ARINC

AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE PART 1 REQUIRED SERVICES

ARINC SPECIFICATION 653P1-5 AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE SET

PUBLISHED: December 23, 2019

Prepared by AEEC
Published by
SAE-ITC
16701 Melford Blvd., Suite 120, Bowie, Maryland 20715 USA

SAE ITC
An SAE International Affiliate

DISCLAIMER

THIS DOCUMENT IS BASED ON MATERIAL SUBMITTED BY VARIOUS PARTICIPANTS DURING THE DRAFTING PROCESS. NEITHER AEEC, AMC, FSEMC NOR SAE-ITC HAS MADE ANY DETERMINATION WHETHER THESE MATERIALS COULD BE SUBJECT TO VALID CLAIMS OF PATENT, COPYRIGHT OR OTHER PROPRIETARY RIGHTS BY THIRD PARTIES, AND NO REPRESENTATION OR WARRANTY, EXPRESS OR IMPLIED, IS MADE IN THIS REGARD.

ARINC INDUSTRY ACTIVITIES USES REASONABLE EFFORTS TO DEVELOP AND MAINTAIN THESE DOCUMENTS. HOWEVER, NO CERTIFICATION OR WARRANTY IS MADE AS TO THE TECHNICAL ACCURACY OR SUFFICIENCY OF THE DOCUMENTS, THE ADEQUACY, MERCHANTABILITY, FITNESS FOR INTENDED PURPOSE OR SAFETY OF ANY PRODUCTS, COMPONENTS, OR SYSTEMS DESIGNED, TESTED, RATED, INSTALLED OR OPERATED IN ACCORDANCE WITH ANY ASPECT OF THIS DOCUMENT OR THE ABSENCE OF RISK OR HAZARD ASSOCIATED WITH SUCH PRODUCTS, COMPONENTS, OR SYSTEMS. THE USER OF THIS DOCUMENT ACKNOWLEDGES THAT IT SHALL BE SOLELY RESPONSIBLE FOR ANY LOSS, CLAIM OR DAMAGE THAT IT MAY INCUR IN CONNECTION WITH ITS USE OF OR RELIANCE ON THIS DOCUMENT, AND SHALL HOLD SAE-ITC, AEEC, AMC, FSEMC AND ANY PARTY THAT PARTICIPATED IN THE DRAFTING OF THE DOCUMENT HARMLESS AGAINST ANY CLAIM ARISING FROM ITS USE OF THE STANDARD.

THE USE IN THIS DOCUMENT OF ANY TERM, SUCH AS SHALL OR MUST, IS NOT INTENDED TO AFFECT THE STATUS OF THIS DOCUMENT AS A VOLUNTARY STANDARD OR IN ANY WAY TO MODIFY THE ABOVE DISCLAIMER. NOTHING HEREIN SHALL BE DEEMED TO REQUIRE ANY PROVIDER OF EQUIPMENT TO INCORPORATE ANY ELEMENT OF THIS STANDARD IN ITS PRODUCT. HOWEVER, VENDORS WHICH REPRESENT THAT THEIR PRODUCTS ARE COMPLIANT WITH THIS STANDARD SHALL BE DEEMED ALSO TO HAVE REPRESENTED THAT THEIR PRODUCTS CONTAIN OR CONFORM TO THE FEATURES THAT ARE DESCRIBED AS MUST OR SHALL IN THE STANDARD.

ANY USE OF OR RELIANCE ON THIS DOCUMENT SHALL CONSTITUTE AN ACCEPTANCE THEREOF "AS IS" AND BE SUBJECT TO THIS DISCLAIMER.

THE AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE SET	
653P0	Avionics Application Software Standard Interface Part 0 – Overview of ARINC 653
653P1	Avionics Application Software Standard Interface Part 1 – Required Services
653P2	Avionics Application Software Standard Interface Part 2 – Extended Services
653P3A	Avionics Application Software Standard Interface Part 3A – Conformity Test Specifications for ARINC 653 Required Services
653P3B	Avionics Application Software Standard Interface Part 3B – Conformity Test Specifications for ARINC 653 Extended Services
653P4	Avionics Application Software Standard Interface Part 4 – Subset Services
653P5	Avionics Application Software Standard Interface Part 5 – Core Software Recommended Capabilities

Note: Within ARINC Standards, their identification is listed in its basic form. When an ARINC Standard is modified by a supplement, the numeric notation is changed by adding the supplement identifier as a suffix, e.g., ARINC 758-2. Where references are made to an ARINC Standard, only the basic number is used. The reader should assume that the reference includes all relevant supplements.

ARINC SPECIFICATION 653P1-5

AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE
PART 1
REQUIRED SERVICES

Published: December 23, 2019

Prepared by the Airlines Electronic Committee (AEEC)
Adopted by the AEEC Executive Committee

	Adoption Date	Publication Date
Specification 653	October 4, 1996	January 1, 1997
	Supplements to this ARINC Standard	
Specification 653-1	September 16, 2003	October 16, 2003
Specification 653P1-2	October 4, 2005	March 7, 2006
Specification 653P1-3	October 6, 2010	November 15, 2010
Specification 653P1-4	April 29, 2015	August 21, 2015
Specification 653P1-5	May 1, 2019	December 23, 2019

A summary of the changes introduced by each supplement is included at the end of this document.

FOREWORD

SAE ITC, the AEEC, and ARINC Standards

ARINC Industry Activities, an industry program of SAE ITC, organizes aviation industry committees and participates in related industry activities that benefit aviation at large by providing technical leadership and guidance. These activities directly support aviation industry goals: promote safety, efficiency, regularity, and cost-effectiveness in aircraft operations.

ARINC Industry Activities organizes and provides the secretariat for international aviation organizations (AEEC, AMC, FSEMC) which coordinate the work of aviation industry technical professionals and lead the development of technical standards for airborne electronic equipment, aircraft maintenance equipment and practices, and flight simulator equipment used in commercial, military, and business aviation. The AEEC, AMC, and FSEMC develop consensus-based, voluntary standards that are published by SAE ITC and are known as ARINC Standards. The use of ARINC Standards results in substantial technical and economic benefit to the aviation industry.

There are three classes of ARINC Standards:

a) ARINC Characteristics – Define the form, fit, function, and interfaces of avionics and other airline electronic equipment. ARINC Characteristics indicate to prospective manufacturers of airline electronic equipment the considered and coordinated opinion of the airline technical community concerning the requisites of new equipment including standardized physical and electrical characteristics to foster interchangeability and competition.

b) ARINC Specifications – Are principally used to define either the physical packaging or mounting of avionics equipment, data communication standards, or a high-level computer language.

c) ARINC Reports – Provide guidelines or general information found by the airlines to be good practices, often related to avionics maintenance and support.

The release of an ARINC Standard does not obligate any organization or SAE ITC to purchase equipment so described, nor does it establish or indicate recognition or the existence of an operational requirement for such equipment, nor does it constitute endorsement of any manufacturer's product designed or built to meet the ARINC Standard.

In order to facilitate the continuous product improvement of this ARINC Standard, two items are included in the back of this volume:

An Errata Report solicits any corrections to existing text or diagrams that may be included in a future Supplement to this ARINC Standard.

An ARINC IA Project Initiation/Modification (APIM) form solicits any proposals for the addition of technical material to this ARINC Standard.

1.0 INTRODUCTION

1.1 Purpose

This document specifies the basic operating environment for application software used within Integrated Modular Avionics (IMA) and traditional ARINC 700-series avionics.

The primary objective of this specification is to define a general-purpose APEX (APplication/EXecutive) interface between the Operating System (O/S) of an avionics computer resource and the application software. Included within this specification are the interface requirements between the application software and the O/S and the list of services which allow the application software to control the scheduling, communication, and status information of its internal processing elements.

This specification defines the data exchanged statically (via configuration) or dynamically (via services) as well as the behavior of services provided by the O/S and used by the application. It is not the intent of this specification to dictate implementation requirements on either the hardware or software of the system, nor is it intended to drive certain system-level requirements within the system which follows this standard.

The majority of this document describes the runtime environment for embedded avionics software. This list of services identifies the minimum functionality provided to the application software and is, therefore, the industry standard interface. It is intended for this interface to be as generic as possible, since an interface with too much complexity or too many system-specific features is normally not accepted over a variety of systems. The software specifications of the APEX interface are High-Order Language (HOL) independent, allowing systems using different compilers and languages to follow this interface.

This document is intended to complement **ARINC Report 651:** *Design Guidance for Integrated Modular Avionics.* It is expected that this document will evolve to contain additional functionality and capability. Supplements to this document will be prepared as needed by the industry.

1.2 Scope

This document specifies both the interface and the behavior of the API services. Behavior is specified to the extent needed to describe functionality relevant to interfacing applications.

Where necessary, assumptions are made as to the support or behavior provided by the operating system and hardware. This should not be construed as a specification for the O/S or hardware. However, where the O/S or hardware does not coincide with the stated assumptions, the API behaviors specified herein may not match the actual behavior.

ARINC 653 is intended for use in a partitioned environment. In order to assure a high degree of portability, aspects of the partitioned environment are discussed and assumed. However, this specification does not define the complete system, hardware, and software requirements for partitioning, nor does it provide guidance on proper implementation of partitioning, and in particular, robust partitioning. It must not be construed that compliance to ARINC 653 assures robust partitioning.

1.3 APEX Phases

(This section removed by Supplement 4. See Section 2.3.5.4.)

1.4 ARINC Specification 653 Basic Philosophy

ARINC 653 Part 0, *Overview of ARINC 653*, Section 1.3 describes the basic philosophies of ARINC 653 and the decomposition of software and hardware that may reside within an integrated module.

1.5 Implementation Guidelines

ARINC 653 Part 0, *Overview of ARINC 653*, Section 1.5 provides implementation guidelines for ARINC 653.

1.6 Document Overview

ARINC 653 Part 0, *Overview of ARINC 653*, Section 1.4 provides an overview of ARINC 653 and its various parts.

This document, ARINC 653 Part 1, defines "Required Services." Part 1 constitutes the basic requirements and guidance for an ARINC 653 compliant O/S API. Part 1 is organized into five sections and a number of appendices, as follows:

Section 1.0 Introduction
Section 2.0 System Overview
Section 3.0 Service Requirements
Section 4.0 Compliance
Section 5.0 XML Configuration
Appendix A Glossary (See ARINC 653 Part 0)
Appendix B Acronyms (See ARINC 653 Part 0)
Appendix C (deleted by Supplement 3)
Appendix D Ada83 and Ada95 Language Interface Specification
Appendix E C Language Interface Specification
Appendix F (deleted by Supplement 1)
Appendix G Graphical view of ARINC 653 XML Schema Types
Appendix H ARINC 653 XML Schema Types
Appendix I Example XML Schema and its Corresponding XML Data Instance

The **Introduction** section describes the purpose of this document and provides an overview of the basic philosophy behind the development of the APEX interface.

The **System Overview** section defines the assumptions pertinent to the hardware and software implementation within the target system, while it also identifies the assumed requirements specific to the O/S and application software. It identifies the basic concepts of the execution environment of the system, providing an overall understanding of the interactions between the individual elements within the system.

The **Service Requirements** section identifies those fundamental features which the application software needs in order to control the operation and execution of its processes. These requirements neither favor one particular type of application nor advocate any specific designs of the application software. This section also identifies the actual list of service requests which satisfy the individual requirements. These requests form the basis of this interface standard.

The **Compliance** section discusses necessary consideration for an O/S or application to be compliant to the APEX interface.

The **XML Configuration** section discusses the use of XML for defining the contents of the configuration.

1.7 Relationship to Other Standards

ARINC 653 Part 0, *Overview of ARINC 653*, Section 1.6 describes other standards that are related to the development of ARINC 653.

1.8 Related Documents

(The content of this section was moved to ARINC 653 Part 0, *Overview of ARINC 653*, Section 1.6.)

2.0 SYSTEM OVERVIEW

2.1 System Architecture

This interface specification has been developed for use with an Avionics Computer Resource (ACR). Its implementation may be Integrated Modular Avionics (IMA) or federated avionics within a Line Replaceable Unit (LRU). The system architecture supports partitioning in accordance with the IMA philosophy.

ARINC 653 Part 0, *Overview of ARINC 653*, Section 1.3 describes the basic philosophies of ARINC 653 and the decomposition of software and hardware that may reside within an integrated module. The first level decomposition of an integrated module consists of two major categories:

1. Partitioned application software.
2. Core module.

A core module can contain one or more **processor cores, from one or more processors, and allocated resources including memory and devices.** The core module architecture has influence on the O/S implementation, but not on the APEX interface used by the application software of each partition. Application software should be portable between core modules and between individual processor **cores** of a core module without modifying its interface with the O/S.

COMMENTARY

In the context of this document, when there are multiple processors or processor cores on a common hardware element, each set of processor **cores** that hosts a single ARINC 653 API execution context (i.e., execute a single module schedule at any given time) is considered to constitute a core module.

To achieve the required functionality and conform to specified timing constraints, the constituent processes of a partition may operate concurrently. The O/S provides services to control and support the operational environment for all processes within a partition. In particular, concurrency of operation is provided by the partition-level scheduling model.

The underlying architecture of a partition is similar to that of a multitasking application within a general-purpose mainframe computer. Each partition consists of one or more concurrently executing processes, sharing access to processor resources based upon the requirements of the application. All processes are uniquely identifiable, having attributes that affect scheduling, synchronization, and overall execution.

To ensure portability, communication between partitions is independent of the location of both the source and destination partition. An application sending a message to, or receiving a message from, another application will not contain explicit information regarding the location of its own host partition or that of its communications partner(s). The information required to enable a message to be correctly routed from source to destination is contained in configuration tables that are developed and maintained by the system integrator, not the individual application developer. The System Integrator configures the environment to ensure the correct routing of messages between partitions within an integrated module and between integrated modules. How this configuration is implemented is outside the scope of this standard.

2.2 Hardware

In order to isolate multiple partitions in a shared resource environment, the hardware should provide the O/S with the ability to restrict memory spaces, processing time, and access to I/O for each individual partition.

Partition timing interrupt generation should be deterministic. Any interrupts required by the hardware should be serviced by the O/S. Time partitioning should not be disturbed by the use of interrupts.

There are several basic assumptions made about the processor:

1. The processor provides sufficient processing capacity to meet worst-case timing requirements.
2. The processor has access to required I/O and memory resources.
3. The processor has access to time resources to implement the time services.
4. The processor provides a mechanism to transfer control to the O/S if the partition attempts to perform an invalid operation.
5. The processor provides atomic operations for implementing processing control constructs. These atomic operations will induce some jitter on time slicing. Also, atomic operations are expected to have minimal effect on scheduling.

2.2.1 Core Modules with Multiple Processor Cores

This standard includes support for the ARINC 653 services to be utilized with a core module that contains a single or multiple processor cores. With multiple processor cores, additional scheduling paradigms are feasible on a core module:

- Multiple processes within a partition scheduled to execute concurrently on different processor cores.
- Multiple partitions scheduled to execute concurrently on different processor cores.

Section 4.0 provides compliance requirements for an ARINC 653 compliant O/S. A multicore O/S compliant to this standard utilizes the compliance requirements defined in Section 4.2.1.

COMMENTARY

This standard covers software service and configuration aspects for an ARINC 653 based system that includes a core module with multiple processor cores. Certification considerations, including partitioning associated with core module resources (e.g., shared cache, **shared** memory controller) shared by multiple processor cores are outside the scope of this standard.

The services and configuration aspects associated with multiple processor cores are **also** intended to be source code compatible with applications developed for single-core processors. It is intended that an application developed originally for a single-core processor will not require source code modification when utilized in a partition allocated a single processor core. **A core module with multiple processor cores may utilize different ARINC 653 related type and constant definitions (i.e., recompilation should be assumed as being required).**

2.3 System Functionality

This section describes the functionality to be provided by the O/S and its interface with the application software.

At the core module level, the O/S manages partitions (partition management) and their interpartition communication, the latter being conducted either within or across integrated module boundaries. At the partition level, the O/S manages processes within a partition (process management) and communication between the constituent processes (intrapartition communication). The O/S may therefore be regarded as multi-level according to its scope of operation. O/S facilities (e.g.,

scheduling, message handling) are provided at each level, but differ in function and content according to their scope of operation. Definition of these facilities therefore distinguishes between the level at which they operate.

At any time instance, the integrated module is in initialization, operational, or idle state. On power-up, the integrated module begins execution in the initialization state. On successful completion of initialization, the integrated module enters the operational state. During operational state, the O/S manages the partitions, processes, and communications. The integrated module continues in the operational state until power is removed, or the module health monitoring function commands the integrated module to reinitialize or to enter idle state.

COMMENTARY

The O/S implementer may further refine integrated module states in support of implementing module health monitoring (see Section 2.4).

2.3.1 Partition Management

Central to the ARINC 653 philosophy is the concept of partitioning, whereby the applications resident in an integrated module are partitioned with respect to space (memory partitioning) and time (temporal partitioning). A partition is therefore a program unit of the application designed to satisfy these partitioning constraints.

The O/S supports robust partitioning. A robustly partitioned system allows partitions with different criticality levels to execute in the same integrated module, without affecting one another spatially or temporally in unexpected ways.

The portion of the O/S that operates at the core module level is responsible for enforcing partitioning and managing the individual partitions within the integrated module. This portion of the O/S should be isolated and protected against failures of any software (e.g., application, O/S, language support libraries, etc.) executing in the context of a partition.

Partitions are scheduled on a fixed, cyclic basis. To assist this cyclic activation, the O/S maintains a **module schedule** of fixed duration **(i.e., major time frame)**, which is periodically repeated throughout the integrated module's runtime operation. Partitions are activated by being assigned to one or more partition time windows within **the module schedule**. Each partition time window is defined by its offset from the start of the **module schedule** and expected duration. This provides a deterministic scheduling methodology whereby the partitions are furnished with a predetermined amount of time to access processor resources.

Within a partition time window, the partition's processes are managed as described in Section 2.3.2.3.

COMMENTARY

Temporal partitioning is influenced by the O/S overhead. Inter-module communications acknowledgements and time-outs may interrupt one partition even though the events relate to a different partition. As a result, the time duration allocated for use by an application may be impacted.

Each partition has predetermined areas of memory allocated to it. These unique memory areas are identified based upon the requirements of the individual partitions and vary in size and access rights.

Configuration of all partitions throughout the whole system is expected to be under the control of the system integrator and maintained with configuration tables. The configuration table for the module schedule defines the major time frame and describes the order of activation of the partition time windows within that major time frame.

Partition management services are available to the application software for setting a partition's operating mode and to obtain a partition's status.

2.3.1.1 Partition Attribute Definition

In order for partitions to be supported by a core module, a set of unique attributes will be defined for each partition. These attributes enable the O/S to control and maintain each partition's operation.

Partition attributes are as follows:

FIXED ATTRIBUTES (defined in configuration tables using the ARINC 653 XML schema **defined in Appendix G**):

1. Identifier – defines a numerical identity for the partition. It is uniquely defined at least on a core module basis. It may be used by the O/S to facilitate partition activation, message routing, and debug support. It may also be used by the application, for example, as a field within a maintenance message.
2. Name – defines a string identity for the partition. It is uniquely defined at least on a core module basis.
3. Memory Requirements – defines the quantity of memory available to the partition, with appropriate code/data segregation. The memory requirements may include separate specification of the process stack sizes. It is O/S implementation dependent whether definitions of process stack sizes are required as separate memory requirements or are included in a data memory allocation.
4. Partition Period – defines the required time interval at which the partition must be activated in order to satisfy its operational requirements.

COMMENTARY

Typically, the partition period is the greatest common factor of the process periods within a partition. If the process periods are harmonic, then this is the period of the process in a partition that has the shortest period. If there are no periodic processes in a partition, then the period is based on the minimum execution frequency required by an application to satisfy its performance requirements. The partition period is used as an input in determining offsets and durations of partition time windows.

5. Partition Duration – the amount of execution time required by the partition within one partition period.

COMMENTARY

This is used as an input in determining offsets and durations of partition time windows.

6. Interpartition Communication Requirements (Ports) – denote those partitions and/or devices with which the partition communicates.
7. Partition Health Monitor Table – denotes Health Monitor (HM) actions to be performed on detection of failures assigned to the partition.

FIXED ATTRIBUTES (not included in the ARINC 653 XML schema):

1. Number of Associated Processor Core(s) – defines the number of logical processor cores, fixed at configuration time, associated with the partition that can be used to schedule processes. An application can use services during the partition initialization phase to obtain the number of logical processor cores available to the partition for running processes. The associated logical processor cores are available to the partition for process scheduling for the duration of each of the partition's time windows. An application schedules processes to run on a specific logical processor core by initializing the process to have an affinity to this logical processor core.

   COMMENTARY

   A single logical processor core can be assigned for a partition (i.e., mono-core partition), even when the core module supports the assignment of multiple logical processor cores.

2. Entry Point (i.e., partition initialization) – denotes partition start/restart address.
3. System Partition – denotes the partition is a system partition (i.e., is not restricted to using ARINC 653 services to interact with the O/S).

VARIABLE ATTRIBUTES (defined and controlled during run time):

1. Lock Level – denotes the current lock level of the process within the partition that has preemption locked (see Section 2.3.2.6).
2. Operating Mode – denotes the partition's execution state (see Section 2.3.1.4).
3. Start Condition – denotes the condition under which the partition was started.

2.3.1.2 Partition Control

The O/S starts the application partitions when the O/S enters operational state. When the O/S starts an application partition after a core module power-up, the partition starts in the COLD_START mode. The resources used by each partition (channels, processes, queues, semaphores, events, etc.) are specified at system build time. The corresponding objects (i.e., data structures) are created during the partition's initialization phase, and then the partition enters NORMAL mode. The application is responsible for invoking the appropriate APEX calls to transition the partition from one operational mode to another. The operational mode of one partition is independent of the operational mode of other partitions. Thus, some partitions may be in the COLD_START mode while other partitions are in the NORMAL or WARM_START mode. The HM function can restart, or set to IDLE mode, a single partition, multiple partitions, or the entire integrated module in response to a fault.

2.3.1.3 Partition Scheduling

Scheduling of partitions is strictly deterministic over time. Based upon the configuration of partitions within an integrated module, overall resource requirements/availability, and specific partition requirements, a time-based activation Module Schedule is generated that identifies the partition time windows allocated to the individual partitions. The Module Schedule is enforced during the operational state of the integrated module. Each partition is scheduled according to its respective Partition Time Windows (i.e., its partition schedule).

A Partition Time Window is an uninterrupted interval of execution time provided to a partition within a partition schedule that is defined by:

a. Time Window Duration – The quantity of execution time in the Partition Time Window.

b. Time Window Offset – Time between the start of the Module Schedule and the activation of the partition time window.

c. Periodic Processing Start – An indication that periodic processes can be released at an offset defined from the beginning of the partition time window (see Section 2.3.2.3).

An example partition schedule for a single partition is shown in Figure 2.3.1.3-1. In this illustration, the partition's period is one half of the major frame time (e.g., if the major frame was 100 milliseconds, the partition's period would be 50 milliseconds). Periodic processes associated with this partition would be scheduled to be released at an offset defined from the beginning of either TW_1 or TW_3 but not at an offset defined from the beginning of either TW_2 or TW_4.

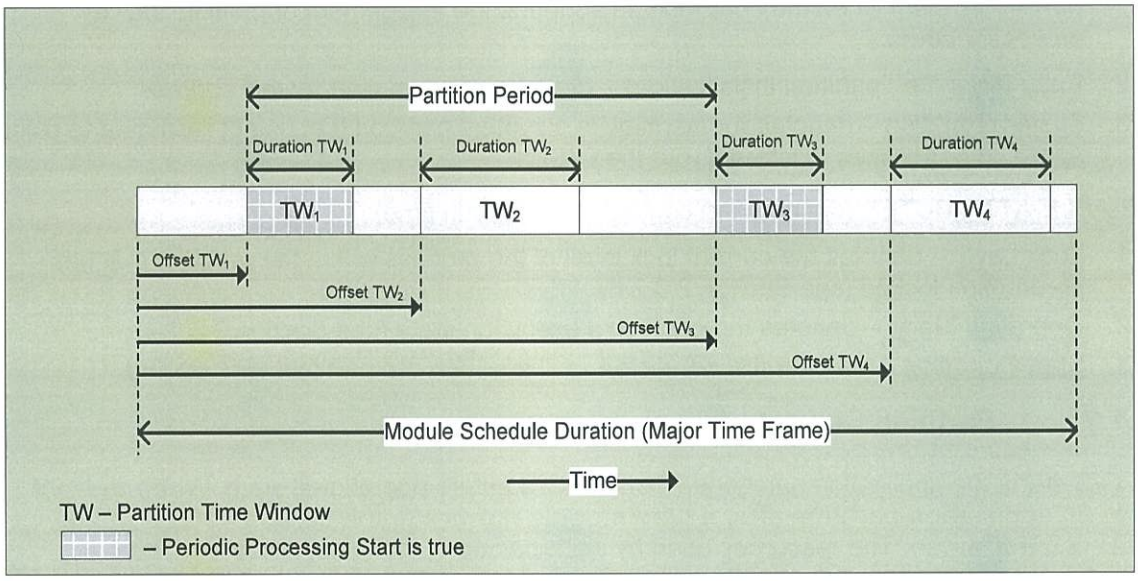

Figure 2.3.1.3-1 – Example Partition Schedule

The Module Schedule is fixed for the particular configuration of partitions within an integrated module. The Module Schedule and Partition Time Window characteristics are:

1. From the application developer perspective, the scheduling unit is a partition.
2. The partition attributes of Partition Period, Partition Duration, and Number of Associated Processor Cores are satisfied by the Module Schedule.
3. The time allocated to a partition is fixed for the Module Schedule (i.e., partitions have no priority).
4. Time allocated to a partition is divided into one or more Partition Time Windows within a partition's period.
5. The sum of the Partition Time Window durations during a Partition Period are equal to (or are greater than) the Partition Duration.
6. For the Module Schedule, a partition is assigned a Number of Associated Processor Cores.
7. A partition executes only during the Partition Time Windows it is allocated and only on the processor cores it is assigned.

COMMENTARY

A core module may support more processor cores than the maximum number required to support a partition. Only the Number of Associated Processor Cores is available to a partition during the partition's time windows. The supported number of processor cores bounds the maximum Number of Associated Processor Cores available to any partition. To increase portability, some partitions may be designed to accommodate a variable number of processor core configurations (e.g., four core, two core, single core).

COMMENTARY

There are scenarios where one or more partitions are scheduled in a specific sequence and/or with a relative time offset. The system integrator can use time window offsets to sequence partition execution in a specific order.

8. At the beginning of the partition's next Partition Time Window, partition execution resumes.

COMMENTARY

Exactly what is executed when the partition resumes depends on the partition's operating mode (see Section 2.3.1.4). In NORMAL mode, execution is influenced by changes that may have occurred to the process states (see Section 2.3.2.2.1.1) and their associated priorities while the partition was suspended.

9. Processes within a partition are executed in a priority-preemptive schedule with consideration for core affinity (see Section 2.3.2.3 for details).
10. During each Partition Time Window, the partition is provided exclusive access to the Associated Processor Cores allocated to the partition.
11. During each Partition Time Window, the partition is provided access to resources allocated to the partition (e.g., memory) in a way that meets Robust Partitioning requirements for the system (See definition of Robust Partitioning in Part 0).
12. At the end of a Partition Time Window, partition execution is suspended.
13. Each partition has at least one Partition Time Window identified as a Periodic Processing Start release point (see Section 2.3.2.3 for use).
14. Partitions do not control the allocation of resources. The platform hardware and core software exclusively control the allocation of the resources to the partition, based on configuration.

2.3.1.4 Partition Operating Modes

The current execution state of the partition (idle, cold start, warm start, or normal) represents the partition's operating mode. The SET_PARTITION_MODE service allows the application associated with a partition to request a change to its operating mode. The Health Monitor, through the health monitoring configuration tables, can also request changes to each partition's operating mode. The current operating mode of the partition can be determined by using the GET_PARTITION_STATUS service.

Partition operating modes and their state transitions are shown in Figure 2.3.1.4.

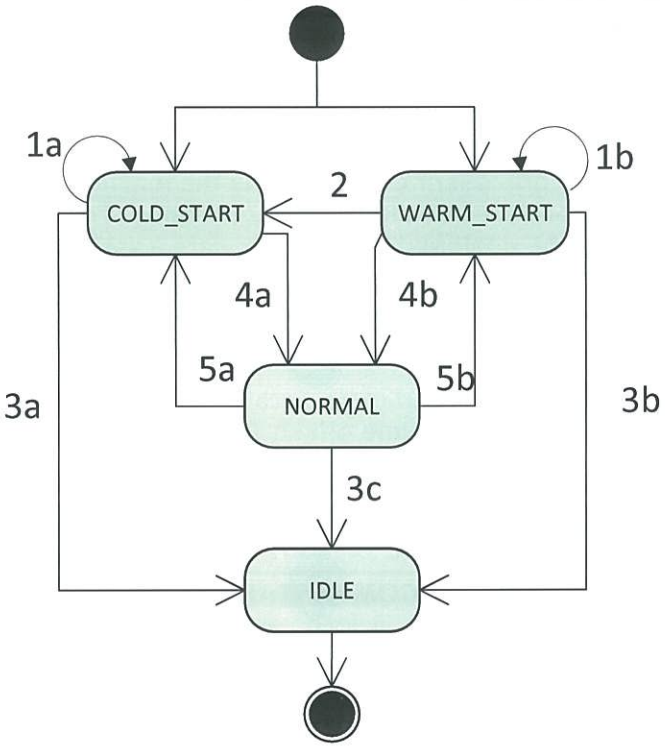

Figure 2.3.1.4 – Partition Operating Modes and Transitions

These modes are described in Section 2.3.1.4.1, and the illustrated transitions are described in Section 2.3.1.4.2. Process scheduling varies between the modes. Every partition is allocated a main process (not an ARINC 653 process created by the application) that is utilized during COLD_START and WARM_START. The main process **(assigned the MAIN_PROCESS_ID by the O/S as its process identifier)** is the default process of the application and is the only process eligible for scheduling during the COLD_START and WARM_START partition operating modes.

2.3.1.4.1 Partition Operating Modes Description

IDLE: In this mode, the partition is not eligible for application execution. The partition time windows allocated to the partition are preserved.

NORMAL: In this mode, the partition's initialization phase is complete and the partition's process scheduler is active. Except when the process-level error handler is running, all processes that have been created and those that are in the ready state are eligible to run. When the process-level error handler is running, a service that may be invoked during initialization controls whether eligible processes continue to run on other processor cores or they are prevented from making progress (i.e., pause). When a partition is allocated only a single processor core, both of these behaviors are equivalent (i.e., does not have to be configured).

COLD_START: In this mode, the partition's initialization phase is in progress. Preemption is locked with LOCK_LEVEL > 0 (main process implicitly has preemption locked and implicitly owns the preemption lock mutex)

and the associated application is executing its respective initialization code.

WARM_START: In this mode, the partition's initialization phase is in progress. Preemption is locked with LOCK_LEVEL > 0 (main process implicitly has preemption locked and implicitly owns the preemption lock mutex) and the associated application is executing its respective initialization code. This mode is similar to the COLD_START, but the initial environment (the hardware context in which the partition starts) may be different (e.g., no need for copying code from Non Volatile Memory to RAM).

2.3.1.4.1.1 COLD_START and WARM_START Considerations

COLD_START and WARM_START modes correspond to the initialization phase of the partition.

The differences between COLD_START and WARM_START depend mainly on hardware capabilities and system specifications. For example, a power interrupt does not imply to start automatically in the COLD_START mode because the content of the memory can be saved during a power interrupt. The WARM_START and COLD_START status may be used in that case to inform the partition that data values have been kept, and they may be reused when power recovers. The system integrator must inform the application developer the initial context differences between these two modes.

COMMENTARY

The main process is the only process that runs in COLD_START/WARM_START mode. This is the default process of the application. All other processes within the partition need to be created by the main process in order to declare their existence to the O/S. Once created, a process must be started in order for it to be executed. Any process started during partition initialization (i.e., to cause automatic execution following partition initialization) will not execute until the partition enters NORMAL_MODE (by calling the SET_PARTITION_MODE service). The process used during partition initialization will not continue to run after the operating mode transitions to NORMAL.

2.3.1.4.2 Partition Modes and Transitions

This section describes ARINC 653 partition modes and the characteristics expected when transitioning from one mode to another.

When a partition is assigned multiple processor cores, the transition to a different mode applies to all assigned processor cores.

2.3.1.4.2.1 COLD_START and WARM_START Mode Transition

Various events may cause a transition to the COLD_START or WARM_START modes: power interrupt, hardware reset, Health Monitoring action, or the SET_PARTITION_MODE service.

For Entry Point management, the O/S will launch the partition at a unique entry point (the same Entry Point for COLD_START and WARM_START).

COMMENTARY

For some languages (Ada, C), an application's entry point is defined by the language's standard.

2.3.1.4.2.2 Transition Mode Description

(1a) COLD_START -> COLD_START

(1b) WARM_START -> WARM_START

When the partition is currently initializing and a condition occurs where the partition is required to restart using the same mode the partition previously started with.

The parameter START_CONDITION returned by the GET_PARTITION_STATUS service provides an application the cause of entering the COLD_START or WARM_START mode. This parameter allows the application associated with a partition to manage its own processing environment on initialization error, depending on its own start context (e.g., create/does not create a particular process or port). It can use this information to prevent continuously restart of a partition due to persistent errors.

(2) WARM_START -> COLD_START

When the partition is currently initializing and a condition occurs where the partition is required to restart using a different mode than the partition previously started with.

The initial context of a WARM_START is considered to be more complete than an initial context of a COLD_START, so it is not possible to go from COLD_START to WARM_START directly or indirectly via a transition through the idle state. If the partition is in the COLD_START mode and the defined HM recovery action is WARM_START, then a COLD_START is performed.

(3a) COLD_START -> IDLE

(3b) WARM_START -> IDLE

(3c) NORMAL -> IDLE

When the partition calls the SET_PARTITION_MODE service with the **operating mode** set to IDLE, or when the Health Monitor makes the recovery action (i.e., according to the partition's health monitor table) to shutdown the partition.

(4a) COLD_START -> NORMAL

(4b) WARM_START -> NORMAL

When the partition has completed its initialization and calls the SET_PARTITION_MODE service with the **operating mode** set to NORMAL. Normal mode process scheduling (see Section 2.3.2.3) begins. When a partition has been allocated multiple processor cores, process scheduling will begin for all processor cores allocated to the partition.

COMMENTARY

When a partition has been allocated multiple processor cores, some process scheduling skew may occur between the cores.

(5a) NORMAL -> COLD_START

(5b) NORMAL -> WARM_START

The partition is in the normal mode and restart is requested. Normal mode process scheduling (see Section 2.3.2.3) is discontinued. When a partition has been allocated multiple processor cores, process scheduling on all of the allocated processor cores will be impacted by the request to restart the partition.

(6a) IDLE -> COLD_START

(6b) IDLE -> WARM_START

These transitions are not shown in Figure 2.3.1.4. The only mechanism available to transition from the IDLE mode is an action external to the partition, such as power interrupt, core module reset, or application reset, if an external means exist**s (i.e., implementation dependent)**. The IDLE to WARM_START transition should be prohibited when IDLE mode is entered from COLD_START.

2.3.2 Process Management

Within the ARINC 653 concept, a partition comprises one or more processes that combine dynamically to provide the functions associated with that partition. According to the characteristics associated with the partition, the constituent processes may operate concurrently in order to achieve their functional and real-time requirements. Multiple processes are therefore supported within the partition. Processes within a partition share the same address space.

An application requires certain scheduling capabilities from the operating system in order to accurately control the execution of its processes in a manner that satisfies the requirements of the application. Processes may be designed for periodic **(scheduled to be released into the ready state at a periodic rate)** or aperiodic **(the starting, stopping, blocking, and release of a process is entirely under application control)** execution, the occurrence of a fault may require processes to be reinitialized or terminated, and a method to prevent a running process from being preempted is required in order to safely access resources that demand mutually-exclusive access. An affinity to run on a specific processor core may be specified for each process.

A process is a programming unit contained within a partition which executes concurrently with other processes of the same partition. It comprises the executable program, data and stack areas, program counter, stack pointer, and other attributes such as priority and deadline. For references within this specification, the term “process” will be used in place of “task” to avoid confusion with the Ada task construct.

Access to process management functions is via the utilization of APEX services. The set of services called in the application software should be consistent with the **partition’s** criticality level. Partition code executes in user mode only (i.e., no privileged instructions are allowed).

The partition should be responsible for the behavior of its defined processes. The processes are not directly visible outside of the partition.

2.3.2.1 Process Attribute Definition

In order for processes to execute, a set of unique attributes needs to be defined for each process. These attributes differentiate between the unique characteristics of each process as well as define resource allocation requirements.

A description of the process attributes is given below. Fixed attributes are statically defined and cannot be changed once the partition has been initialized. Variable attributes are defined with initial values, but can be changed through the use of service requests once the partition transitions to the NORMAL partition operating mode.

FIXED ATTRIBUTES (not included in the ARINC 653 XML schema):

1. Name – Defines a value unique to each process in the partition.
2. Entry Point – Denotes the starting address of a process.
3. Stack Size – Identifies the overall size for the run-time stack of a process. Each process is allocated a unique stack. Each process's stack may be configured to be identical or different in size.
4. Base Priority – Denotes the priority of a process defined when the process was started.
5. Period – Identifies the period of activation for a periodic process. A distinct and unique value is used to designate a process as aperiodic.
6. Time Capacity – Defines the elapsed time within which a process should complete its execution.
7. Deadline – Specifies the type of deadline relating to a process and may be "hard" or "soft." When a process fails to meet its deadline, the O/S does not react differently between "hard" and "soft" deadlines. The O/S will take the action defined for deadline missed defined in the HM tables. If a process level error handler is defined, the process level error handler can utilize the GET_PROCESS_STATUS service to obtain the deadline type and take application-specific actions based on "hard" versus "soft" deadlines.

VARIABLE ATTRIBUTES:

1. Current Priority – Defines the priority with which a process may access and receive resources. It is set to base priority at process creation and can be changed dynamically after process creation.
2. Deadline Time – Defines the absolute system time when a process will exceed its deadline. The deadline time is evaluated by the operating system to determine whether a process is satisfactorily completing its processing within the allotted time. When a process is dormant (i.e., stopped), the deadline time returned by GET_PROCESS_STATUS is undefined.
3. Process State – Identifies the current scheduling state of a process. The state of a process could be dormant, ready, running, faulted, or waiting.
4. Processor Core Affinity – Identifies the logical processor core affinity of the process. The assignment of a process to have an affinity for a specific logical processor core is used during process scheduling to restrict the process to being eligible for scheduling only on the assigned logical processor core. If no affinity is assigned during initialization, the process will have the default processor core affinity (for ARINC 653 Part 1, logical processor core #0) of the partition.

Once the process attribute information is defined for each process within the partition, a method is required to provide this information to the O/S. Normally, the linker provides a certain amount of attribute information (i.e., entry point, stack size, etc.) pertinent to the main (or initial) process of the system. Some linkers also allow for the creation of user-defined data and/or code segments during the link procedure.

2.3.2.1.1 Process Core Affinity

When multiple processor cores are assigned to a partition, an application may require control over which processor core each process runs on. An application exercises this control by assigning a processor core affinity to each process. A process can only be assigned a core affinity for one of the processor cores assigned to the partition to which the process belongs.

If a process's core affinity is not explicitly set during initialization, the process's core affinity will be assigned using the default processor core affinity. The same default is used during the initialization phase as the processor core used to execute the main process.

2.3.2.2 Process Control

The system resources available to manage the processes of a partition are statically defined at build time and system initialization.

Processes are created (e.g., resources are allocated) and initialized during partition initialization. This implies that all the processes of a partition be defined in such a way that the necessary resource utilization for each process can be determined at system build time. Each process is created only once during partition initialization (i.e., on partition restart, the processes will be recreated).

Process scheduling (see Section 2.3.2.3) starts when a partition transitions to NORMAL mode.

COMMENTARY

When a partition is assigned multiple processor cores and two or more waiting processes have been assigned an affinity for different processor cores, the processes start running on the processor cores in an implementation dependent sequence.

An application should be able to restart (i.e., reinitialize) any of its processes at any time and should also be able to prevent a process from becoming eligible to receive processor resources. Certain fault and failure conditions may necessitate a partition to restart or terminate any of its processes. These conditions may arise due to either hardware or software faults, or as a result of a distinct operational phase of the application.

Each process has a priority level, and its behavior may be synchronous (periodic) or asynchronous (aperiodic). Both types of processes can co-exist in the same partition. A running process could be preempted at any time by a ready process with a higher current priority. When a partition is assigned a single processor core (e.g., single core processor), the process in the ready state with the highest (most positive) current priority is selected over other ready processes for execution when the partition is active. When a partition is assigned multiple **logical** processor cores, a set of processes in the ready state may be selected based on current priority and core affinity to run concurrently on the assigned **logical** processor cores.

COMMENTARY

Applications designed to execute on a single core should not be configured to execute on multiple cores without consideration for impacts of concurrent process execution. When multiple processor cores are assigned to a partition containing an application that previously executed on a single core, processes that previously executed sequentially may now execute in parallel. An application that did not utilize ARINC 653 services to handle critical sections may have multiple processes

concurrently accessing the same data. Such erroneous applications may need to be updated (e.g., to add access controls over data structures used by multiple processes) to operate correctly when multiple processor cores are assigned to a partition.

The LOCK_PREEMPTION and UNLOCK_PREEMPTION services are included in the APEX specification to provide a means to control process preemption (mutex services are another means, see Section 2.3.6.2.3). Preemptibility control allows a running process to be guaranteed to execute without preemption by other processes within the partition. If a process that locked preemption is interrupted by the end of a partition time window, it is guaranteed to still have preemption locked when the partition is resumed.

An O/S may or may not support static or dynamic creation for processes or any other communication mechanisms. This should be transparent to the application software.

The mechanisms used by the processes, for inter-process communication and synchronization, are created during the initialization phase, and are not destroyed.

To become eligible for running, a process not only needs to be created, but also needs to be started. At least one process in the partition is started shortly after creation. Running processes can start others, stop themselves or others, and restart as required by the application; this is governed by the process state transition model within the O/S. Processes are never destroyed (i.e., the memory areas assigned to the process are not de-allocated).

2.3.2.2.1 Process State Transitions

The O/S views the execution of a process in the form of a progression through a succession of states. Translation between these states is according to a defined process state transition model. Figure 2.3.2.2.1.2 depicts corresponding state transitions in accordance with the operational modes of the partition.

2.3.2.2.1.1 Process States

The process states as viewed by the O/S are as follows:

1. **Dormant** – A process that is ineligible to receive resources for scheduling. A process is in the dormant state before it is started and after it is stopped (or terminated, e.g., as a result of a health monitoring event).
2. **Ready** – A process that is eligible for scheduling. A process is in the ready state if it is able to be executed but is not currently in a running state.
3. **Running** – A process that is currently executing on a processor core. A process is in the running state if it is currently being executed by a processor core. A process may remain in the running state due to a health-monitoring event causing the error handler process to be invoked and the partition is configured to pause running processes while the error handler process is running.
4. **Waiting** – A process that is not allowed to receive resources eligible for scheduling until a particular event condition occurs. A process is in the waiting state if one or both of the following are applicable:
    - The first reason is one of the following (waiting on a resource):
        - waiting on a delay
        - waiting on a semaphore
        - waiting on a process period

- waiting on an event
- waiting on a message
- waiting on the start of the partition's NORMAL mode
- waiting to own a mutex (when multiple processor cores are assigned to a partition), including the preemption lock

- The second reason is:
  - suspended (waiting for resume)

5. **Faulted** – A process that is no longer running due to the process causing a health-monitoring error detected by the O/S that does not permit the process to continue to run (i.e., fatal error). When the process causes a fatal error (see Section 2.4.2.1), continued operation is not possible. Processes that report an application error or overrun their deadlines will continue to be eligible to run (i.e., will not result in a faulted process state). A faulted process must be stopped before it can be started again.

2.3.2.2.1.2 Process State Transitions Versus Partition Mode Transitions

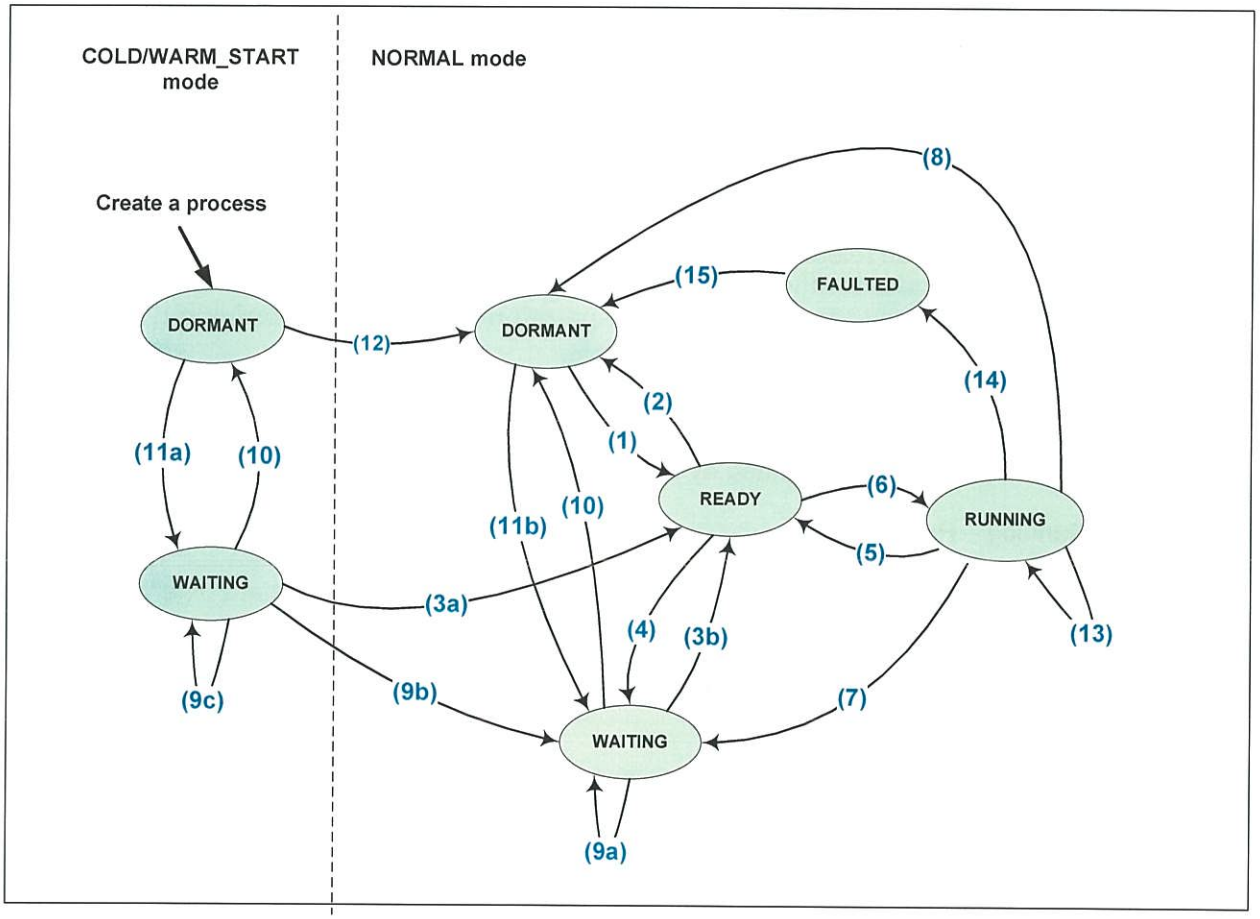

Figure 2.3.2.2.1.2 – Process States and State Transitions in Accordance with the Modes of the Partition

2.3.2.2.1.3 Process State Transitions

COLD_START and WARM_START modes correspond to the initialization phase of the partition.

State transitions occur as follows:

(1) Dormant – Ready

When a process is started by another process and the partition is in the NORMAL mode. Also, when an aperiodic process is delay started (DELAY_TIME = 0) by another process and the partition is in the NORMAL mode.

(2) Ready – Dormant

When a process is stopped by another process and the partition is in the NORMAL mode.

(3a) Waiting – Ready

When the partition transitions from the initialization phase to the NORMAL mode, and a process is an aperiodic process started during initialization phase (and was not suspended during initialization phase).

(3b) Waiting – Ready

Either when a suspended process is resumed, or the resource that a process was waiting on becomes available or a time-out a process was waiting on expires.

Comment: A periodic process waiting on its period goes automatically to the ready state when the awaited release point is reached. A periodic process's first activation (e.g., after being started while in the Normal mode) will be calculated using the start of the next time window associated with the partition that has been configured for periodic processing start (see Section 2.3.1.3).

(4) Ready - Waiting

When a process is suspended by another process within the partition.

(5) Running – Ready

When a running process waits on a DELAY_TIME of zero and there are other ready processes that have the same priority value. Also, when a running process is preempted by another process within the partition.

(6) Ready – Running

When a process is selected for execution.

(7) Running – Waiting

When a running process suspends itself or when a process attempts to access a resource (semaphore, event, message, delay or process period) which is not currently available and the process chooses (e.g., by setting a non-zero timeout) to wait.

Waiting on the process period means that the periodic process waits until the next release point by using the PERIODIC_WAIT service.

When a running process is suspended by another running process (when multiple processor cores are allocated to the partition).

(8) Running – Dormant

When a running process stops itself.

When a running process is stopped by another running process (when multiple processor cores are allocated to the partition).

(9a) Waiting – Waiting

When a process already waiting to access a semaphore, an event, a message, or a delay is suspended. Also, when a process which is both waiting to access a resource and is suspended, is either resumed, or the resource becomes available, or the time-out expires.

(9b) Waiting – Waiting

When the partition goes from initialization phase to NORMAL mode, and a process is a periodic process that was started during initialization phase. Also, when a process is an aperiodic process that was suspended or delayed started during the initialization phase.

(9c) Waiting – Waiting

When a process is suspended while the partition is in the initialization phase. **Also, when a suspended process is resumed while the partition is in the initialization phase (i.e., it remains in the waiting state).**

(10) Waiting – Dormant

When a process is stopped by another process within the partition. Note this applies to both initialization phase and NORMAL mode.

(11a) Dormant – Waiting

When a process is started and the partition is in the initialization phase.

(11b) Dormant – Waiting

When a periodic process is started (either with a delay or not) and the partition is in **the** NORMAL mode. **Also, when an aperiodic process is delay started (with DELAY_TIME > 0) and the partition is in the NORMAL mode.**

(12) Dormant – Dormant

When the partition transitions from initialization phase to NORMAL mode and a process has not yet been started.

(13) Running – Running

When a running process invokes an APEX service that generally results in a change in process state and it returns without changing the process state (e.g., passes a DELAY_TIME of zero, or use of PERIODIC_WAIT under some conditions, see Section 2.3.3).

When a running process does not make progress because the error handler process is running and the partition has been configured to pause processes running concurrently on other processor cores when the error handler process is running. When the error handler process completes, the processes may continue to make progress.

(14) Running – Faulted

When a running process is halted by the O/S due to the process causing a health-monitoring error for which continued execution is not feasible (i.e., fatal error).

(15) Faulted – Dormant

When a faulted process is stopped by another process while the partition is in NORMAL mode.

When the partition restarts in COLD_START or WARM_START mode, the previously created processes no longer exist and will need to be re-created.

State transitions may occur automatically as a result of certain APEX services called by the application software to perform its processing. They may also occur as a consequence of normal O/S processing, due to time-outs, faults, etc.

2.3.2.3 Process Scheduling

One of the main activities of the O/S is to arbitrate the competition that results in a partition when multiple processes of a partition concurrently request use of the available processor cores. The O/S manages the execution environment for each partition, dispatching and preempting processes based on their processor core affinity assignments, respective priority levels, and current states.

The main characteristics of the process scheduling model used at the partition level are:

1. The scheduling unit is an APEX process.
2. Each process has a current priority.
3. The set of processes eligible for running is restricted to those processes in the ready or running states for the active partition.
4. The error handler process (see Section 2.4.2.1), if defined, has higher priority over all other processes, including a process that has locked preemption. Partition configuration (via a service call) controls whether non-faulted processes continue to run in parallel with the error handler process or whether other non-faulted running processes do not make progress (i.e., are paused) while the error handler process is running.
5. A running process that has locked preemption can only be preempted by the error handler process.
6. The process scheduling algorithm is priority preemptive. If a running process is higher priority than any process in the ready state, the running process will not be preempted.
7. During any process rescheduling event (caused either by a direct request from a running process or any partition internal event), the O/S will select for running up to 'N' processes concurrently, where 'N' is the number of processor cores assigned to the partition. Any process whose affinity is a specific processor core is restricted to run only on that processor core. Processes that have higher (most positive) current priority will be scheduled on the processor cores over lower priority processes, taking into account any core affinities assigned to the processes.

COMMENTARY

By all processes having an affinity for a specific processor core, the resulting behavior is that each processor core assigned to the partition will have a fixed set of processes that will be scheduled on it.

When a partition is assigned a single processor core (e.g., single core processor), the O/S selects the highest current priority process in the ready state within the partition to receive processor resources.

8. If several ready processes have the same current priority, the O/S selects a process that has been continuously eligible for running the longest time at that priority value.

   Note however that if process A is running and is preempted by higher-priority process B, process A (given that its priority was not modified) will be selected to run before other processes that have the same priority as process A.

   If process A lowers its priority to be the same priority as ready process C (and process A has not locked preemption), the current process A will be the newest process ready at that priority level and will be preempted by process C.

   A running process will control the assigned **logical** processor core resources until another process rescheduling event occurs.
9. Periodic and aperiodic scheduling of processes are both supported.

COMMENTARY

The requirement for periodic and aperiodic scheduling of processes does not imply that the O/S must distinguish between two types of processes. It means that the APEX interface provides the application software with the ability to request scheduling of processes on a periodic or aperiodic (e.g., event-driven) basis.

10. All the processes within a partition share the resources allocated to the partition.

When periodic processes are started, **the time they actually start** is **based on** the partition's configured periodic processing start designations. Each of the partition's allocated partition time windows are individually designated as whether they are or are not **the** partition's periodic processing start **release points**. For periodic processes, the first release point of the process is relative to the start of **the partition's** next partition time window **configured** as a periodic processing start **release point**. For periodic processes, subsequent process release points will occur at the process's defined period.

COMMENTARY

When two or more periodic processes within a partition have the same priority and delay, it is implementation dependent as to the order the processes will be started at the release point. When two or more processes have the same delay but affinities for different processor cores, it is implementation dependent as to the order the processes will be started at the release point.

Aperiodic processes only have an initial release point. When aperiodic processes are started, their start time is relative to the current time.

2.3.2.4 Process Priority

Three types of priority are represented in the system, current, base, and retained. Current priority is the priority used by the O/S for scheduling and process queues. Base priority is the priority used when the process is initially started. Service activities such as changing process priority or obtaining

a mutex have no impact on base priority. Retained priority is the priority that a process will be restored to when it releases ownership of a mutex. The retained priority is typically the current priority of the process when it obtained ownership of the mutex. If the SET_PRIORITY service is invoked on a process that owns a mutex, the retained priority will be updated to the new priority value. The GET_PROCESS_STATUS service can be used to obtain the base and current priority.

2.3.2.5 Process Waiting Queue

When a running process makes a service request that causes the process to block, the process's state is set to WAITING, and the process is placed on a process queue. Associated with each potentially blocking resource (e.g., queuing port, buffer, semaphore, event, mutex) is a process queue. Processes are placed on the process queue based on a First In/First Out (FIFO) or priority **queuing discipline** defined when the create service for the resource was invoked and the current priority of the process (when queuing discipline is priority order).

2.3.2.6 Process Preemption Locking

Support for process preemption locking is provided. Basic preemption locking services (LOCK_PREEMPTION and UNLOCK_PREEMPTION) permit a running process to own a preemption lock that prevents other processes from preempting its use of processor core resources. The owning process could be executing a critical section or accessing resources shared by multiple processes of the same partition. A critical section may be for access for specific areas of memory, certain physical devices, or simply the normal calculations and activity of a particular process. The preemption locking services manage a single preemption lock mutex predefined by the O/S for the partition. When a process owns the preemption lock mutex, other processes are prevented from owning the preemption lock through either process scheduling (e.g., partition only has a single processor core or all processes that use the preemption lock mutex have an affinity for the same processor core) or by blocking (e.g., when processes running on other processor cores invoke LOCK_PREEMPTION). A set of mutex services (see Section 2.3.6.2.3) provide additional mutual exclusive access services that can be utilized concurrently by multiple processes (i.e., multiple processes accessing different mutexes).

The LOCK_PREEMPTION and UNLOCK _PREEMPTION services are used to manage the preemption locking mutex and the partition's lock level. When the partition's lock level value is greater than 0, the running process that locked preemption cannot be preempted by another process in the same partition. When the lock level value is 0, a running process can be preempted by another ready process.

Preemption locking has no impact on the scheduling of other partitions. It only affects scheduling of processes within a partition.

When a process owns the preemption lock (or a mutex), the process is not permitted to block as part of calling an ARINC 653 service (an error code will be returned).

COMMENTARY

The ability to intervene with the normal rescheduling operations of the operating system does not imply that the application software is directly controlling the O/S software. Since lock preemption is provided by the O/S and all resultant actions and effects are known beforehand, the integrity of the O/S remains intact and unaffected by the service requests.

2.3.2.6.1 Concurrent Running Process Preemption Locking

When a partition is configured to have access to multiple processor cores, multiple processes may run concurrently. Only one running process at a time can have preemption locked (i.e., own the preemption lock mutex). Other running processes that attempt to lock preemption will be blocked and set to wait on the preemption lock mutex's queue in a FIFO basis. When a running process no longer has preemption locked, a process will be removed from the preemption lock mutex's queue, be configured as having locked preemption, and scheduled to run on one of the processor cores the process is eligible to run on. Other processes can concurrently run on other processor cores assigned to the partition as long as they do not attempt to lock preemption.

A running process may have been assigned a core affinity. Use of preemption locking does not impact the assigned core affinity (i.e., the process is scheduled to run using its assigned core affinity).

COMMENTARY

When multiple processes use lock preemption (e.g., to control access to different resources or execution sequences), some process blocking may occur even though the resources and execution sequences are used by separate sets of processes. Separate mutexes (see Section 2.3.6.2.3) can be used in lieu of lock preemption to separately manage access to each resource and execution sequence.

2.3.3 Time Management

Time management is an important characteristic of an O/S used in real-time systems. Time is unique and independent of partition execution within an integrated module. All time values or capacities are related to this unique time and are not relative to any partition execution. The O/S manages the system clock and uses it to provide partition time windows for module scheduling; deadline, periodicity, and delays for process scheduling; and time-outs for intrapartition and interpartition communication. These mechanisms are defined through attributes or services.

From an application perspective, the system clock is an O/S-managed resource that monotonically increases with elapsed time at a fixed rate.

A time capacity is assigned to each process as part of process creation. It represents the response time given to a process for satisfying its processing requirements.

COMMENTARY

Time capacity is assigned to each periodic and aperiodic process and is used to set the process's deadline. A blocked process's deadline will continue to be evaluated while it is blocked. A periodic process's deadline is reset by invoking the PERIODIC_WAIT service. An aperiodic process's deadline is reset by using the stop and start services.

When a process is started (either explicitly via a service request or at the end of the initialization phase), its deadline is set to the value of current time plus time capacity. This deadline time may be postponed by means of the REPLENISH service. Note that this capacity is an absolute duration of time, not an execution time. This means that a process deadline overrun will occur even when a process is not running inside or outside the partition time window, but will be acted upon only inside a partition time window of its own partition.

COMMENTARY

Since the deadline time is absolute, configuration modifications to the partition's time windows can result in deadline overruns. Dependencies between process deadline and the partition time windows include whether a partition's duration is divided into multiple time windows.

Time replenishment determines the next deadline value as represented in Figure 2.3.3.

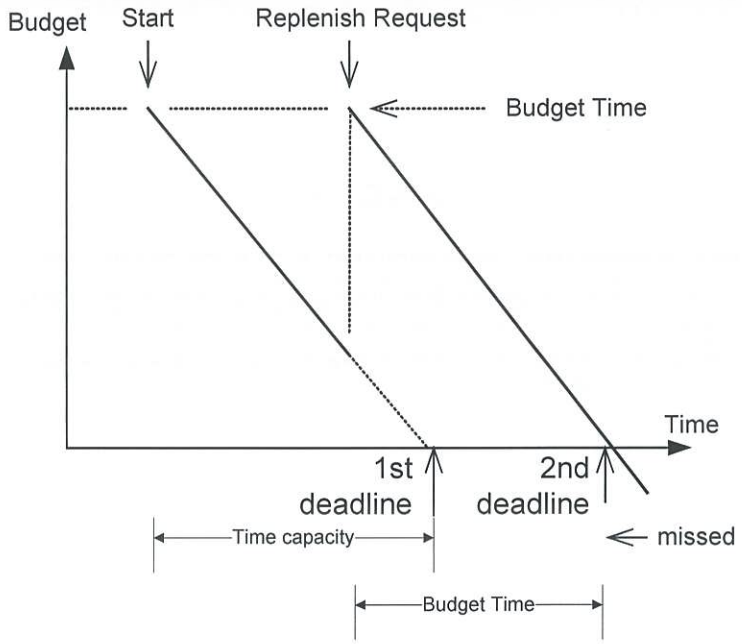

Figure 2.3.3 – Time Replenishment

As long as a process performs its entire processing without using its whole time capacity, the deadline is met. If a process requires more processing than the time capacity, the deadline is missed. When the deadline is missed, a health monitoring error will be raised (see Section 2.4).

COMMENTARY

The health monitoring configuration tables define at what level (process, partition, or module) a deadline miss error will be handled for each partition. If the allocated handler does not resolve the deadline miss error (i.e., error handler process did not modify the erroneous process' state, or an "ignore" recovery action was utilized), the subsequent behavior of the process that missed the deadline is O/S implementation dependent. It is strongly recommended (for portability reasons) that deadline miss errors be managed by health monitoring instead of relying on implementation dependent behavior. Implementation dependent behaviors could include:

- The process retains its pre-deadline miss state (e.g., still running if it was the previously running process), with a new deadline time not being set for the process until the process invokes the PERIODIC_WAIT service (i.e., expiration of a process' deadline does not affect the process' period). Invocation of PERIODIC_WAIT during the process' next period results in the process being eligible to run immediately (i.e., to perform its intended function

for its next period). Failure to invoke a 2nd PERIODIC_WAIT before the deadline calculated for the process' next period results in another deadline miss HM event.

- The process retains its pre-deadline miss state (e.g., still running if it was the previously running process), with the O/S extending the process' deadline time to its next release point (i.e., skip one process period and utilize it to complete the overrun of the previous period). Invocation of PERIODIC_WAIT during the extended period results in process waiting on its next release point. Failure to invoke PERIODIC_WAIT during the extended period results in another deadline miss HM event.

A time-out delay or deadline can expire outside of any of the partition's time windows. It is acted upon at the beginning of the partition's next time window.

COMMENTARY

The notion of time used here is local to an integrated module and is needed for time management within the partitions on that integrated module. If a partition needs a timestamp for a reason such as fault reporting, a different time may be used. That time value could come from an aircraft clock via normal interpartition communication means.

2.3.4 Memory Management

Partitions, and therefore their allocated memory spaces, are defined during system configuration and initialization. There are no memory allocation services in the APEX interface.

A memory space is a range of addresses that typically represent access to a corresponding range of physical memory. A memory space can also represent access to memory mapped registers (e.g., an I/O device). A unique set of memory region instances are utilized to configure the mapping of memory spaces into a partition's address space. Each partition (unless explicitly configured to share a memory space) will be allocated unique memory spaces. All processes of a partition will have access to the same memory spaces. This includes processes that are running on different processor cores.

If an application uses only ARINC 653 services, then no other action on the part of the application is necessary to maintain memory coherency and consistency of the data transferred through these services.

COMMENTARY

The application is responsible for resolving memory coherency and consistency concerns when data is exchanged by means other than ARINC 653 services (e.g., global variables).

A platform may contain memory and cache devices that are intended to contain the same memory range. Memory coherency is ensuring that the contents in these devices are updated when the same memory range in another device is modified. When a partition is allocated multiple processor cores, this may include ensuring coherency between contents of cache devices dedicated to each processor core. The platform may include mechanisms (including hardware) that support or provide a means to support memory coherency (e.g., cache flushing instructions).

Memory consistency involves ensuring the order of a sequence of memory updates amongst a set of processor cores (when order is important). The platform may include mechanisms (including hardware) that support or provide a means to support memory consistency (e.g., memory barrier instructions).

2.3.5 Interpartition Communication

A major part of this standard is the definition of the communication between APEX partitions.

Interpartition communication is a generic expression used in this standard. The interpartition communication definitions contained in this standard are intended to facilitate communications between ARINC 653 application partitions residing on the same integrated module or on different integrated modules, as well as communications between an ARINC 653 application partition and non-ARINC 653 equipment external to that partition's core module. Standard interpartition communication is a basic requirement for support of reusable and portable application software.

All interpartition communication is conducted via messages. A message is defined as a contiguous block of data of finite length. A message is sent from a single source to one or more destinations. The destination of a message is a partition, and not a process within a partition.

COMMENTARY

The expression "contiguous block" means a sequential data arrangement in the source and destinations memory areas. Sending a message without format conversion is achieved by copying the message from memory to memory via the communication network. This principle does not require the message to be entirely transmitted in a single packet.

The APEX interface supports communication of the full message, regardless of any message segmentation applied by the O/S. Note that this does not prohibit a partition from decomposing a large message into a set of smaller ones and communicating them individually. The O/S regards them purely as separate unrelated messages.

COMMENTARY

From the application viewpoint, a message is a collection of data which is either sent or received to/from a specific port. Depending on the port implementation, the messages allowed to be passed in that port may be either fixed or variable length.

The basic mechanism for linking partitions by messages is the channel. A channel defines a logical link between one source and one or more destinations, where the source and the destinations may be one or more partitions. It also specifies the mode of transfer of messages from the source to the destinations together with the characteristics of the messages that are to be sent from that source to those destinations.

Partitions have access to channels via defined access points called ports. A channel consists of one or more ports and the associated resources. A port provides the required resources that allow a specific partition to either send or receive messages in a specific channel. A partition is allowed to exchange messages through multiple channels via their respective source and destination ports. The channel describes a route connecting one sending port to one or several receiving ports.

From the perspective of a partition, APEX communication services support the transmission and receipt of messages via ports, and are the same regardless of the system boundaries crossed by the communicated message. Therefore, the system integrator (not the application developer)

configures the channel connections within an integrated module and the channel connections between an integrated module and components external to the integrated module.

2.3.5.1 Communication Principles

ARINC 653 communications are based on the principle of transport mechanism independence at partition level. It is assumed that the underlying transport mechanism transmits the messages and ensures that the messages leave the source port and reach the destination ports in the same order. Communications between partitions, or between partitions and external entities, use the same services and are independent of the underlying transport mechanism. Messages exchanged between two compliant ARINC 653 applications are identical regardless of whether the applications reside on the same integrated module, or on different integrated modules.

The core module is responsible for encapsulating and transporting messages, such that the message arrives at the destination(s) unchanged. Any fragmentation, segmentation, sequencing, and routing of the message data by the core module, required to transport the data from source to destination, is invisible to the application(s). The core module is responsible for ensuring the integrity of the message data, i.e., messages should not be corrupted in transmission.

At the application level, messages are atomic entities (i.e., either the whole message is received or nothing is received). Applications are responsible for assuring the data meets requirements for processing by that application. This might include range checks, voting between different sources, or other means.

It is the responsibility of the application designer, in conjunction with the system integrator, to ensure that the transport mechanism chosen meets the message transport latency and reliability required by the application.

COMMENTARY

This technique provides:

- The system integrator with freedom to optimize the allocation of partitions to processing resources within a core module or among multiple core modules.
- Portability of applications across platforms.
- Network technology upgrades without significant impact to applications.

Communication within a partition uses services that are analogous to the external communication services.

The following are limitations on the behavior of APEX communications:

1. The performance and integrity of interpartition communications is dependent on the underlying transport mechanism (i.e., communication media and protocols), which is beyond the scope of this standard. When defining the functional behavior and integrity of queuing and sampling port services, it is assumed the underlying transport mechanism supports this behavior. It is the systems integrator responsibility to specify the end-to-end behavior of the communications system, assure functional requirements are met, and communicate behavioral differences between this standard and the implementation to application providers.
2. Although data abstraction should assure parameter consistency, equivalent performance between differing transport mechanisms is not guaranteed.
3. Synchronous behavior between communication ports is not guaranteed by the APEX interface. This may or may not be a feature of the underlying system.

4. The O/S should ensure that the messages provided by the application are transmitted to the transport mechanism in the same order. The O/S should ensure that the messages received from the transport mechanism are delivered to the application in the same order. It is assumed that the transport mechanism maintains ordinal integrity of messages between source and destination.
5. Any particular message can only originate from a single source.
6. When a recipient accesses a new instance of a message, it can no longer request access to an earlier instance. It cannot be assumed that the O/S will save old versions of messages.

2.3.5.2 Message Communication Levels

Messages may be communicated across different message passing boundaries as defined below:

1. **Within core modules** – allows messages to be passed between partitions supported by the same core module. Whether the message is passed directly, or across a data bus, is configurable by the system integrator for each message source.
2. **Between the core modules** – allows messages to be passed between multiple core modules via a communication bus.
3. **Between core modules and a non-ARINC 653 component** – allows messages to be passed between core modules and devices that do not host an ARINC 653 O/S (traditional LRUs, sensors, etc.) via various communication buses.

In traditional LRUs which use an APEX interface, messages may be passed either directly between partitions on the same LRU (for LRUs supporting multiple partitions) or between partitions and the LRU buses.

Interpartition communication conducted externally across core module boundaries should conform to the appropriate message protocol. The message protocol to be used for this communication is system specific. The system partition and/or the O/S I/O device driver are responsible for communicating these messages over the communication bus, taking the physical constraints of the bus into consideration.

Characteristics of communication media may require the segmentation of an externally communicated message. This segmentation is transparent to the application software and is the responsibility of the I/O mechanisms provided by the O/S.

COMMENTARY

This version of ARINC 653 does not address the communication protocol between core modules.

2.3.5.3 Message Types

The messages may be of the types described in the following sections.

2.3.5.3.1 Fixed/Variable Length

A fixed length message has a defined constant size for every occurrence of the message. A variable length message can vary in size, and therefore, requires the source to specify the size within the message when transmitting it.

COMMENTARY

Fixed length is best suited to the transmission of measurements, commands, status, etc. Variable length is more appropriate to the transmission of messages whose amount of data varies during run time (e.g., list of airframes for the TCAS function).

2.3.5.3.2 Periodic/Aperiodic

Periodic means that the communication of a particular message is performed on a regular iterative basis. Aperiodic means that the communication is not necessarily periodic.

COMMENTARY

For periodic messages, it is usual for recipients to be designed to cope with intermittent loss of data. Typically, the last valid data sample is used by the recipient until either the continued data loss is unacceptable or a new valid sample is received.

Periodic messages are best suited to the communication of continuously varying data (e.g., Air Speed, Total Pressure). Aperiodic messages are best suited to the communication of irregular events that may occur at random intervals (e.g., due to an operator-instigated action).

No distinction is made by the O/S between periodic and aperiodic messages as the instances of message generation are implicitly defined according to the nature of the partition's runtime activation (i.e., the periodicity of a message is dictated by the periodicity at which the message is sent).

Discrete events should not be reported in single periodic messages or should be acknowledged within applications, at a level above that of the message passing protocol. Discrete events are best suited to being announced in aperiodic messages.

2.3.5.3.3 Broadcast, Multicast and Unicast Messages

This standard provides support for broadcast, multicast, and unicast messages. A broadcast message is sent from a single source to all destinations. A multicast message is sent from a single source to more than one destination. A unicast message is sent from a single source to a single destination. For queuing mode, only unicast messages are required.

COMMENTARY

The behavior of broadcast and multicast queuing is implementation dependent.

This standard does not directly support client/server messages. Client/server support may be implemented by applications on top of queuing or sampling ports.

2.3.5.4 Message Type Combinations

(This section deleted by Supplement 1.)

2.3.5.5 Channels and Ports

Channels and ports are defined as part of the system configuration activities. The definition of channels is not confined to the core module O/S but is rather distributed on the constituent core

modules and LRUs of the system. Each communication node (core module, gateway, I/O module, etc.) is assumed to be separately configurable (i.e., through configuration tables) to define how messages are handled at that node. It is the system integrator responsibility to ensure that the different nodes crossed by each channel are consistently configured. The consequence is that the source, destinations, mode of transfer, and unique characteristics of each channel cannot be changed at run time.

2.3.5.6 Modes of Transfer

Each individual port may be configured to operate in a specific mode. Two modes of transfer may be used: sampling mode and queuing mode.

COMMENTARY

It may be possible to configure communication such that ports operating in different modes (sampling or queuing) can be connected to the same channel; however, this is dependent on the underlying media and is outside the scope of this standard.

A full set of services has to be provided to the application software to accommodate the two modes of transfer, sampling mode and queuing mode, and the two transfer directions, source and destination. Only the subset of services whose functions are compatible with the port configuration (mode, transfer direction) will execute correctly when that port is addressed; others will return an appropriate value of the return code indicating that the request failed.

Sampling ports and queuing ports can be used in any mode, i.e., NORMAL, COLD_START, and WARM_START. It is assumed that the underlying system supports the behavior defined herein.

Messages are allowed to be segmented and reassembled. The segmentation and reassembling of messages are transparent to the application.

2.3.5.6.1 Sampling Mode

In the sampling mode, successive messages typically carry identical but updated data. No queuing is performed in this mode. A message remains in the source port until it is transmitted by the channel or it is overwritten by a new occurrence of the message, whichever occurs first. This allows the source partition to send messages at any time. Each new instance of a message overwrites the current message when it reaches a destination port, and remains there until it is overwritten. This allows the destination partitions to access the latest message.

COMMENTARY

The periodicity of transmission is determined by the source application in its sending of the messages. Reception of such messages is additionally influenced by the latency in communicating the messages through the intermediate communication media (i.e., backplane bus, gateway, LRU-specific media).

It is the responsibility of the underlying port implementation to properly handle data segmentation. A partial message will not be delivered to the destination port.

2.3.5.6.2 Queuing Mode

In the queuing mode, each new instance of a message may carry uniquely different data and is not allowed to overwrite previous ones during the transfer. No message should be unintentionally lost in the queuing mode.

The ports of a channel operating in queuing mode are allowed to buffer multiple messages in message queues. A message sent by the source partition is stored in the message queue of the source port until it is transmitted by the channel. When the message reaches the destination port, it is stored in a message queue until it is received by the destination partition. A specific protocol is used to manage the message queues and transmit messages in the source port and in the destination port in a First In/First Out (FIFO) order (see Section 2.3.5.1, Communication Principles item d). This allows the source and destination ports to manage the situation where either the source or destination queues are full. Application software is responsible for handling overflow when the queuing port is full.

Variable length messages are supported in the queuing mode.

COMMENTARY

Since the amount of data carried by a message may vary considerably, encapsulating the data in a fixed length message may not be acceptable because of excessive usage of RAM and bus bandwidth. Applications are therefore expected to manage different types of messages, each with an appropriate length. The channel does not need to distinguish between those different types of messages, but is only required to accept transmitting successive messages with different lengths.

It is not a requirement to have true aperiodic transmission of messages between the source and destination ports.

A partial message will not be delivered to the destination port in the queuing mode.

2.3.5.7 Port Attributes

In order for the required communication resources to be provided to the partitions, a set of unique attributes needs to be defined for each port. These attributes enable the O/S to control and maintain the operation of ports and portions of channels located within the core module (or the LRUs). The port attributes are defined in the configuration tables or by the application when invoking a service call to create the port. The required port attributes are described in the following sections.

2.3.5.7.1 Partition Identifier

The partition identifier denotes the partition which is allowed to have access to the port.

2.3.5.7.2 Port Name

The port name attribute consists of a pattern that uniquely identifies the port within the partition that has access to it. This name is intended to be used by the application software to designate the port.

COMMENTARY

By using port names instead of addressing directly the source/destination partitions, the application software is more independent of the communication network architecture. The configuration of port names is the responsibility of the system integrator. It should be noted that a well-chosen port name should refer to the data passed in the port rather than to the producer/consumers of those data (e.g., "MEASURED ELEVATOR POSITION").

2.3.5.7.3 Mode of Transfer

The mode of transfer attribute denotes the mode (i.e., sampling mode or queuing mode), which is expected to be used to manage the messages in the port.

2.3.5.7.4 Direction

The direction attribute indicates whether the port allows messages to be sent (the partition is viewed as a source) or received (the partition is viewed as a destination).

2.3.5.7.5 Maximum Message Size

The maximum message size attribute defines the maximum number of bytes a single message may contain for the port.

2.3.5.7.6 Maximum Number of Messages

The maximum number of messages attribute applies to messages in the queuing mode only. This attribute defines the maximum number of messages the queuing port must handle without loss.

COMMENTARY

The maximum message size and maximum number of messages attributes are used to define storage areas in the memory space of the partition, allowing messages passed in the port to be temporarily buffered. Depending on the mode of transfer and the transfer direction, different message storage areas may be required:

1. Sampling mode, send direction – No particular message storage is required when a send request is issued by the application software.
2. Sampling mode, receive direction – A one-message area is required to buffer the last correct message received by the partition.
3. Queuing mode, send direction – A message queue, managed on a FIFO basis, is required to buffer the successive messages to be sent. The queue is filled upon send requests issued by the application software, and cleared as the O/S moves the messages from the queue.
4. Queuing mode, receive direction – A message queue, managed on a FIFO basis, is required to buffer the successive correct messages received by the partition. The queue is filled as the O/S moves correct received messages to the queue, and cleared upon receive requests issued by the application software.

COMMENTARY

In all cases, applications should allocate enough memory space to assure the configuration defined maximum message size can be received, regardless of any application protocol defined size assumptions.

2.3.5.7.7 Refresh Period

The refresh period attribute applies to messages received in the sampling mode only. This allows the O/S to determine whether new messages arrive at a specified period from the last message, regardless of the receive request rate.

2.3.5.7.8 Port Mapping

The port mapping attributes define the connection between that port and the physical communication media (e.g., memory, backplane bus,). The port mapping consists of attributes defining a physical address and/or procedure that may be used to map the port. Use, meaning, and limitations of these attributes are implementation dependent.

COMMENTARY

Messages received by a port may originate either from another port of the same integrated module or via an inter-module communications mechanism. Messages sent by a port are routed to one or more other ports of the module and/or the inter-module communications mechanism in the sampling mode, whereas they are routed to either one other port of the integrated module or the inter-module communications mechanism in the queuing mode.

Ports can be mapped to physical addresses or to procedures (e.g., device drivers), both of which can map the ports to a backplane or communications device. A procedure mapping might be used when an intelligent I/O processor is not available to move data to/from an external device. In addition, the O/S implementer may define any other suitable mapping mechanisms. Any mechanism for mapping ports must ensure that multiple partitions do not have write access to the memory space to which the port is being mapped whether it is accomplished by a physical address mapping, a procedure mapping, or some other mechanism. The notion of shared memory can only be supported when the O/S controls the access to the memory location.

2.3.5.8 Port Control

The core module resources required to manage a port are statically defined at build time and initialized after power is applied to the core module. Once the ports assigned to a partition have been initialized, the application software is allowed to perform send or receive operations in those ports. The O/S does not ensure that the channels connected to the ports have been entirely initialized. The reason is that the channels may cross different nodes, and these nodes may not necessarily be initialized simultaneously (i.e., if located on different cabinets).

Creating a port associates, **for the current partition,** a port identifier with a port name. The port identifier is an O/S-defined value associated with the port name at the core module initialization. Use of the port identifier allows more efficient access to the port resources than use of the port name.

COMMENTARY

If messages are transmitted in a channel and some portions of that channel have not been initialized, then a fault will be detected either by the destination ports (message freshness check) in the sampling mode, or by the source port (lack of response) in the queuing mode.

Either fixed or variable length messages are allowed to be communicated via a port. A length indication is provided by the application software when sending a message and by the O/S when the application software receives a message.

Depending on its mode of transfer and its transfer direction, a port operates in different ways:

1. Sampling mode, send direction – Each new message passed by an application's send request overwrites the previous one. Each occurrence of a message cannot be transmitted more than once.
2. Sampling mode, receive direction – Each correct (internally consistent) new received message is copied to the temporary storage area of the port where it overwrites the previous one. This area is allowed to be polled at random times, upon application software receive requests. The copied message and a validity indicator are returned to the application software. The validity indicator indicates whether the age of the copied message is consistent with the required refresh period defined for the port. The age of the copied message is defined as the difference between the value of the system clock (when READ_SAMPLING_MESSAGE is called) and the value of the system clock when the O/S copied the messages from the channel into the destination port.
3. Queuing mode, send direction – Each new message passed by an application's send request is temporarily stored in the send message queue of the port. If the queue is full or contains insufficient space for the entire message to be stored, then the requesting process of the partition enters the waiting state or the send request may be cancelled. The queued messages are segmented as required to satisfy the underlying communication media before they are transmitted in FIFO order.
4. Queuing mode, receive direction – Each correct (internally consistent) new received message is moved to the receive message queue. The oldest message in the message queue is removed from the queue and passed to the application software upon receipt of the receive request. If the message queue is empty, the requesting process may enter the waiting state or the receive request may be cancelled. The channel protocol may prevent further segments from being received when the receive message queue is full, and request resending of failed segments. This applies only to unicast transmissions.

2.3.5.9 Process Queuing Discipline

The communication between partitions is done by processes which are sending or receiving messages. In queuing mode, processes may wait on a full message queue (sending direction) or on an empty message queue (receiving direction). Rescheduling of processes will occur when a process attempts to wait. The use of time-outs will limit or avoid waiting times.

Processes waiting for a port in queuing mode are queued in FIFO or priority order. In the case of priority order, for the same priority, processes are also queued in FIFO order. The queuing discipline is defined by the application as part of the service call used for creation of the port.

COMMENTARY

When queued messages requiring responses are passed between partitions with a non-zero wait time, care must be used to ensure that faults (such as excessive delays or missing data from the other partition) do not result in undesired effects upon the receiving partition.

2.3.6 Intrapartition Communication

Provisions for processes within a partition to communicate with each other without the overhead of the global message passing system are included as part of this standard. The intrapartition communication mechanisms are buffers, blackboards, semaphores, events, and mutexes. Buffers and blackboards provide general inter-process communication (and synchronization), whereas semaphores, events, and mutexes provide inter-process synchronization. All intrapartition message

passing mechanisms must ensure atomic message access (i.e., partially written messages cannot be read).

There are no configuration table attributes associated with intrapartition communications. Attributes associated with intrapartition communications are specified by the application when invoking the corresponding create operation.

As with process and port creation, the amount of memory required to create intrapartition communication mechanisms is allocated from the partition's memory resources, which is defined at system build time. An application can create as many intrapartition communication mechanisms as the partition's memory resources will support.

2.3.6.1 Buffers and Blackboards

Inter-process communication of messages is conducted via buffers and blackboards. These mechanisms support the communication of messages between multiple source and destination processes. The communication is indirect in that participating processes address the buffer or blackboard rather than the opposing processes directly, thus providing a level of process independence. Buffers allow the queuing of messages whereas blackboards do not allow message queuing.

Rescheduling of processes will occur when a process must wait for a message. The use of time-outs will limit or avoid waiting times.

2.3.6.1.1 Buffers

In a buffer, each new instance of a message may carry uniquely different data and therefore is not allowed to overwrite previous ones during the transfer.

Buffers are allowed to store multiple messages in message queues. A message sent by the sending process is stored in the message queue in FIFO order. When using buffers, no messages are lost (the sender will block if the queue is full). The number of messages that can be stored in a buffer is determined by the defined maximum number of messages, which is specified at buffer creation time.

The CREATE_BUFFER operation creates a buffer for use by any of the processes in the partition. At creation, the buffer's maximum message size, maximum number of messages, and queuing discipline are defined.

Processes waiting on a buffer are queued in FIFO or priority order. In the case of priority order, processes with the same priority are queued in FIFO order.

If there are processes waiting on a buffer and if the buffer is not empty, the queuing discipline algorithm (FIFO or priority order) will be applied to determine which queued process will receive the message. The O/S will remove this process from the process queue and will put it in the ready state. The O/S will also remove the message from the buffer message queue.

Rescheduling of processes will occur when a process attempts to receive a message from an empty buffer or to send a message to a full buffer. The calling process will be queued for a specified (as part of the service call) duration of real-time. If a message is not received or is not sent in that amount of time, the O/S will automatically remove the process from the queue and put it in the ready state (unless another process has suspended it).

2.3.6.1.2 Blackboards

For a blackboard, no message queuing occurs. Any message written to a blackboard remains there until the message is either cleared or overwritten by a new instance of the message. This allows sending processes to display messages at any time, and receiving processes to access the latest message at any time.

The CREATE_BLACKBOARD operation creates a blackboard for use by any of the process in the partition. At creation, the blackboard's maximum message size is defined.

A process can read a message from a blackboard, display a message on a blackboard, or clear a blackboard.

Rescheduling of processes will occur when a process attempts to read a message from an empty blackboard. The calling process will be queued for a specified (as part of the service call) duration of real-time. If a message is not displayed in that amount of time, the O/S will automatically remove the process from the queue and put it in the ready state (unless another process has suspended it).

When a message is displayed on the blackboard, the O/S will remove all waiting processes from the process queue and will put them in the ready state. The message remains on the blackboard.

When a blackboard is cleared, it becomes empty.

2.3.6.2 Semaphore, Events, and Mutexes

Inter-process synchronization is conducted using semaphores, events, and mutexes. Semaphores and mutexes provide controlled access to resources. Events support control flow between processes by notifying the occurrences of conditions to waiting processes.

2.3.6.2.1 Semaphores

The semaphores defined in this standard are counting semaphores, and are commonly used to provide controlled access to partition resources. A process waits on a semaphore to gain access to the partition resource, and then signals the semaphore when it is finished. A semaphore's value in this example indicates the number of currently available partition resources.

The CREATE_SEMAPHORE operation creates a semaphore for use by any of the process in the partition. At creation, the semaphore's initial value, maximum value, and queuing discipline are defined.

Processes waiting on a semaphore are queued in FIFO or priority order. In the case of priority order, for the same priority, processes are also queued in FIFO order (with the oldest waiting process being at the front of the FIFO).

The WAIT_SEMAPHORE operation decrements the semaphore's value if the semaphore is not already at its minimum value of zero. If the value is already zero, the calling process may optionally be queued until either the semaphore is signaled or until a specified (as part of the service call) duration of real-time expires.

The SIGNAL_SEMAPHORE operation increments the semaphore's value. If there are processes waiting on the semaphore, the queuing discipline algorithm, FIFO or priority order, will be applied to determine which queue process will receive the signal. Signaling a semaphore where processes are waiting increments and decrements the semaphores value in one request. The end result is there are still no available resources, and the semaphore value remains zero.

Rescheduling of processes will occur when a process attempts to wait on a zero value semaphore, and when a semaphore is signaled that has processes queued on it.

When a process is removed from the queue, either by the semaphore being signaled or by the expiration of the specified time-out period, the process will be moved into the ready state (unless another process has suspended it).

2.3.6.2.2 Events

An event is a communication mechanism which permits notification of an occurrence of a condition to processes which may wait for it. An event is composed of a bi-valued state variable (states called "up" and "down") and a set of waiting processes (initially empty).

COMMENTARY

On the arrival of an aperiodic message, a receiving process may send a synchronous signal (event) to another process.

Processes within the same partition can SET and RESET events and also WAIT on events that are created in the same partition.

The CREATE_EVENT operation creates an event object for use by any of the processes in the partition. Upon creation, the event is set in the state "down."

To indicate occurrence of the event condition, the SET_EVENT operation is called to set the specified event in the up state. All of the processes (unless suspended by another process) waiting on that event are then moved into the ready state, and process scheduling takes place.

The transition of waiting processes to the ready state should appear to be atomic, so that all processes are available for scheduling at the same time. There is no queuing discipline assignment associated with events. The processes that were waiting on an event will be eligible for scheduling on a priority followed by FIFO (from the time the process began waiting on the event) basis.

The RESET_EVENT operation sets the specified event in the state "down."

The WAIT_EVENT operation moves the calling process from the running state to the waiting state if the specified event is "down" and if there is a specified time-out that is greater than 0. The calling process goes on executing if the specified event is "up" or if there is a conditional wait (event down and time-out is equal to 0).

2.3.6.2.3 Mutexes

Mutex services are supported. A mutex is a synchronization object commonly used to control access to partition resources. Only one process at a time can own a specific mutex. Each mutex includes a priority value that a process's current priority is raised to when the mutex is obtained. When a process releases a mutex, the current priority of the process is typically restored to the process priority value it had when it obtained the mutex. If the priority of a process is modified using SET_PRIORITY while it owns a mutex, it is restored to the last priority value it was set to.

A mutex must be created using the CREATE_MUTEX operation during the partition's initialization phase before it can be used. A name, priority, and queuing discipline are given at mutex creation. This name is local to the partition and is not an attribute of the partition configuration table. The creation of mutex (e.g., names used, number of mutexes) for one partition has no impact on the creation of mutexes for other partitions.

A process obtains ownership of a mutex using the ACQUIRE_MUTEX operation. A process can own at most one mutex at a time (i.e., preventing potential deadlock issues when two or more processes obtain a series of mutexes in different orders). When owning a mutex, the process is prevented from blocking as a result of a service call.

If another process attempts to obtain a mutex that is already owned, the process is put on the mutex's waiting queue. Processes waiting on a mutex are queued in FIFO or priority order. In the case of priority order, for the same priority, processes are also queued in FIFO order (with the oldest waiting process being at the front of the FIFO).

A process releases ownership of a mutex using the RELEASE_MUTEX operation. Each mutex incorporates a locking count. A process must invoke RELEASE_MUTEX the same number of times as it invoked ACQUIRE_MUTEX for the mutex to be released. A process can also release ownership of a mutex using the RESET_MUTEX operation. Using this operation results in the mutex's locking count to be set to zero, immediately releasing the process from ownership.

A process can be tested for ownership of a mutex using the GET_PROCESS_MUTEX_STATE operation. If the process owns a mutex (including the preemption lock mutex), the corresponding identifier is returned. If the process does not own a mutex, a value representing no mutex owned is returned.

Other than the GET_PROCESS_MUTEX_STATE service, the mutex services cannot be utilized with the preemption lock mutex (an error will be returned if attempted).

2.4 Health Monitor

In general, Health Monitor (HM) functions are responsible for responding to and reporting hardware, application, and O/S software errors and failures. ARINC 653 supports HM by providing HM configuration tables and an application level error handler process.

The HM helps to isolate errors and to prevent failures from propagating. Components of the HM are contained within the following software elements:

- O/S – All HM implementations will use the ARINC 653 O/S. The O/S uses the HM configuration tables to respond to pre-defined faults.
- Application Partitions – Where errors are system specific and determined from logic or calculation, application partitions may be used, passing error data to the O/S via the HM service calls or to an appropriate system partition.
- System Partitions – System partitions can be used by the platform integrators for error management.

Figure 2.4 illustrates the ARINC 653 HM decision logic. The decision logic is further described in the following subsections and Section 2.5.2.3. The O/S evaluates the error based on the HM decision logic and HM configuration tables. If the HM decision logic determines this error has been assigned to this partition as a process level error and a process level error handler is present for this partition (e.g., error handler created and use of the error handler has been enabled by the partition being in NORMAL mode), the O/S will activate the process level error handler to run. When an error handler is created during the initialization phase, the error handler has not yet been enabled. Errors that occur during the initialization phase result in the enforcement of the action defined in the Partition HM table for the detected error.

COMMENTARY

It is implementation dependent how the HM decision processing and application of module and partition recovery actions are enforced. The execution environment (e.g., processor privilege mode, partition context, process context switches) used for HM decision processing is implementation dependent.

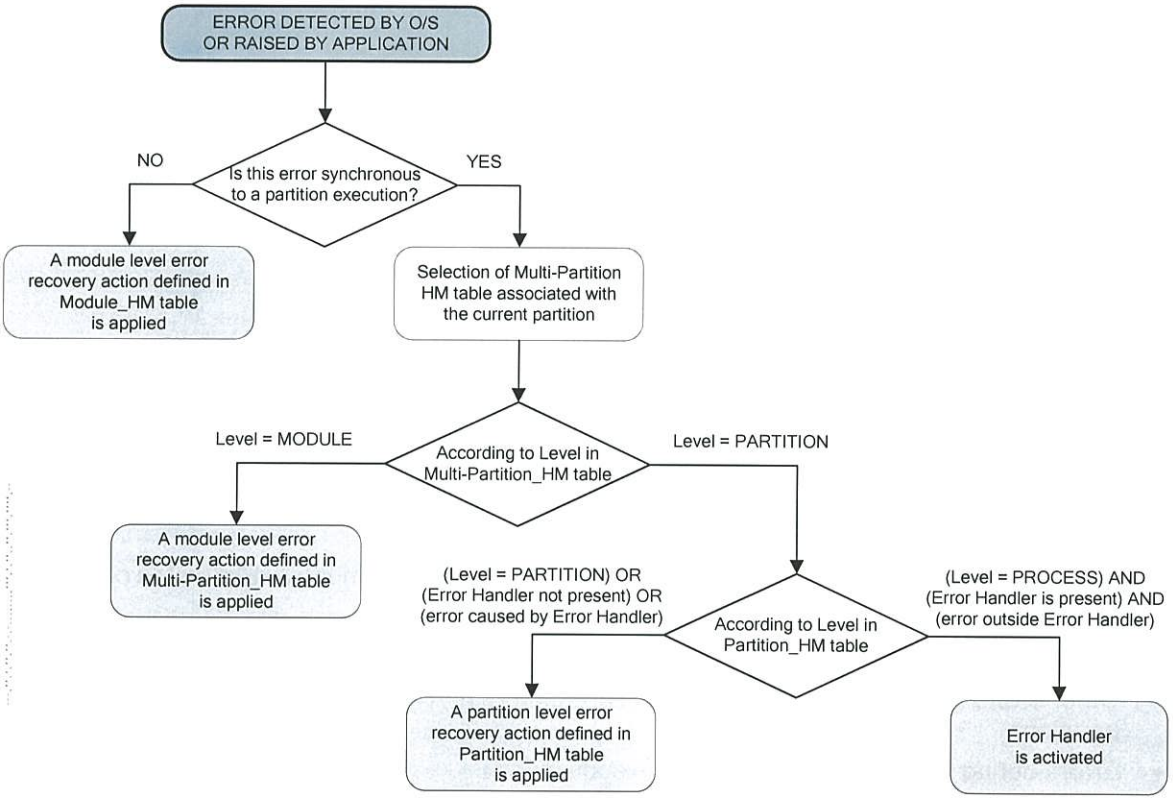

Figure 2.4 – HM Decision Logic

2.4.1 Error Levels

Errors may occur at the module, partition, or process level. These error levels are described in the following sections. The level of an error is defined in the module HM, multi-partition HM and partition HM tables in accordance with the detected error and the state of the system.

COMMENTARY

System-level errors and their reporting mechanisms are outside the scope of this document. It is the responsibility of the system integrator to ensure the system-level error handling and lower-level error handling are consistent, complete and integrated.

2.4.1.1 Process Level Errors

A process level error impacts one or more processes in a partition or the entire partition. Examples of process level errors are:

- Application error raised by an application process
- Illegal O/S request
- Process execution errors (stack overflow, memory violation, etc.)
- Process not completing its execution cycle before its deadline

The HM will not violate partitioning when handling process level errors.

2.4.1.2 Partition Level Errors

A partition level error impacts only one partition. Examples of partition level errors are:

- Partition configuration table error during partition initialization
- Partition initialization error
- Errors that occur during process management
- Execution errors caused by the error handler process

The HM will not violate partitioning when handling partition level errors.

2.4.1.3 Module Level Errors

A module level error may impact all the partitions within an integrated module. Examples of module level errors are:

- Module configuration table error during integrated module initialization
- Other errors during integrated module initialization (e.g., checksum error detected for one partition)
- Errors during system-specific function execution
- Errors during partition switching
- Power fail

2.4.2 Error Detection and Response

Errors are detected by several elements:

- Hardware – memory protection violation, privilege execution violation, stack overflow, zero divide, timer interrupt, I/O error
- O/S – configuration, deadline
- Application – failure of sensor, discrepancy in a multiple redundant output

The specific list of errors and where they are detected is implementation specific.

Error response depends on the detected error and on the operational state in which the error is detected. The operational state of the system (core module initialization, system-specific function, partition switching, partition initialization, process management, process execution, etc.) is managed by the O/S. The response to errors in each operational state is implementation dependent. For example, a hardware failure affecting a core module I/O could either cause the O/S to shut down all the partitions immediately or signal the problem to the partition only when a

process of this partition would use the failing I/O device. The system integrator chooses the error management policy.

2.4.2.1 Process Level Error Response Mechanisms

Responses to process level errors are determined by the application programmer using a special (highest priority, no process identifier) process of the partition referred to as the error handler process. The error handler process is active (i.e., present) in NORMAL mode only. The error handler process can identify the error and the causing process via the GET_ERROR_STATUS service and then takes the recovery action at the process level (e.g., stop followed by start process) or at the partition level (e.g., set partition mode to IDLE, COLD_START, or WARM_START). An error code is assigned to several process level errors (e.g., numeric error corresponds to overflow, divide by zero, floating-point underflow, etc.). To maintain application portability, error codes are implementation independent. The process level error codes and examples of each are listed below:

- Deadline_Missed – process deadline violation
- Application_Error – error raised by an application process
- Numeric_Error – during process execution, error types of overflow, divide by zero, floating-point error
- Illegal_Request – illegal O/S request by a process
- Stack_Overflow – process stack overflow
- Memory_Violation – during process execution, error types of memory protection, supervisor privilege violation
- Hardware_Fault – during process execution, error types of memory parity, I/O access error
- Power_Fail – notification of power interruption (e.g., to save application specific state data)

COMMENTARY

Some process level errors prevent the process from being eligible for scheduling **(i.e., result in the process going to the Faulted state; see Section 2.3.2.2.1.1)**. The set of process level errors included in this category are referred to as fatal errors and result in the process being set to the faulted state.

Typically, memory violations, stack overflows, and numeric errors caused by a process are fatal errors. Illegal requests and hardware faults caused by a process may also be considered fatal errors (implementation dependent).

Power failures generally will be asynchronous to the active partition schedule (i.e., generally no assurance which partition time window the event will occur in). Handling of a power failure as a process level error may require power hold-up capabilities that can cover multiple partition time windows **(i.e., to hold up power until the partition's next time window becomes active)**. Power hold-up capabilities are module dependent.

If an error handler process does not exist within a partition, then the partition level error response mechanism is invoked by the O/S.

Process level errors that occur inside the error handler process are always promoted to partition level (irrespective of the HM configuration) and consequently handled by the partition level error response mechanisms. The error handler process failing during its execution cycle to obtain the identification of at least one causing process via the GET_ERROR_STATUS service will also be promoted to the partition level (Illegal_Request).

COMMENTARY

It is possible for an application within a partition to fail in such a way that the error is not correctly reported, or not reported at all, by the application itself (e.g., the error handler process). The system integrator may need to account for this in the overall system design (rate monitors, watchdog timers, etc.).

2.4.2.1.1 Process Level HM when Partitions have Multiple Processor Cores

Multiple processor cores assigned to the same partition may impact the process level health monitoring function.

The error handler process, when activated, is the highest priority process that can be scheduled. Only one instance of the error handler process will run at a time. A service (invocable during initialization) is provided that controls whether other processes are scheduled on other processor cores concurrently with the error handler process, including a process that has locked preemption. A process that has caused an error (deadline overrun excepted) should not continue to run in parallel with the error handler process. Such processes (processes that raise application errors are accepted) are set to be in the faulted state.

When a partition is assigned a single processor core, the error handler (being highest priority) will interrupt the currently running process, regardless of whether this process caused the error or not.

When a partition is assigned multiple processor cores, the CONFIGURE_ERROR_HANDLER service can be used to control the scheduling of the error handler process. Configuring the error handler process to a specific processor core will result in that processor core always being utilized to run the error handler. Configuring the error handler process to CORE_AFFINITY_NO_PREFERENCE means there is no preference as to which processor core is utilized. When CORE_AFFINITY_NO_PREFERENCE is utilized, it is O/S implementation dependent as to how a processor core is selected. Once selected, the error handling process runs on the selected processor core until it completes. The error handler process will have a default affinity of CORE_AFFINITY_NO_PREFERENCE.

2.4.2.2 Partition Level Error Response Mechanisms

Responses to partition level errors are handled in the following way:

- The O/S looks up the error code response action in the HM configuration tables.
- The O/S performs the response identified in the configuration table.

2.4.2.2.1 Partition Level HM when Partitions have Multiple Processor Cores

The partition level health monitoring enforces the partition level error recovery actions (e.g., idle the partition, restart the partition). The recovery action will be applied against the partition on all processor cores being used by the partition (not just against the processor core that was used to detect or process the error). Processes within a partition could temporarily continue to run while the recovery action is being enforced (i.e., enforcement of the recovery action does not necessarily occur instantaneously). When other processes temporarily continue to run on the other processor cores allocated to the partition, the O/S prevents these processes from interfering with enforcement of partition level health monitoring actions.

2.4.2.3 Module Level Error Response Mechanisms

Responses to module level errors are handled in the following way:

- The O/S looks up the error code response action in the HM configuration tables.
- The O/S performs the response identified in the configuration table.

2.4.2.3.1 Module Level HM when Partitions have Multiple Processor Cores

The module level health monitoring enforces the multiple module level error recovery actions (e.g., shutdown the module, reset the module). The recovery action will be applied against the integrated module on all processor cores associated with the integrated module (not just against the processor core that was used to detect or process the error). Processes within an active partition could temporarily continue to run while the recovery action is being enforced (i.e., enforcement of the recovery action does not necessarily occur instantaneously). When other processes temporarily continue to run on the other processor cores allocated to an active partition, the O/S prevents these processes from interfering with enforcement of module level health monitoring actions.

2.4.3 Recovery Actions

Recovery actions include actions at the module level, partition level and process level.

Recovery actions at all levels include the capability to ignore an error. However, an error should only be ignored if it does not leave module, partition, or process in an undefined state. For instance, ignoring memory violations may leave the partition in an undefined state.

2.4.3.1 Module Level Error Recovery Actions

The recovery action is specified for a module error in the Module HM configuration table depending on system state. Possible recovery actions are listed below:

- Ignore
- Shutdown the integrated module
- Reset the integrated module
- Recovery Actions defined by the implementation (O/S implementation dependent)

The O/S must support the capability of adding platform specific recovery actions.

COMMENTARY

The context in which an O/S performs module level error recovery actions is implementation dependent.

2.4.3.2 Partition Level Error Recovery Actions

The recovery action is specified for partition errors in the HM configuration tables. For each partition, the response for the error type takes into account the partition behavior capability (re-settable or not, degraded mode, etc.). The recovery actions are listed below:

- Ignore
- Stop the partition (IDLE)
- Restart the partition (COLD_START or WARM_START)
- Recovery Actions defined by the implementation (O/S implementation dependent)

The O/S must support the capability of adding platform specific recovery actions.

COMMENTARY

The context in which an O/S performs partition level error recovery actions is implementation dependent.

2.4.3.3 Process Level Error Recovery Actions

The application supplier defines an error handler process of the partition for process level errors. Possible recovery actions are listed below:

- Ignore, log the failure but take no action
- Ignore the error n times before action recovery
- Stop faulty process and re-initialize it from entry address
- Stop faulty process and start another process
- Stop faulty process (assume partition detects and recovers)
- Restart the partition (COLD_START or WARM_START)
- Stop the partition (IDLE)
- Attempt a recovery and resume the faulty process

COMMENTARY

Resuming a faulty process that caused a fatal error (e.g., memory violation) will result in the execution of the same instruction that caused the error. Recovery of most fatal exceptions may not be feasible.

When there is a process level error handler, recovery actions are always performed in the context of the partition associated with the erroneous process.

The HM design may allow support for Ada exceptions, but there is potential for conflicts between the Ada runtime system and the HM.

2.5 Configuration Considerations

It is a goal of this specification to define an environment that allows full portability and reuse of application source code. Applications must however be integrated into a system configuration which satisfies the functional requirements of the applications and meets the availability and integrity requirements of the aircraft functions. A single authoritative organization will be responsible for the integration. This organization will be known as the system integrator. The system integrator could be the platform supplier, the airframe manufacturer, a third party, or a combination of all participants.

The system integrator is responsible for allocating partitions to core modules. The resulting configuration should allow each partition access to its required resources and should ensure that each application's availability and integrity requirements are satisfied. The system integrator should, therefore, know the timing requirements, memory usage requirements, and external interface requirements of each partition to be integrated.

It is the responsibility of the application developer to configure the processes within a partition.

The system integrator should ensure that all partitions have access to the external data required for their correct execution. This data is passed around the system in the form of messages. Each

message should, therefore, be unique and identifiable. From an application point of view, the only means of identifying a message is the port identifier, which is local to the partition. It is the responsibility of the system integrator to ensure that message formats are compatible between the sending and receiving partitions.

2.5.1 Configuration Information Required by the Integrator

To enable configuration of partitions onto core modules, the integrator requires the following information about each partition as a minimum:

- Memory requirements
- Period
- Duration
- Port attributes
- Processor core requirements (when multiple logical processor cores are available). Each logical processor core is associated with a unique physical processor core.

Use of this information will allow the partitions to be allocated to the integrated modules in a configuration which ensures that the memory and time requirements of each partition can be satisfied. The location of partitions on integrated modules dictates the routes of messages, and so the system integrator must also provide the mapping between nodes on the message paths.

2.5.2 Configuration Tables

Configuration tables are required by the O/S for ensuring that the system is complete during initialization and to enable communications between partitions. Configuration tables are static data areas accessed by the O/S. They cannot be accessed directly by applications, and they are not part of the O/S.

XML is used to describe the configuration as discussed in Section 5 of this document. Appendices G and H define the XML schema for the data needed to specify any ARINC 653 configuration. This XML schema is integrated by the ARINC 653 O/S implementers into the schema for their implementation.

2.5.2.1 Configuration Tables for System Initialization

(This material deleted by Supplement 3.)

2.5.2.2 Configuration Tables for Interpartition Communication

(This material deleted by Supplement 3.)

2.5.2.3 Configuration Tables for Health Monitor

The Health Monitor (HM) uses configuration tables to handle each occurring error. These tables are the Module HM table, Multi-Partition HM tables and Partition HM tables.

For the Module HM table:

- The O/S uses the Module HM table when an error is detected outside a partition time window (e.g., during core module initialization, partition switch, etc.).
- The Module HM table defines the recovery action (e.g., shut down the module, reset the module, etc.) assigned to module level errors detected outside a partition time window.

- The recovery action is set according to the detected error and the system state (e.g., initialization, switching partition, etc.).
- There is one Module HM table per integrated module, and it is configured by the integrated module supplier.

For the Multi-Partition HM tables:

- The O/S uses the Multi-Partition HM table when an error is detected inside a partition time window.
- The Multi-Partition HM table defines global behavior of HM for a set of partitions.
- The Multi-Partition HM table defines the error level (Module or Partition) **assigned** to errors detected inside a partition time window.
- When the error is at module level, the Multi-Partition HM table defines also the module recovery action (e.g., shut down the module, reset the module, etc.).
- The error level and module recovery action (when applicable) are set according to the detected error. Since the Multi-Partition HM table is used only when one partition is inside its time window, there is only one system state (i.e., partition execution).
- There are one or many Multi-Partition HM tables per module, all of them configured by the system integrator. Each Multi-Partition HM table is **assigned** to one or many partitions. These **assignments** are configured also by the system integrator.

For the Partition HM tables:

- The O/S uses the Partition HM table when an error is detected inside a partition time window and the corresponding Multi-Partition HM table indicates the Partition level for this particular error in the current partition.
- The Partition HM table defines local behavior of HM for a single partition.
- The Partition HM table defines the error level (Partition or Process) **assigned** to errors detected inside a partition time window.
- The Partition HM table defines also the partition recovery actions (e.g., stop the partition, restart the partition in warm or cold mode). This partition recovery action will be performed when the error is at partition level, and also when the error is at process level, but no error handler is created for that partition.
- When the error is at process level, the Partition HM table defines also the **assigned** error codes (e.g., APPLICATION_ERROR, NUMERIC_ERROR, etc.).
- The error level, partition recovery action, and error codes are set according to the detected error. The list of error identifiers in the Partition HM table is restricted to the errors which can be raised by the partition itself (e.g., deadline missed, memory violation, etc.).
- There is one Partition HM table per partition. Each table is configured by the respective application supplier.

2.6 Verification

The system integrator will be responsible for verifying that the complete system fulfills its functional requirements when applications are integrated and for ensuring that availability and integrity requirements are met. Verification that application software fulfills its functional requirements will be carried out by the supplier of the application.

The system integrator, through use of supporting tools, will be responsible for verifying the contents of the configuration tables against the capabilities supported by the core modules (e.g., available memory, available processor cores) and the capabilities required by the applications hosted on the core modules.

3.0 SERVICE REQUIREMENTS

3.1 Service Request Categories

This section specifies the service requests corresponding to the functions described in Section 2.0 of this document. The requests are grouped into the following major categories:

1. Partition Management
2. Process Management
3. Time Management
4. Memory Management
5. Interpartition Communication
6. Intrapartition Communication
7. Health Monitoring

Each service request has a sample specification written in the APEX specification grammar. The service requests in this section describe the functional requirements of APEX services. The data type names, service request names, parameter names, and order of parameters are definitive, the implementation is not (i.e., the order of the error tests are not defined by this standard). Some data type declarations are common to all categories, and are duplicated in each of the following sections for clarity. The exceptions are RETURN_CODE_TYPE and SYSTEM_TIME_TYPE, which are referenced from the other sections of this document.

In order to preserve the integrity of information managed by the services, the requesting process is assumed to never be preempted during the execution of a service, except at the scheduling points which are explicitly mentioned in the semantic description.

When a partition is assigned multiple cores, multiple processes may run concurrently on the assigned cores. If a process attempts to access an ARINC 653 object (e.g., process, semaphore, port) or group of objects that are already being accessed by another process, the process may temporarily not make progress in order to prevent the process from introducing object inconsistencies.

Partition level objects required for create services may be statically or dynamically allocated. For a static system, the object attributes of the create service need to be checked against the corresponding statically configured objects. For a dynamic system, the object attributes need to be checked for reasonableness (e.g., range checking) and non-violation of system limits (e.g., sufficient memory available) before the object is created.

Since this interface may be applied to Integrated Modular Avionics (IMA), it does not include services (or service requirements) which only pertain to a specific architecture design, implementation, or partition configuration. This list of services is viewed as a minimum set of the total services which may be provided by the individual systems.

The interface defined in this standard provides the operations necessary for basic multi-process execution. The inclusion of extra services pertinent to a specific system would not necessarily violate this standard, but would make the application software which uses these additional services less portable.

The convention followed for error cases in these service requests is to not specify assignments for output parameters other than the return code. Implementers should be aware that, in many cases, programming practices will require some value to be assigned to all output parameters.

Ada and C language specifications for the types, records, constants, and service request interfaces used in this section are defined in Appendices D and E. Unless specifically annotated as implementation dependent, the values specified in these appendices should be implemented as defined. For C, the APEX specifications will be available via the "ARINC653.h" header file.

The definition of ARINC 653 NAME_TYPE is based on an 'n-character array (i.e., fixed length).

COMMENTARY

To avoid unnecessary complexity in the underlying O/S, there are no restrictions on the allowable characters for NAME_TYPE. However, it is recommended that usage be limited to printable characters. The O/S should treat the contents of NAME_TYPE objects as case insensitive.

The use of a NULL character is not required in an object of NAME_TYPE. If a NAME_TYPE object contains NULL characters, the first NULL character in the array of characters should be considered to define the end of the name.

3.1.1 Return Code Data Type

This section contains the return code data type declaration common to all APEX services. The return codes are listed only in this section for clarity. The allowable return codes and their descriptions are:

```
NO_ERROR            request valid and operation performed
NO_ACTION           system's operational status unaffected by request
NOT_AVAILABLE       the request cannot be performed immediately
INVALID_PARAM       parameter specified in request invalid
INVALID_CONFIG      parameter specified in request incompatible with current
                    configuration (e.g., as specified by system integrator)
INVALID_MODE        request incompatible with current mode of operation
TIMED_OUT           time-out associated with request has expired
```

The distinction between INVALID_PARAM and INVALID_CONFIG is that INVALID_PARAM denotes invalidity regardless of the system's configuration whereas INVALID_CONFIG denotes conflict with the current integrator-specified configuration and so may change according to the degree of configuration imposed. Note that the setting of INVALID_PARAM may sometimes be redundant for strongly typed languages.

The return code data type declaration is:

```
type RETURN_CODE_TYPE      is (NO_ERROR,
                               NO_ACTION,
                               NOT_AVAILABLE,
                               INVALID_PARAM,
                               INVALID_CONFIG,
                               INVALID_MODE,
                               TIMED_OUT);
```

3.1.2 OUT Parameters Values

The validity of OUT parameters is dependent on the RETURN_CODE value returned by the considered service.

3.2 Partition Management

This section contains types and specifications related to partition management.

3.2.1 Partition Management Types

These types are used in partition management services:

```
type OPERATING_MODE_TYPE    is (IDLE, COLD_START, WARM_START, NORMAL);
type PARTITION_ID_TYPE      is a numeric type;
type LOCK_LEVEL_TYPE        is a numeric type;

type START_CONDITION_TYPE   is (NORMAL_START,
                                PARTITION_RESTART,
                                HM_MODULE_RESTART,
                                HM_PARTITION_RESTART);
```

Where:

```
NORMAL_START           is a normal power-up.
PARTITION_RESTART      is due to COLD_START or WARM_START by the partition
                       itself, through the SET_PARTITION_MODE service. It could
                       also be through a non-HM module action (e.g., as part of a
                       Part 2 service).
HM_MODULE_RESTART      is a recovery action taken at module level by the HM.
HM_PARTITION_RESTART   is a recovery action taken at partition level by the HM.
```

The NORMAL_START/PARTITION_RESTART can be set if there is a manual module/partition reset through external means.

```
type PROCESSOR_CORE_ID_TYPE         is a numeric type;
type NUM_CORES_TYPE                 is a numeric type;

type PARTITION_STATUS_TYPE is record
   PERIOD                           : SYSTEM_TIME_TYPE;
   DURATION                         : SYSTEM_TIME_TYPE;
   IDENTIFIER                       : PARTITION_ID_TYPE;
   LOCK_LEVEL                       : LOCK_LEVEL_TYPE;
   OPERATING_MODE                   : OPERATING_MODE_TYPE;
   START_CONDITION                  : START_CONDITION_TYPE;
   NUM_ASSIGNED_CORES               : NUM_CORES_TYPE;
end record;
```

The RETURN_CODE_TYPE is common to all APEX services and is defined in Section 3.1.1 of this document.

The SYSTEM_TIME_TYPE is common to many APEX services and is defined in Section 3.4.1 of this document.

3.2.2 Partition Management Services

The partition management services are:

```
GET_PARTITION_STATUS
SET_PARTITION_MODE
```

3.2.2.1 GET_PARTITION_STATUS

The GET_PARTITION_STATUS service request is used to obtain the status of the current partition.

```
procedure GET_PARTITION_STATUS
   (PARTITION_STATUS : out PARTITION_STATUS_TYPE;
    RETURN_CODE      : out RETURN_CODE_TYPE) is

normal
   PARTITION_STATUS  := current value of partition status;
   RETURN_CODE       := NO_ERROR;

end GET_PARTITION_STATUS;
```

The return codes for this service are explained below.

GET_PARTITION_STATUS	
Return Code Value	Commentary
NO_ERROR	Successful completion

3.2.2.2 SET_PARTITION_MODE

The SET_PARTITION_MODE service request is used to set the operating mode of the current partition to normal after the application portion of the initialization of the partition is complete. The service is also used for setting the partition back to idle (partition shutdown) and to cold start or warm start (partition restart) when a fault is detected and processed.

When operating mode is set to NORMAL, the main process used during the partition initialization mode does not continue to run after completion of the mode transition.

```
procedure SET_PARTITION_MODE
   (OPERATING_MODE : in  OPERATING_MODE_TYPE;
    RETURN_CODE    : out RETURN_CODE_TYPE) is

error
   when (OPERATING_MODE does not represent an existing mode) =>
      RETURN_CODE := INVALID_PARAM;
   when (OPERATING_MODE is NORMAL and current mode is NORMAL) =>
      RETURN_CODE := NO_ACTION;
   when (OPERATING_MODE is WARM_START and current mode is COLD_START) =>
      RETURN_CODE := INVALID_MODE;

normal
   set current partition's operating mode := OPERATING_MODE;
   if (OPERATING_MODE is IDLE) then
      shut down the partition;
   end if;
   if (OPERATING_MODE is WARM_START or OPERATING_MODE is COLD_START) then
      inhibit process scheduling and switch back to initialization mode;
   end if;
   if (OPERATING_MODE is NORMAL) then
      set to READY all previously started (not delayed) aperiodic processes
      (unless the process was suspended);
      set release point of all previously delay started aperiodic processes
       to the system clock time plus their delay times;
      set first release points of all previously started (not delayed) periodic
       processes to the partition's next periodic processing start;
      set first release points of all previously delay started periodic processes
       to the partition's next periodic processing start plus their delay times;
      -- at their release points, the processes are set to READY (if not DORMANT)
      calculate the DEADLINE_TIME of all non-dormant processes in the partition;
```

```
      -- a DEADLINE_TIME calculation may cause an overflow of the underlying
      -- clock. If this occurs, HM is invoked with an illegal request error code
      set the partition's lock level to zero;
      if (an error handler process has been created) then
         enable the error handler process for execution and fault processing;
      end if;
      activate the process scheduling;
   end if;
   RETURN_CODE := NO_ERROR;

end SET_PARTITION_MODE;
```

The return codes for this service are explained below.

SET_PARTITION_MODE

Return Code Value	Commentary
NO_ERROR	Successful completion
INVALID_PARAM	OPERATING_MODE does not represent an existing mode
NO_ACTION	OPERATING_MODE is normal and current mode is NORMAL
INVALID_MODE	OPERATING_MODE is WARM_START and current mode is COLD_START

3.3 Process Management

This section contains types and specifications related to process management. The scope of a process management service is restricted to a partition.

3.3.1 Process Management Types

These types are used process management services:

```
type PROCESS_ID_TYPE          is a numeric type;
type PROCESS_NAME_TYPE        is a n-character string;
type PRIORITY_TYPE            is a numeric type;
type STACK_SIZE_TYPE          is a numeric type;
type LOCK_LEVEL_TYPE          is a numeric type;
type SYSTEM_ADDRESS_TYPE      is language dependent;
type PROCESS_STATE_TYPE       is (DORMANT, READY, RUNNING, WAITING, FAULTED);
type DEADLINE_TYPE            is (SOFT, HARD);
type PROCESS_INDEX_TYPE       is a numeric type;


type PROCESS_ATTRIBUTE_TYPE is record
   PERIOD           : SYSTEM_TIME_TYPE;
   TIME_CAPACITY    : SYSTEM_TIME_TYPE;
   ENTRY_POINT      : SYSTEM_ADDRESS_TYPE;
   STACK_SIZE       : STACK_SIZE_TYPE;
   BASE_PRIORITY    : PRIORITY_TYPE;
   DEADLINE         : DEADLINE_TYPE;
   NAME             : PROCESS_NAME_TYPE;
end record;

type PROCESS_STATUS_TYPE is record
   DEADLINE_TIME    : SYSTEM_TIME_TYPE;
   CURRENT_PRIORITY : PRIORITY_TYPE;
   PROCESS_STATE    : PROCESS_STATE_TYPE;
   ATTRIBUTES       : PROCESS_ATTRIBUTE_TYPE;
end record;
```

The RETURN_CODE_TYPE is common to all APEX services and is defined in Section 3.1.1 of this document.

The SYSTEM_TIME_TYPE is common to many APEX services and is defined in Section 3.4.1 of this document.

3.3.2 Process Management Services

A process must be created during the partition initialization phase before it can be used.

The process management services are:

```
GET_PROCESS_ID
GET_PROCESS_STATUS
CREATE_PROCESS
SET_PRIORITY
SUSPEND_SELF
SUSPEND
RESUME
STOP_SELF
STOP
START
DELAYED_START
LOCK_PREEMPTION
UNLOCK_PREEMPTION
GET_MY_ID
INITIALIZE_PROCESS_CORE_AFFINITY
GET_MY_PROCESSOR_CORE_ID
GET_MY_INDEX
```

3.3.2.1 GET_PROCESS_ID

The GET_PROCESS_ID service request allows a process to obtain a process identifier by specifying the process name.

```
procedure GET_PROCESS_ID
   (PROCESS_NAME   : in  PROCESS_NAME_TYPE;
    PROCESS_ID     : out PROCESS_ID_TYPE;
    RETURN_CODE    : out RETURN_CODE_TYPE) is

error
   when (there is no current partition process named PROCESS_NAME) =>
      RETURN_CODE := INVALID_CONFIG;

normal
   PROCESS_ID  :=  unique identifier for the current partition assigned to the
      process named PROCESS_NAME;
   RETURN_CODE := NO_ERROR;

end GET_PROCESS_ID;
```

The return codes for this service are explained below.

```
GET_PROCESS_ID
Return Code Value       Commentary
NO_ERROR                Successful completion
INVALID_CONFIG          There is no current partition process named PROCESS_NAME
```

3.3.2.2 GET_PROCESS_STATUS

The GET_PROCESS_STATUS service request returns the current status of the specified process. The current status of each of the individual processes of a partition is available to all processes within that partition.

If this service is invoked on a process that owns the preemption lock mutex, the current priority of the process will be returned as the maximum process priority.

```
procedure GET_PROCESS_STATUS
   (PROCESS_ID     : in  PROCESS_ID_TYPE;
    PROCESS_STATUS : out PROCESS_STATUS_TYPE;
    RETURN_CODE    : out RETURN_CODE_TYPE) is

error
   when (PROCESS_ID does not identify an existing process for
      the current partition) =>
      RETURN_CODE := INVALID_PARAM;

normal
   RETURN_CODE    := NO_ERROR;
   PROCESS_STATUS := current value of process status;

end GET_PROCESS_STATUS;
```

The return codes for this service are explained below.

GET_PROCESS_STATUS

Return Code Value	Commentary
NO_ERROR	Successful completion
INVALID_PARAM	PROCESS_ID does not identify an existing process for the current partition

3.3.2.3 CREATE_PROCESS

The CREATE_PROCESS service request creates a process and returns an identifier that denotes the created process. For each partition, as many processes as the pre-allocated memory space will support can be created. The creation of processes (e.g., names used, number of processes) for one partition has no impact on the creation of processes for other partitions.

Defining the PERIOD of a process with an INFINITE_TIME_VALUE inherently defines an aperiodic process.

Defining the TIME_CAPACITY of a process with an INFINITE_TIME_VALUE inherently defines a process that does not have a deadline time (i.e., will not cause a DEADLINE_MISSED HM event). Deadline times can be defined for periodic and aperiodic processes.

```
procedure CREATE_PROCESS
   (ATTRIBUTES  : in  PROCESS_ATTRIBUTE_TYPE;
    PROCESS_ID  : out PROCESS_ID_TYPE;
    RETURN_CODE : out RETURN_CODE_TYPE) is

error
   when (insufficient storage capacity for the creation of the specified
      process or maximum number of processes have been created) =>
      RETURN_CODE := INVALID_CONFIG;
   when (the process named ATTRIBUTES.NAME is already created for
      the current partition) =>
      RETURN_CODE := NO_ACTION;
   when (ATTRIBUTES.STACK_SIZE out of range) =>
```

```
      RETURN_CODE := INVALID_PARAM;
   when (ATTRIBUTES.BASE_PRIORITY out of range) =>
      RETURN_CODE := INVALID_PARAM;
   when (ATTRIBUTES.PERIOD out of range) =>
      RETURN_CODE := INVALID_PARAM;
   when (ATTRIBUTES.PERIOD is finite (i.e., periodic process) and
      ATTRIBUTES.PERIOD is not an integer multiple of partition period defined
      in the configuration tables) =>
      RETURN_CODE := INVALID_CONFIG;
   when (ATTRIBUTES.TIME_CAPACITY out of range) =>
      RETURN_CODE := INVALID_PARAM;
   when (ATTRIBUTES.PERIOD is finite (i.e., periodic process) and
      ATTRIBUTES.TIME_CAPACITY is greater than ATTRIBUTES.PERIOD) =>
      RETURN_CODE := INVALID_PARAM;
   when (operating mode is NORMAL) =>
      RETURN_CODE := INVALID_MODE;

normal
   create a new process with the process attributes set to ATTRIBUTES;
   set the process state to DORMANT;
   initialize process context, unique process index, and stack;
   set the process's core affinity to the default process core affinity value;
   PROCESS_ID  := unique identifier assigned by the O/S to the created process;
   RETURN_CODE := NO_ERROR;

end CREATE_PROCESS;
```

The return codes for this service are explained below.

CREATE_PROCESS

Return Code Value	Commentary
NO_ERROR	Successful completion
INVALID_CONFIG	Insufficient storage capacity for the creation of the specified process or maximum number of processes have been created
NO_ACTION	The process named ATTRIBUTES.NAME is already created for the current partition
INVALID_PARAM	ATTRIBUTES.STACK_SIZE out of range
INVALID_PARAM	ATTRIBUTES.BASE_PRIORITY out of range
INVALID_PARAM	ATTRIBUTES.PERIOD out of range
INVALID_CONFIG	ATTRIBUTES.PERIOD is finite and ATTRIBUTES.PERIOD is not an integer multiple of partition period defined in the configuration tables
INVALID_PARAM	ATTRIBUTES.TIME_CAPACITY out of range
INVALID_PARAM	ATTRIBUTES.PERIOD is finite and ATTRIBUTES.TIME_CAPACITY is greater than ATTRIBUTES.PERIOD
INVALID_MODE	Operating mode is NORMAL

3.3.2.4 SET_PRIORITY

The SET_PRIORITY service request changes a process's current priority. If the process's current state is ready, the process is considered as having been in the ready state with the set priority for the shortest elapsed time (i.e., other processes at the same priority eligible for the same processor core(s) will be selected to run before this process).

Process rescheduling is performed after this service request only when the process whose priority is changed is in the ready or running state.

COMMENTARY

If the process is waiting on a resource, the process queue for that resource may or may not be reordered based on the new priority. Therefore, from the viewpoint of portability, applications should not rely on the reordering of the process queue.

```
procedure SET_PRIORITY
   (PROCESS_ID      : in  PROCESS_ID_TYPE;
    PRIORITY        : in  PRIORITY_TYPE;
    RETURN_CODE     : out RETURN_CODE_TYPE) is

error
   when (PROCESS_ID does not identify an existing process for
      the current partition) =>
      RETURN_CODE := INVALID_PARAM;
   when (PRIORITY is out of range) =>
      RETURN_CODE := INVALID_PARAM;
   when (specified process is in the DORMANT state) =>
      RETURN_CODE := INVALID_MODE;

normal
   if (the specified process owns a mutex) then
      -- current priority of the process cannot be modified without
      -- impacting mutex properties
      set the retained priority of the specified process to PRIORITY;
   else
      set the current priority of the specified process to PRIORITY;
   end if;
   check for process rescheduling;
   -- A running process may be preempted by the process whose priority
   -- was modified if the running process does not have preemption locked
   RETURN_CODE := NO_ERROR;

end SET_PRIORITY;
```

The return codes for this service are explained below.

SET_PRIORITY

Return Code Value	Commentary
NO_ERROR	Successful completion
INVALID_PARAM	PROCESS_ID does not identify an existing process
INVALID_PARAM	PRIORITY is out of range
INVALID_MODE	The specified process is in the DORMANT state

3.3.2.5 SUSPEND_SELF

The SUSPEND_SELF service request suspends the execution of the current process, if aperiodic. The process remains suspended until the RESUME service request is issued or the specified time-out value expires.

Periodic processes cannot be suspended.

COMMENTARY

If a process suspends (using SUSPEND service) an already self-suspended process, it has no effect.

```
procedure SUSPEND_SELF
   (TIME_OUT     : in  SYSTEM_TIME_TYPE;
    RETURN_CODE  : out RETURN_CODE_TYPE) is

error
   when (current process owns a mutex or
         current process is error handler process) =>
      RETURN_CODE := INVALID_MODE;
   when (TIME_OUT is out of range) =>
      -- e.g., calculation causes overflow of underlying clock
      RETURN_CODE := INVALID_PARAM;
   when (process is periodic) =>
      RETURN_CODE := INVALID_MODE;

normal
   if (TIME_OUT is zero) then
      RETURN_CODE := NO_ERROR;
   else
      set the current process's state to WAITING;
      if (TIME_OUT is not infinite) then
         initiate a time counter with duration TIME_OUT;
      end if;
      ask for process scheduling;
      -- The current process is blocked and will return in READY state
      -- by expiration of time-out or by RESUME service request from another
      -- process
      if (expiration of the time-out) then
         RETURN_CODE := TIMED_OUT;
      else  -- RESUME service request from another process
         RETURN_CODE := NO_ERROR;
         -- RESUME service request will stop the time counter.
      end if;
   end if;

end SUSPEND_SELF;
```

The return codes for this service are explained below.

SUSPEND_SELF

Return Code Value	Commentary
NO_ERROR	Resumed by another process or TIME_OUT is zero
INVALID_MODE	Current process owns a mutex or current process is error handler process
INVALID_PARAM	TIME_OUT is out of range
INVALID_MODE	Process is periodic
TIMED_OUT	The specified time-out is expired

3.3.2.6 SUSPEND

The SUSPEND service request allows the current process to suspend the execution of any aperiodic process except itself. The suspended process remains suspended until resumed by another process. If the process is pending in a queue at the time it is suspended, it is not removed from that queue. When it is resumed, it will continue pending unless it has been removed from the queue (by availability of the resource it was waiting on or expiration of a time-out) before the end of its suspension. If the process is waiting on a timed-wait time counter, the process will continue to wait on the time counter when it is resumed unless the time counter has expired.

Periodic processes cannot be suspended.

COMMENTARY

If a process suspends an already suspended process, it has no effect.

A process may suspend any other process asynchronously, including a running process (feasible when a partition is allocated two or more processor cores). Though this practice is not recommended, there may be partitions that require this capability.

The state of a process obtained using the GET_PROCESS_STATUS service may have changed in between calling the service and attempting to suspend the process.

```
procedure SUSPEND
  (PROCESS_ID   : in  PROCESS_ID_TYPE;
   RETURN_CODE : out RETURN_CODE_TYPE) is

error
   -- Cannot suspend a process that has called LOCK_PREEMPTION or ACQUIRE_MUTEX
   when (PROCESS_ID is a process that owns a mutex or is waiting on a mutex's
      queue) =>
      RETURN_CODE := INVALID_MODE;
   when (PROCESS_ID does not identify an existing process for
      the current partition or identifies the current process) =>
      RETURN_CODE := INVALID_PARAM;
   when (the state of the specified process is DORMANT or FAULTED) =>
      RETURN_CODE := INVALID_MODE;
   when (specified process is periodic) =>
      RETURN_CODE := INVALID_MODE;

normal
   if (specified process has already been suspended,
       either through SUSPEND or SUSPEND_SELF) then
      RETURN_CODE := NO_ACTION;
   else
      set the specified process state to WAITING;
      RETURN_CODE := NO_ERROR;
   end if;

end SUSPEND;
```

The return codes for this service are explained below.

SUSPEND

Return Code Value	Commentary
NO_ERROR	Successful completion
NO_ACTION	Specified process has already been suspended
INVALID_PARAM	PROCESS_ID does not identify an existing process for the current partition or identifies the current process
INVALID_MODE	PROCESS_ID is a process that currently owns a mutex or is waiting on a mutex's queue
INVALID_MODE	The state of the specified process is DORMANT or FAULTED
INVALID_MODE	Specified process is periodic

3.3.2.7 RESUME

The RESUME service request allows the current process to resume a previously suspended process (i.e., via SUSPEND or SUSPEND_SELF services). The resumed process will become ready if it is not waiting on a process queue (e.g., resource associated with a queuing port, buffer, semaphore, event) or a time delay associated with the TIMED_WAIT service.

A periodic process cannot be suspended, so it cannot be resumed.

```
procedure RESUME
   (PROCESS_ID   : in  PROCESS_ID_TYPE;
    RETURN_CODE : out RETURN_CODE_TYPE) is

error
   when (PROCESS_ID does not identify an existing process for
         the current partition or identifies the current process) =>
         RETURN_CODE := INVALID_PARAM;
   when (the state of the specified process is DORMANT) =>
         RETURN_CODE := INVALID_MODE;
   when (PROCESS_ID identifies a periodic process and is not FAULTED) =>
         RETURN_CODE := INVALID_MODE;
   when (identified process is not a suspended process and is not FAULTED) =>
         RETURN_CODE := NO_ACTION;

normal
   if (the specified process was suspended with a time-out) then
      stop the affected time counter;
   end if;
   if (the specified process is not waiting on a process queue or TIMED_WAIT
      time delay or DELAYED_START time delay) then
      set the specified process state to READY;
      check for process rescheduling;
      -- A running process may be preempted by the resumed process
      -- if the running process does not have preemption locked
   end if;
   RETURN_CODE := NO_ERROR;

end RESUME;
```

The return codes for this service are explained below.

RESUME

Return Code Value	Commentary
NO_ERROR	Successful completion
NO_ACTION	Identified process is not a suspended process and is not FAULTED
INVALID_PARAM	PROCESS_ID does not identify an existing process for the current partition or identifies the current process
INVALID_MODE	The state of the specified process is DORMANT
INVALID_MODE	PROCESS_ID identifies a periodic process and is not FAULTED

3.3.2.8 STOP_SELF

The STOP_SELF service request allows the current process to stop itself. No return code is returned to the requesting process procedure.

The error handler process uses this service when it has completed processing reported process-level errors.

COMMENTARY

This service should not be called when the partition is in the WARM_START or the COLD_START mode. In this case, the behavior of this service is not defined.

```
procedure STOP_SELF is

normal
   if (current process owns the preemption lock mutex) then
      set the partition's LOCK_LEVEL counter to zero (i.e., enable preemption);
      release the partition's lock preemption mutex per RESET_MUTEX semantics
       ignoring PREEMPTION_LOCK_MUTEX error check;
   end if;
   set the current process state to DORMANT;
   prevent the specified process from causing a deadline overrun fault;
   check for process rescheduling;
   -- A process interrupted by the error handler process may continue
   -- running, especially if it had preemption locked

end STOP_SELF;
```

3.3.2.9 STOP

The STOP service request makes a process ineligible for processor resources until another process issues the START service request.

This service allows the current process to stop the execution of any process except itself. When a process stops another process that is currently waiting in a process queue, the stopped process is removed from the process queue.

COMMENTARY

Stopping (e.g., by the error handler) a process that has preemption locked may result in improper operation of the application. Stopping a process does not impact the state of sampling ports, queuing ports, semaphores, events, blackboards, and buffers. If the process being stopped owns a mutex other than the preemption lock mutex, the mutex is not released. The mutex can be released by another process (e.g., the error handler process) using the RESET_MUTEX service.

```
procedure STOP
   (PROCESS_ID   : in  PROCESS_ID_TYPE;
    RETURN_CODE : out RETURN_CODE_TYPE) is

error
   when (PROCESS_ID does not identify an existing process for
      the current partition or identifies the current process) =>
      RETURN_CODE := INVALID_PARAM;

   when (the state of the specified process is DORMANT) =>
      RETURN_CODE := NO_ACTION;
```

```
normal
   if (specified process owns the preemption lock mutex) then
      set the partition's LOCK_LEVEL counter to zero (i.e., enable preemption);
      release the partition's lock preemption mutex per RESET_MUTEX semantics
       ignoring PREEMPTION_LOCK_MUTEX error check;
   end if;
   set the specified process state to DORMANT;
   if (specified process is waiting in a process queue) then
      remove the process from the process queue;
   end if;
   stop any time counters associated with the specified process;
   prevent the specified process from causing a deadline overrun fault;
   ask for process scheduling;
   -- process could have been running on another core
   RETURN_CODE := NO_ERROR;

end STOP;
```

The return codes for this service are explained below.

STOP

Return Code Value	Commentary
NO_ERROR	Successful completion
INVALID_PARAM	PROCESS_ID does not identify an existing process for the current partition or identifies the current process
NO_ACTION	The state of the specified process is DORMANT

3.3.2.10 START

The START service request initializes all attributes of a process to their default values and resets the runtime stack of the process. If the partition is in the NORMAL mode, the process' deadline expiration time and next release point are calculated.

This service allows the current process to start the execution of another process during runtime.

```
procedure START
   (PROCESS_ID  : in  PROCESS_ID_TYPE;
    RETURN_CODE : out RETURN_CODE_TYPE) is

error
   when (PROCESS_ID does not identify an existing process for
      the current partition) =>
      RETURN_CODE := INVALID_PARAM;
   when (the state of the specified process is not DORMANT) =>
      RETURN_CODE := NO_ACTION;
   when (DEADLINE_TIME calculation is out of range) =>
      -- e.g., calculation causes overflow of underlying clock
      RETURN_CODE := INVALID_CONFIG;

normal
   if (the process is an aperiodic process) then
      -- Start_aperiodic_process
      set the current priority of specified process to its base priority;
      reset context and stack;
      if (operating mode is NORMAL) then
         set the specified process state to READY;
         set the DEADLINE_TIME value for the specified process to
          (current system clock plus TIME_CAPACITY);
```