```

The `Timeout` property specifies the timeout used by an ARINC 653 process when sending/receiving a data. Depending on which component it is used, it could be useful for sender or receiver side.

```
Supported_DAL_Type : type enumeration
   (LEVEL_A, LEVEL_B, LEVEL_C, LEVEL_D, LEVEL_E);
```

The `Supported_DAL_Type` enumeration corresponds to the different Development

Assurance Levels supported by the ARINC 653 standard.

```
DAL : Supported_DAL_Type
  applies to (virtual processor, process, thread, subprogram);
```

The DAL property defines the Development Assurance Level of a component. It is associated to software component to capture their associated DAL. When applied to a virtual processor, this is a requirement for this partition and all software components associated by this partition must have at least the same or higher DAL value.

```
Module_Version : aadlstring applies to (processor);
```

The Module_Version property adds a description to the ARINC 653 module (textual comments using natural language) .

```
Module_Identifier: aadlstring applies to (processor);
```

The Module_Identifier property specifies the ARINC 653 identifier for a module specifies with an AADL processor component.

```
Partition_Name: aadlstring applies to (virtual processor);
```

The Partition_Name property defines the name for a specific partition with natural language.

```
Partition_Identifier: aadlinteger applies to (virtual processor);
```

The Partition_Identifier property defines the identifier of the partition that potentially corresponds to the one used by the execution platform.

```
System_Partition: aadlboolean applies to (virtual processor);
```

The System_Partition property specifies if a given partition is operating as a system partition (performing I/O or other operations requiring special privileges).

```
Error_Handling: reference (thread) applies to (virtual processor);
```

The Error_Handling property specifies the ARINC 653 process (AADL thread) operating within the partition as the error handler. This ARINC 653 process is then supposed to receive notification of errors and take appropriate actions to recover them.

```
Error_Level_Type: type enumeration
                    (Module_Level, Partition_Level, Process_Level);
```

The Error_Level_Type type lists all potential error levels: in a module (AADL processor), partition (AADL virtual processor) or process (AADL thread).

```
HM_Error_ID_Level_Type: type record (
      ErrorIdentifier : aadlinteger;
      Description     : aadlstring;
      ErrorLevel      : ARINC653::Error_Level_Type;
      ErrorCode       : ARINC653::Supported_Error_Code;);
```

The HM_Error_ID_Level_Type property records all information related to a fault that may occur within a module. ErrorIdentifier is unique and is then re-used by the HM_Error_ID_Actions property. The Description provides a basic description using natural language. The ErrorLevel indicates at which level the error occurs while the

`ErrorCode` designates the related error (such as scheduling error).

```
HM_Error_ID_Levels: list of ARINC653::HM_Error_ID_Level_Type
                                         applies to (processor);
```

The HM_Error_ID_Levels property lists all errors that may occurs within a module using the HM_Error_ID_Level_Type type.

```
HM_Error_ID_Action_Type : type record (
      ErrorIdentifier : aadlinteger;
      Description     : aadlstring;
      Action          : aadlstring;);
```

The `HM_Error_ID_Action_Type` type lists all useful information related to error recovery. The `ErrorIdentifier` is the unique identifier for an error and is one included with the `HM_Error_ID_Levels` property. The `Description` is a textual description of the error using natural language. The `Action` is a string describing the action, potentially the name of a function used in the underlying ARINC 653 runtime to recover the error.

```
HM_Error_ID_Actions: list of ARINC653::HM_Error_ID_Action_Type
                  applies to (processor, virtual processor, thread);
```

The `HM_Error_ID_Actions` is a list of action used to recover an error, either at the module (AADL `processor`), partition (AADL `virtual processor`) or process (AADL `thread`) level.

```
State_Information_Type: type record (
      Identifier      : aadlinteger;
      Description     : aadlstring;);
```

The `State_Information_Type` type contains all information related to an ARINC 653 mode: an `Identifier` (that is potentially similar to the one of the ARINC 653 implementation) and a `Description` that uses natural language.

```
State_Information: ARINC653::State_Information_Type applies to (mode);
```

The `State_Information` property associates an ARINC 653 mode with an AADL `mode`.

```
Queueing_Discipline_Type: type enumeration (Fifo, By_Priority);
```

The `Queueing_Discipline_Type` type lists all potential queueing discipline on ARINC 653 communication mechanisms (such as ARINC 653 queueing ports or ARINC 653 buffers).

```
Queueing_Discipline: ARINC653::Queueing_Discipline_Type
                                    applies to (port);
```

The `Queueing_Discipline` property reflects the queueing discipline of the ARINC 653 runtime for a given port. It indicates the dispatching policy for an ARINC 653 process (AADL `thread` component) waiting for a new value on a port.

```
Schedule_Window : type record (
   Partition : reference (virtual processor, processor);
   Duration  : time;
   Periodic_Processing_Start : aadlboolean;
);
```

The `Schedule_Window` type specifies a time slot of the ARINC 653 module. The `Partition` attribute represents the partition assigned to this slot. The `Duration` attribute capture the time

allocated to the partition. The `Periodic_Processing_Start` specifies if all periodic tasks should start when the partition is activated.

`Module_Schedule : list of ARINC653::Schedule_Window`
`  applies to ( processor, virtual processor );`
The `Module_Schedule` property the schedule time slots in the module. This is a list of all time slots assigned for each partition.

`Deadline_Type : enumeration (soft, hard)`
`  applies to ( thread );`
The `Deadline_Type` property specifies the kind of deadline associated for a task. It corresponds to the same attribute associated to an ARINC 653 process.

`Time_Capacity : Time`
`  applies to ( thread );`
The `Time_Capacity` represents the time allocated to each task.

```
end ARINC653;
```

Appendix B Informative section

This informative section illustrates ARINC 653 systems modeling by providing examples and validation examples.

B.1 System validation using the ARINC 653 annex

(54) The ARINC 653 standard defines an XML-style notation to describe system requirements and deployment of an ARINC 653 architecture.

(55) Here is a partial list of what could be validated from the AADL model:

a. Scheduling of each ARINC 653 module. AADL models describe partitions content, including threads (ARINC 653 processes) and intra-partition communication. Thanks to this finer-grain modeling, the user can verify partition-level scheduling, as well as the scheduling of the overall system.

b. Bus load. With AADL, system designers can precisely specify which connections use a bus. Thus, the user can verify the bus loading and if data might be delayed, due to various issues (overload, …)

(56) In general, modeling ARINC 653 systems with AADL would offer many benefits, because the language models the complete runtime system with its requirements, not just a subset of the system. Thus, using a model-based approach to design ARINC 653 systems with AADL can provide useful verification features.

(57) The definition of the verification rules is beyond the scope of this annex. Readers would refer to the toolset that supports this AADL ARINC 653 annex.

B.2 Example with one module

(58) The following example shows a system with two partitions. It shows the components involved in the modeling of ARINC 653 system and illustrates the mapping of ARINC 653 concepts to the AADL.

a. The first ARINC 653 partition is modeled with the AADL process `partition1_process` (which models the space isolation) bound to the AADL virtual processor `part1` (which models intra-partition runtime).

b. In the same way, the second partition is modeled with the process `partition2_process` and the virtual processor `part2`.

c. The scheduling algorithm used in each partition is specified in the `virtual processor` component that models the runtime within each partition.

d. The main memory (`mem`) is divided into two memory components (`part1em` and `part2em`). Each partition process (`partition1_process` and `partition2_process`) is bound to a memory component.

e. The ARINC 653 module (kernel) schedules the partitions on the processor and therefore provides the runtime at the processor level. In AADL, the `processor` concept includes the runtime environment. Therefore, the ARINC 653 kernel and the physical processor itself are specified with the AADL `processor` component. It contains properties (`ARINC653::Module_Schedule`, `ARINC653::Module_Major_Frame`) that describe scheduling requirements (major time frame, window slots and their allocation).

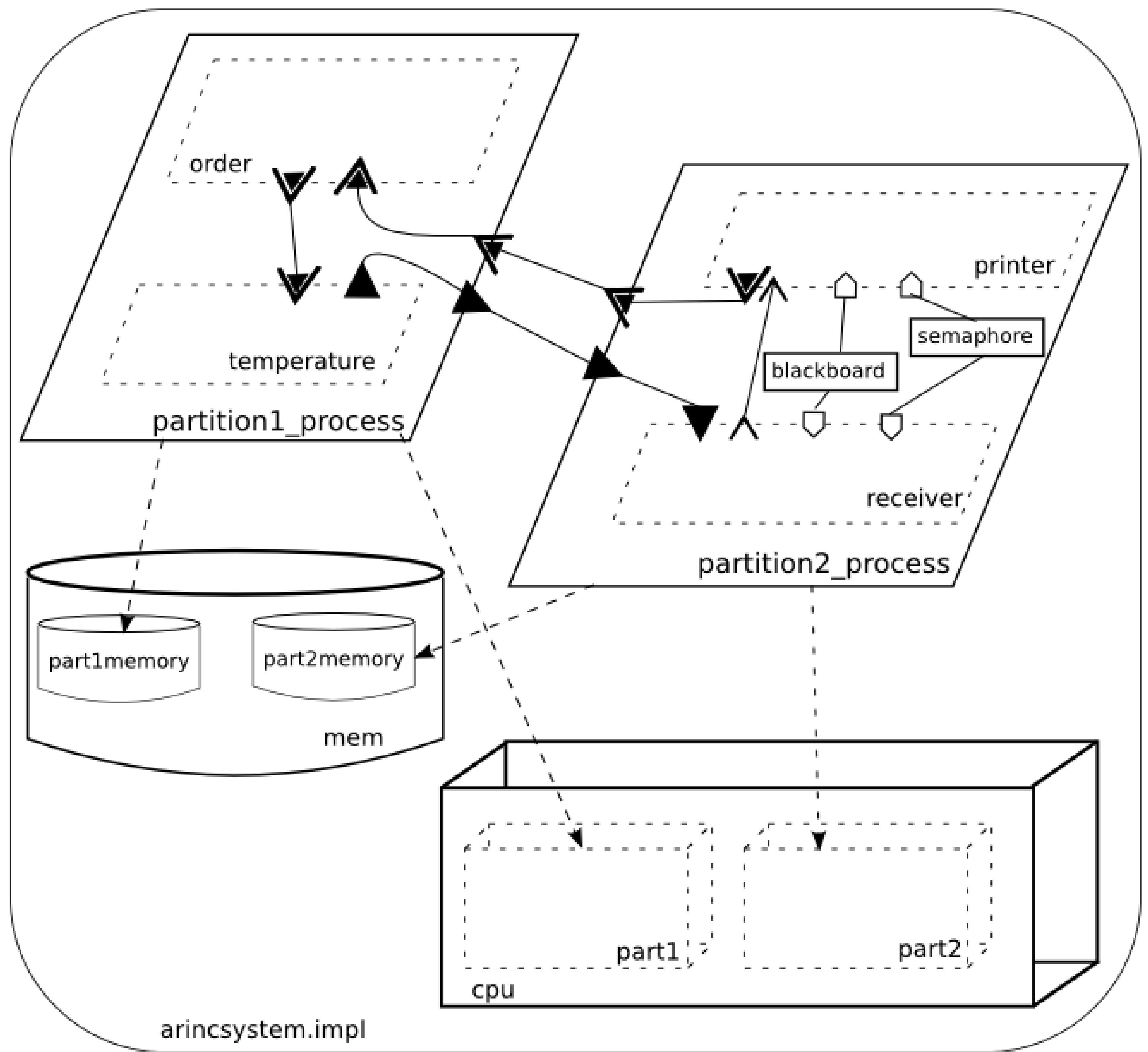
f. In the first partition (`partition1_process`), an intra-partition communication that uses ARINC 653 buffers is specified (AADL `event data ports`). This communication is made between AADL `thread` components representing ARINC 653 processes (order and temperature).

g. In the second partition (`partition2_process`), an intra-partition communication that corresponds to blackboards is added (AADL shared data component). This communication is made between the two threads of the partition (`printer` and `receiver`).

h. In the second partition (`partition2_process`), a communication using AADL `event ports` between two AADL `thread` components (ARINC 653 processes) is added.

i. A shared `data` component between the two threads (`printer` and `receiver`) of the second partition is added. The shared data is protected from concurrent accesses using ARINC 653 semaphores.

(59) There is the graphical representation of our example.



Graphic representation of our example

(60) There is the textual representation of our example. The textual representation includes properties on each component.

```
package arincexample1

public

with ARINC653;

data integer
end integer;

data ordercmd
end ordercmd;

data protected_data
properties
   Concurrency_Control_Protocol => Protected_Access;
end protected_data;


-- Now, declare the virtual processors that model
-- partition runtime.

virtual processor partition1_rt
properties
      Scheduling_Protocol => (RMS);
end partition1_rt;

virtual processor implementation partition1_rt.impl
end partition1_rt.impl;

virtual processor partition2_rt
properties
      Scheduling_Protocol => (RMS);
end partition2_rt;

virtual processor implementation partition2_rt.impl
end partition2_rt.impl;

subprogram  sensor_temperature_spg
end         sensor_temperature_spg;

subprogram  sensor_receiveinput_spg
end         sensor_receiveinput_spg;

subprogram  commandboard_receiveinput_spg
end         commandboard_receiveinput_spg;

subprogram  commandboard_printinfos_spg
end         commandboard_printinfos_spg;


-- Threads for the first partition

thread sensor_temperature_thread
```

```
features
   tempout: out data port integer;
   order: in event data port ordercmd;
properties
   Initialize_Entrypoint =>
classifier (arincexample1::sensor_temperature_spg);
   Priority => 42;
   Stack_Size => 100 Kbyte;
   Period => 20 ms;
   Compute_Execution_Time => 10 ms .. 12 ms;
   Deadline => 40 ms;
end sensor_temperature_thread;

thread implementation   sensor_temperature_thread.impl
end                     sensor_temperature_thread.impl;

thread sensor_receiveinput_thread
features
   commandin: in event data port integer;
   order: out event data port ordercmd;
properties
   Initialize_Entrypoint =>
classifier (arincexample1::sensor_receiveinput_spg);
   Priority => 10;
   Stack_Size => 100 Kbyte;
   Period => 20 ms;
   Compute_Execution_Time => 8 ms .. 10 ms;
   Deadline => 40 ms;
end sensor_receiveinput_thread;

thread implementation   sensor_receiveinput_thread.impl
end                     sensor_receiveinput_thread.impl;

--  Threads for the second partition

thread commandboard_receiveinput_thread
features
   temp: in data port integer;
   tempavg : requires data access integer {ARINC653::Queueing_Discipline => FIFO;};
   newavg: out event port;
   need_semaphore : requires data access protected_data  {ARINC653::Queueing_Discipline => FIFO;};
properties
   Initialize_Entrypoint =>
classifier (arincexample1::commandboard_receiveinput_spg);
   Priority => 42;
   Stack_Size => 100 Kbyte;
   Period => 20 ms;
   ARINC653::Time_Capacity => 7 ms;
   Compute_Execution_Time => 5 ms .. 7 ms;
   Deadline => 40 ms;
end commandboard_receiveinput_thread;

thread commandboard_printinfos_thread
features
   ordersensor: out event data port integer;
   tempavg : requires data access integer  {ARINC653::Queueing_Discipline => FIFO;};
   newavg: in event port;
   need_semaphore : requires data access protected_data  {ARINC653::Queueing_Discipline => FIFO;};
```

```
properties
   Initialize_Entrypoint =>
classifier (arincexample1::commandboard_printinfos_spg);
   Priority => 43;
   Stack_Size => 100 Kbyte;
   Period => 20 ms;
   ARINC653::Time_Capacity => 6 ms;
   Compute_Execution_Time => 2 ms .. 6 ms;
   Deadline => 40 ms;
end commandboard_printinfos_thread;

-- Now, declare process that model partition address space

process partition1_process
features
   queueingin: in event data port integer
{Queue_Size => 4;
                              ARINC653::Timeout => 5ms;
                              ARINC653::Queueing_Discipline => FIFO;};
   --  In the context of a event data port, the ARINC653::Timeout property
   --  is the timeout we used in the APEX functions.
   --  More, the Queue_Size property is used to compute the size of the queue
   --  of the port.
   --  Finally, the ARINC653::Queueing_Discipline indicates how you handle queuing
   --  data and how data are classified in the queue.
   samplingout: out data port integer;
end partition1_process;

process implementation partition1_process.impl
subcomponents
   temperature : thread sensor_temperature_thread.impl;
   order       : thread sensor_receiveinput_thread.impl;
connections
  bufferconnectionexample: port order.order -> temperature.order;
   c1 : port queueingin -> order.commandin;
   c2 : port temperature.tempout -> samplingout;
end partition1_process.impl;

process partition2_process
features
   queueingout: out event data port integer {ARINC653::Timeout => 10ms;};
   --  In the context of a event data port, the ARINC653::Timeout property
   --  is the timeout we used in the APEX functions.
   samplingin: in data port integer
{ARINC653::Sampling_Refresh_Period => 10ms;};
   --  The ARINC653::Timeout apply only to in data port. It is the refresh
   --  period for sampling ports.
end partition2_process;

process implementation partition2_process.impl
subcomponents
   receiver   : thread commandboard_receiveinput_thread;
   printer    :  thread commandboard_printinfos_thread;
   sem        : data protected_data;
   blackboard : data integer;
connections
   --  example of intra-partition communication with data ports (blackboards)
   blackboardconnection1: data access blackboard -> printer.tempavg;
```

```
   blackboardconnection2: data access blackboard -> receiver.tempavg;
   --  example of intra-partition communication with event port (events)
   eventconnectionexample: port receiver.newavg -> printer.newavg;
   c0 : port printer.ordersensor -> queueingout;
   c1 : port samplingin -> receiver.temp;
   c2 : data access sem -> receiver.need_semaphore {ARINC653::Timeout => 20 ms;};
   c3 : data access sem -> printer.need_semaphore {ARINC653::Timeout => 10 ms;};
end partition2_process.impl;

-- Main runtime

processor powerpc
end powerpc;

processor implementation powerpc.impl
subcomponents
   part1:   virtual processor partition1_rt.impl;
   part2:   virtual processor partition2_rt.impl;
properties
   ARINC653::Module_Major_Frame => 50ms;

     ARINC653::Module_Schedule =>
         ( [Partition => reference (part1);
           Duration  => 10 ms;
           Periodic_Processing_Start => false;],
           [Partition => reference (part2);
           Duration  => 10 ms;
           Periodic_Processing_Start => false;],
           [Partition => reference (part1);
           Duration  => 30 ms;
           Periodic_Processing_Start => false;]
         );
end powerpc.impl;


-- Memory
memory partition1_memory
properties
   Base_Address => 0;
   ARINC653::Memory_Type => (Code_Memory);
end partition1_memory;

memory partition2_memory
properties
   Base_Address => 100;
   ARINC653::Memory_Type => (Code_Memory);
end partition2_memory;

memory main_memory
end main_memory;

memory implementation main_memory.impl
subcomponents
   part1mem: memory partition1_memory;
   part2mem: memory partition2_memory;
end main_memory.impl;

system arincsystem
```

```
end arincsystem;

system implementation arincsystem.impl
subcomponents
   mem            : memory main_memory.impl;
   cpu            : processor powerpc.impl;
   partition1_pr  : process partition1_process.impl;
   partition2_pr  : process partition2_process.impl;
connections
   samplingconnection: port partition1_pr.samplingout ->
 partition2_pr.samplingin;
   queueingconnection: port partition2_pr.queueingout ->
partition1_pr.queueingin;
properties
   --  bind partition process to their associated
   --  runtime (virtual processor)
   Actual_Processor_Binding =>
(reference (cpu.part1)) applies to partition1_pr;
   Actual_Processor_Binding =>
(reference (cpu.part2)) applies to partition2_pr;

   --  bind partition process to their address spaces
   --  (memory components)
   Actual_Memory_Binding =>
(reference (mem.part1mem)) applies to partition1_pr;
   Actual_Memory_Binding =>
(reference (mem.part2mem)) applies to partition2_pr;
end arincsystem.impl;


end arincexample1;
```

B.3 Example with two modules

(61) The following example illustrates the modeling of a distributed ARINC 653 system with two modules. This example illustrates the modeling of a communication between two ARINC 653 modules with AADL.

a. In the first module, three partitions are defined. One partition communicates (it sends data) ; the second does not communicate. The third contains the device driver for the network interface. Since the third partition contains a device driver it is considered in ARINC 653 to be a system partition.

b. Device drivers are specified using the Device_Driver property in the textual representation. Driver internal are not represented in the graphic version since there is no standardized way to represent properties in the graphic notation of AADL.

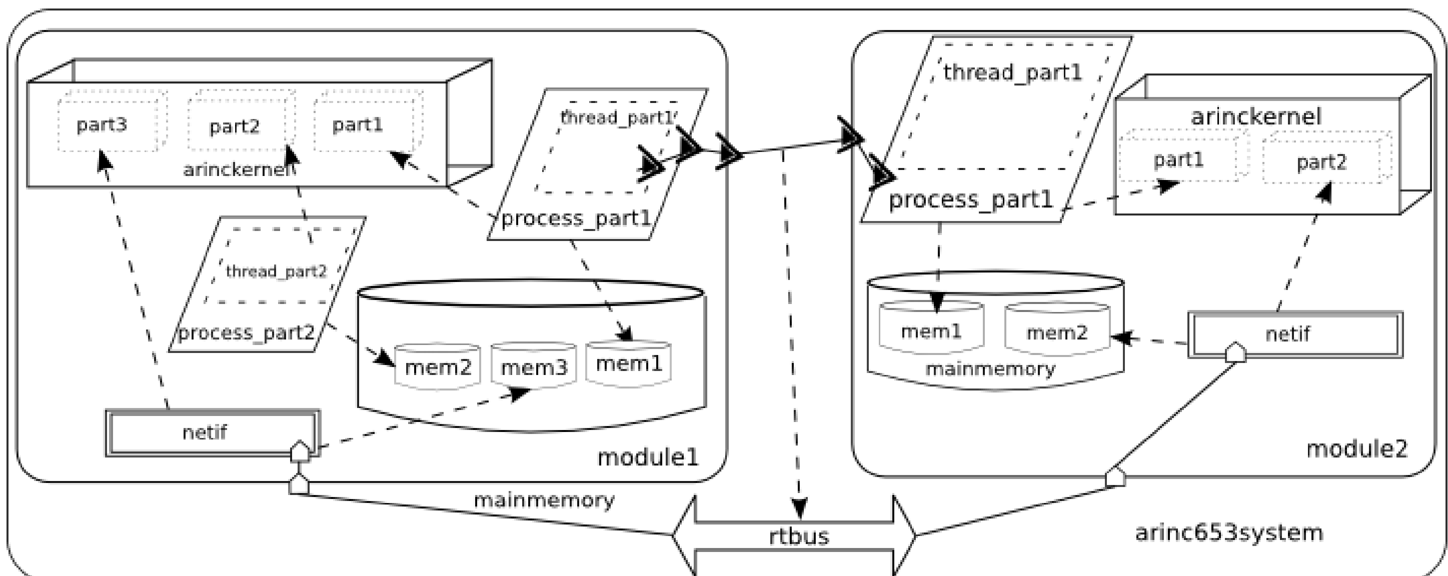
c. Each partition (even system partitions that execute device drivers) is bound to a part of the main memory (modeling of different address spaces).

d. The second module, contains two partitions : one communicates and the second contains the device driver for the network interface.

e. These two ARINC 653 modules communicate through AADL event data ports. It maps the concept of ARINC 653 queueing ports between two ARINC653 modules.

f. Notice that the ARINC 653 modules communicate across a bus named `rtbus`. The association between the bus and its driver is modeled with an `access` connection.

(62) There is the graphical representation of this example



(63) There is the textual representation of this example

```
package arincexample2
public
with ARINC653;

-- First, define generic component
data integer
end integer;

memory memchunk
end memchunk;

memory mainmemory
end mainmemory;

bus anybus
end anybus;

bus implementation anybus.i
end anybus.i;
thread network_driver_thread
properties
   Stack_Size => 4 Bytes;
   Code_Size => 10 Bytes;
   Period => 200 ms;
   Compute_Execution_Time => 5 ms .. 10 ms;
end network_driver_thread;

thread implementation network_driver_thread.i
end network_driver_thread.i;

abstract network_driver_partition
end network_driver_partition;

abstract implementation network_driver_partition.i
subcomponents
   thr : thread network_driver_thread.i;
end network_driver_partition.i;

device network_interface
features
   thebus: requires bus access anybus.i;
properties
          Device_Driver => classifier (arincexample2::network_driver_partition.i);
end network_interface;

device implementation network_interface.i
end network_interface.i;

virtual processor partition_runtime
properties
      Scheduling_Protocol => (RMS);
end partition_runtime;

processor arinckernel
```

```
end arinckernel;

-- Then, we define the first module and its subcomponents.

processor implementation arinckernel.module1
subcomponents
   part1 : virtual processor partition_runtime
{ARINC653::DAL => LEVEL_A;};
   part2 : virtual processor partition_runtime
                {ARINC653::DAL => LEVEL_B;};
   part3 : virtual processor partition_runtime
{ARINC653::DAL => LEVEL_C;};
properties
   ARINC653::Module_Major_Frame => 40ms;

     ARINC653::Module_Schedule =>
         ( [Partition => reference (part1);
           Duration  => 10 ms;
           Periodic_Processing_Start => false;],
           [Partition => reference (part2);
           Duration  => 10 ms;
           Periodic_Processing_Start => false;],
           [Partition => reference (part3);
           Duration  => 20 ms;
           Periodic_Processing_Start => false;]
         );
end arinckernel.module1;

memory implementation mainmemory.module1
subcomponents
   mem1 : memory memchunk;
   mem2 : memory memchunk;
   mem3 : memory memchunk;
end mainmemory.module1;

thread module1_thread_part1
features
   sensorout : out event data port integer;
end module1_thread_part1;

-- Thread for the first partition

thread module1_thread_part2
end module1_thread_part2;

-- Thread for the second partition

process module1_process_part1
features
   sensorout : out event data port integer{Queue_Size => 1;
                                          ARINC653::Timeout => 5ms;
                                          ARINC653::Queueing_Discipline => FIFO;};
end module1_process_part1;

process implementation module1_process_part1.impl
subcomponents
   mythread : thread module1_thread_part1;
connections
```

```
   c0 : port mythread.sensorout -> sensorout;
end module1_process_part1.impl;

-- Process for the first partition

process module1_process_part2
end module1_process_part2;

process implementation module1_process_part2.impl
subcomponents
   mythread : thread module1_thread_part2;
end module1_process_part2.impl;

-- Process for the second partition

system module1_system
features
   thebus: requires bus access anybus.i;
   sensorout : out event data port integer;
end module1_system;

system implementation module1_system.impl
subcomponents
   netif : device network_interface.i;
   mainmemory : memory mainmemory.module1;
   cpu : processor arinckernel.module1;
   process_part1 : process module1_process_part1;
   process_part2 : process module1_process_part2;
connections
   c0 : port process_part1.sensorout -> sensorout;
   c1 : bus access thebus -> netif.thebus;
properties
   Actual_Processor_Binding   => (reference (cpu.part1)) applies to process_part1;
   Actual_Processor_Binding   => (reference (cpu.part2)) applies to process_part2;
   Actual_Processor_Binding   => (reference (cpu.part2)) applies to netif;
   Actual_Memory_Binding  => (reference (mainmemory.mem1)) applies to process_part1;
   Actual_Memory_Binding  => (reference (mainmemory.mem2)) applies to process_part2;
   Actual_Memory_Binding  => (reference (mainmemory.mem3)) applies to netif;
end module1_system.impl;

-- System that contain everything for the first module

-- Now, we declare the second module

processor implementation arinckernel.module2
subcomponents
   part1 : virtual processor partition_runtime;
properties
   ARINC653::Module_Major_Frame => 40ms;
     ARINC653::Module_Schedule =>
         ( [Partition => reference (part1);
           Duration  => 20 ms;
           Periodic_Processing_Start => false;],
           [Partition => reference (part1);
           Duration  => 20 ms;
           Periodic_Processing_Start => false;]
         );
end arinckernel.module2;
```

```
thread module2_thread_part1
features
   sensorin : in event data port integer;
end module2_thread_part1;

process module2_process_part1
features
   sensorin : in event data port integer{Queue_Size => 1;
                                        ARINC653::Timeout => 5ms;
                                        ARINC653::Queueing_Discipline => FIFO;};
end module2_process_part1;

process implementation module2_process_part1.impl
subcomponents
   thread_part1 : thread module2_thread_part1;
connections
   c0 : port sensorin -> thread_part1.sensorin;
end module2_process_part1.impl;

memory implementation mainmemory.module2
subcomponents
   mem1 : memory memchunk;
   mem2 : memory memchunk;
end mainmemory.module2;

system module2_system
features
   thebus   : requires bus access anybus.i;
   sensorin : in event data port integer;
end module2_system;

system implementation module2_system.impl
subcomponents
   mainmemory     : memory mainmemory.module2;
   cpu            : processor arinckernel.module2;
   process_part1  : process module2_process_part1.impl;
   netif          : device network_interface.i;
connections
   c0 : port sensorin -> process_part1.sensorin;
   c1 : bus access thebus -> netif.thebus;
properties
   Actual_Processor_Binding   => (reference (cpu.part1)) applies to process_part1;
   Actual_Processor_Binding   => (reference (cpu.part1)) applies to netif;
   Actual_Memory_Binding  => (reference (mainmemory.mem1)) applies to process_part1;
   Actual_Memory_Binding  => (reference (mainmemory.mem2)) applies to netif;
end module2_system.impl;


-- Now, we declare the main system that contains both modules

system arinc653system
end arinc653system;

system implementation arinc653system.impl
subcomponents
   module1  : system module1_system.impl;
   module2  : system module2_system.impl;
```

```
   rtbus    : bus anybus.i;
connections
   conn1 : port module1.sensorout -> module2.sensorin;
   busaccess_module1 : bus access rtbus -> module1.thebus;
   busaccess_module2 : bus access rtbus -> module2.thebus;
properties
   Actual_Connection_Binding => (reference (rtbus)) applies to conn1;
end arinc653system.impl;
end arincexample2;
```