WIND™

AN INTEL COMPANY

VXWORKS® 653

PROGRAMMER'S GUIDE

3.0.1.1

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1-510-748-4100
Facsimile: +1-510-749-2010

For additional contact information, see the Wind River website:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

VxWorks® 653

Programmer's Guide, 3.0.1.1

6 February 2017

Contents

1

Overview

Overview of VxWorks 653

This documentation describes the VxWorks 653 real-time operating system (RTOS) and how to use its run-time facilities to develop embedded, safety-critical applications and systems.

VxWorks 653 fully complies with the *Avionics Application Software Standard Interface*, ARINC Specification 653 Part 1, Supplement 3: Required Services.

A subset of VxWorks 653 is ready for use in systems requiring certification to Level A of the RTCA DO-178B/DO-178C avionics software guidelines. VxWorks 653 is, therefore, suitable for safety-critical applications.

Overview of a VxWorks 653 Module

A VxWorks 653 module is the system controlled by the module OS (MOS). Unless stated otherwise, this documentation assumes you are working within one module.

Within a module, VxWorks 653 supports complete separation between applications and between applications and the MOS. As a result, applications can interact with each other only through explicit mechanisms that the MOS controls. Applications cannot affect the operation of the module, except in a controlled manner through resources that the MOS explicitly allocates to them.

Each application runs in a discrete partition. The MOS controls the partitions by providing time and space partitioning and memory management services. Partitions manage their own resources within the time slot that the MOS provides. Performance is optimized by keeping as many routine calls as possible within the partition.

Each partition contains a partition-level OS (the partition OS) with a set of OS services. VxWorks 653 provides the VxWorks Cert partition OS, which supports APEX interfaces.

VxWorks 653 supports warm start and cold start of partitions and of the entire module.

Overview of the VxWorks Cert Partition OS

The VxWorks Cert partition OS includes its own set of objects (for example, threads, semaphores, and mutexes), other libraries, and internal scheduling. The MOS performs the following for VxWorks Cert:

- timer facility
- some scheduling
- interpartition communication

VxWorks Cert can access only its own memory heap. VxWorks Cert drivers are used and are granted access to devices based on the XML configuration.

A subset of VxWorks Cert is ready for use in systems requiring certification to Level A of the RTCA DO-178B/DO-178C avionics software guidelines.

Run-time System

Applications are compiled against the appropriate partition OS header files and are linked against the libraries available within the partition. At boot time, the MOS loads the partition OS, its application and the configuration vector.

The VxWorks Cert partition OS offers the kernel API for C applications. Also, VxWorks Cert lets applications use the APEX interfaces or use the C++ language. You can add application or third-party header files and libraries to the compiling and linking mechanisms for the partition OS.

Applications call routines located in their partition OS. The partition OS completes the routine autonomously if it provides the requested service. Otherwise, if the application's privileges permit, the partition OS makes a system call to the MOS. For example, a system call occurs when the I/O subsystem calls **RAISE_APPLICATION_ERROR( )** or when a message is sent to an application in another partition.

Cert Versus Non-Cert API Differences

In some cases, there may be differences in the behavior of an API function when used in a cert versus non-cert environment.

Any specific differences are provided in the API reference documentation.

In general, most of the differences are due to the requirement that a running system in a cert environment must not release memory that was allocated during the initialization phase. This affects APIs related to explicit memory block manipulation as well as task handling (for example, whether a task is destroyed or suspended) and so on.

Key MOS Services for VxWorks Cert

The MOS does the following for each VxWorks Cert partition OS:

- Allocates system resources
- Schedules partitions

- Traps exceptions on behalf of the partition OS
- Defines and enforces partition boundaries
- Loads partitions
- Passes messages between partitions using ports and channels
- Performs system calls on behalf of the applications
- Monitors the health of partitions and the system

Supported Target Architectures

The following target architectures are supported:

- PowerPC QorIQ P3041 (e500mc)
  - Multi-core is supported with reference BSP:
    - **fsl_p3041_ds**
- PowerPC QorIQ P4080 (e500mc)
  - Multi-core is supported with reference BSP:
    - **fsl_p4080_ds**
- PowerPC QorIQ T2080 (e6500)
  - Multi-core is supported with reference BSP:
    - **fsl_t2080_qds**

For more information, please refer to the following documents:

- *VxWorks 653 MOS BSP Developer's Guide*
- *VxWorks 653 POS BSP Developer's Guide*
- *VxWorks 653 POS BSP Reference*

Loading and Booting

For details, see the *VxWorks 653 MOS BSP Developer's Guide*.

Run-time Model

The MOS handles system calls from each partition and validates all arguments of each system call before running it. Applications that use the VxWorks Cert partition OS have the complete set of VxWorks Cert intertask communication mechanisms available to them for use within a partition.

As well, applications that use APEX libraries to provide ARINC 653 support have additional capabilities. APEX provides partition management, process management, and time management that conform to the ARINC 653 specification. APEX provides messages, channels, and ports for interpartition communication, as well as buffers, blackboards, semaphores, and events for intra-partition communication For more information, see Communicating Between Partitions on page 21 and Communicating within APEX Partitions on page 25.

RTCA DO-178B/DO-178C Certification

To support Level A certification to the RTCA DO-178B/DO-178C avionics software guidelines, a cert subset of VxWorks 653 is required.

This release does not yet include the certification evidence products. Please contact your Wind River representative for more information.

The cert subset is selected to comply with the objectives of RTCA DO-178B/DO-178C. Selection is based on the deterministic nature of the code. The subset excludes operations that can compromise the integrity of safety-critical systems (for example, dynamically deallocating memory).

After you develop, debug, and fine-tune an application on the full VxWorks 653, you can move it to the cert subset.

Because VxWorks 653 configuration does not prevent you from including debugging components in a cert image, it is your responsibility to ensure all debugging components are removed before the application is deployed.

Recommendations to Ensure RTCA DO-178B/C Level A Certification

The following recommendations are made to ensure VxWorks 653 is used in a manner consistent with the Level A objectives defined by RTCA DO-178B/C:

- System objects and resources (for example, memory, queues, tasks, and semaphores) must be allocated only when an application is initialized.
- The system must be configured so that allocating memory is not possible after an application is initialized.
- The system must be configured so that system objects and resources cannot be deleted or freed.The system must be configured so that system objects and resources cannot be deleted or freed.
- Application tasks must be designed to run forever.
- Because VxWorks 653 might not detect an invalid pointer that an application passes to it, when an application requests that data be stored, it must first check that memory pointers are not corrupted.
- Applications must not modify a task control block (TCB) directly, but must use the provided API only.
- Applications must use semaphore types and options that protect against priority inversion.
- Applications must use exclusion mechanisms that protect against deadlock and race conditions.
- All interrupt vectors must have handlers assigned to them.
- Handlers (interrupt, watchdog, and exception) must not call blocking routines.
- The target hardware must have enough CPU power to handle interrupts and to process the computing load.

2

Developing APEX Applications (ARINC 653 API)

Developing APEX Applications Introduction

APEX is an API between an application program and an operating system that supports the ARINC 653 specification. For VxWorks 653, the operating system is the VxWorks Cert partition OS.

The major enhancement APEX brings to a VxWorks Cert partition is in time and process management and the ability to manage periodic and aperiodic processes and their associated deadlines.

This chapter discusses programming concepts for writing APEX applications that run in VxWorks Cert partitions in a VxWorks 653 module. It explains the Wind River implementation of APEX. It does not discuss what is included in the ARINC 653 specification. If you need that level of detail, read the specification before you read this chapter.

APEX Services

APEX support provides services to do the following:

- Manage partitions
- Manage processes
- Manage time
- Communicate with other partitions (using messages, ports, and channels)
- Communicate within partitions (using buffers, blackboards, semaphores, and events)
- Monitor health

Some of these services (such as communicating within and outside partitions) can instead be handled using VxWorks Cert objects (such as pipes, message queues, and semaphores). However, such an implementation does not comply with the ARINC 653 specification.

APEX Configuration for VxWorks Cert Partitions

To provide a VxWorks Cert partition with APEX services, use the **INCLUDE_APEX** component in the kernel.

Note the following:

- APEX services are included in the **PROFILE_653** and **PROFILE_DEVELOPMENT** profiles (including the corresponding IBLL profiles).

For information on how to include components, refer to the *VxWorks 653 Configuration and Build Guide*.

Terminology and Concepts: APEX Versus VxWorks Cert

Some of the terminology in this document is specific to the ARINC 653 specification and APEX.

The following table lists some ARINC 653 terms and concepts and their VxWorks Cert equivalents.

Table 1 **Terminology and Concepts: APEX Versus VxWorks Cert**

Term or Concept	APEX	VxWorks Cert
Service or routine	Service	Routine
	ALLCAPS	**mixedCase( )**
	ACTION_OBJECT (for example **CREATE_PROCESS**)	**objectAction( )** (for example **taskDelete( )**)

Term or Concept	APEX	VxWorks Cert
	Services are issued or requested	Routines are called
Schedulable unit	Process	Task (APEX processes are implemented as VxWorks Cert tasks)
Scheduling method	FIFO	Priority-preemptive (FIFO) or round-robin
Priority numbering	Higher the value, higher the priority	Lower the value, higher the priority (0 is the highest priority)
Buffer	Buffer	N/A
Event	Event	VxWorks Cert event

Managing APEX Partitions

Managing a partition includes allocating partition memory and initializing the partition in accordance with the ARINC 653 specification.

Allocating Partition Memory

Each partition has predetermined areas of physical memory allocated to it. These unique memory spaces vary in size based on the requirements of the individual partitions. At most, one partition has write access to any particular area of memory. Memory partitioning is ensured by prohibiting write access outside a partition's defined memory areas. To ensure complete separation of applications, read access is also prohibited outside a partition.

Initializing Partitions: Cold and Warm Starts

Whereas the resource allocation necessary for each partition is specified in the XML-based configuration and build process (see the *VxWorks 653 Configuration and Build Guide*), the corresponding objects are defined when the partition is initialized. The MOS exclusively controls the allocation of resources to the partition by reserving specific memory. The partition uses this reserved memory to create the specified objects.

COLD_START

The cold-start partition operating mode is used when a partition is created and when the VxWorks 653 module starts from a powered-off state. During a cold start, partition objects are allocated and initialized.

WARM_START

The warm-start partition operating mode causes a partition to be re-initialized or restarted because of an error. During a warm start, persistent data is not re-initialized, and the partition code is not reloaded.

Partition Attributes

Partition attributes are defined in the XML configuration file at configuration and build time. For details, see the *VxWorks 653 Configuration and Build Guide*.

Getting Partition Status

The **GET_PARTITION_STATUS** service gets the partition status of the current condition (also called the start condition or the partition mode reason). The following table lists the possible values and their meanings.

The **apexPartitionModeReasonPtrGet( )** routine gets a pointer to the partition mode reason. This routine is not an APEX service.

Table 2 **APEX Partition Status Values**

Partition Status (Start Condition, Partition Mode Reason)	Reason for Current Partition Mode
HM_MODULE_RESTART	Recovery action taken at the VxWorks 653 module level
HM_PARTITION_RESTART	Recovery action taken at the partition level
NORMAL_START	Power-up
PARTITION_RESTART	Request for a **COLD_START** or **WARM_START** partition mode
POWER_ERROR_RESTART	A Wind River extension to the ARINC 653 specification

Setting the Partition Mode

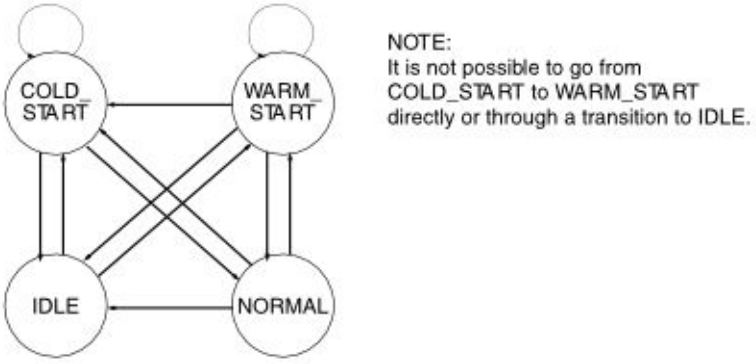
The **SET_PARTITION_MODE** service sets the operation mode for the current partition. The following table lists the available modes and their resulting actions.

Table 3 **APEX Partition Modes**

Partition Mode	Resulting Action
COLD_START	The partition restarts using the cold start initialization sequence.
IDLE	The partition shuts down. The partition is not initialized (that is, none of the ports associated with the partition are initialized), no processes are running, but the time windows allocated to the partition are unchanged.
NORMAL	The activate process is scheduled.
WARM_START	The partition restarts using the warm start sequence.

The following figure shows the allowable transitions in a partition's operating mode.

Figure 1: APEX Partition Mode Transitions



Controlling Preemption in Partitions

A process can issue the **LOCK_PREEMPTION** service (defined in the **apexProcess** library) to lock preemption in the partition. The service increments the lock level of the partition and disables processes from being rescheduled in the partition. This ability is important when processes are accessing critical sections or resources that are shared by multiple processes in the same partition. These critical sections may be specific areas of memory, certain physical devices, or the normal calculations and activity of a particular process.

The ability to intervene with normal rescheduling operations does not imply that the application is directly controlling VxWorks Cert. Since VxWorks Cert provides this service and knows all resulting actions and effects beforehand, the integrity of VxWorks Cert is not affected.

In addition, the **LOCK_PREEMPTION** service does not affect the scheduling of other partitions: if a process within a critical section is interrupted when the partition window ends, that process is the first to run when the partition runs again.

A process can issue the **UNLOCK_PREEMPTION** service (defined in the **apexProcess** library) to unlock preemption in the partition. The service decrements the lock level of the partition. Rescheduling of processes resumes only when the lock level is zero.

The **partitionCurrentLockLevelPtrGet( )** routine gets a pointer to the partition's current lock level. This routine is not an APEX service.

NOTE: Preemption locking does not prevent the error handler process from running.

Because preemption locking is partition wide, it is our recommendation not to issue blocking calls while a process is in preemption locked state. Issuing a blocking call in preemption locked state will result in potential scheduling of another process; this other process will inherit preemption locking and therefore will not itself be preemptible which may lead to unexpected or undesirable system behavior.

The following is a list of potentially blocking APIs:

- **__assert**
- **__sfvwrite**
- **__smakebuf**
- **__sread**
- **__srefill**

- **__srget**
- **__stderr**
- **__stdin**
- **__stdout**
- **__swbuf**
- **__swsetup**
- **_Lockfilelock**
- **_Locksyslock**
- **_Mtxdst**
- **_Mtxinit**
- **_Mtxlock**
- **_Once**
- **abort**
- **clearlocks**
- **close**
- **CREATE_BLACKBOARD**
- **CREATE_BUFFER**
- **CREATE_ERROR_HANDLER**
- **CREATE_EVENT**
- **CREATE_PROCESS**
- **CREATE_QUEUING_PORT**
- **CREATE_SAMPLING_PORT**
- **CREATE_SEMAPHORE**
- **DELAYED_START**
- **delete**
- **fclose**
- **fdopen**
- **fflush**
- **fgetc**
- **fgetpos**
- **fgets**
- **fopen**
- **fprintf**
- **fputc**
- **fputs**
- **fread**

- **freopen**
- **fscanf**
- **fseek**
- **fsetpos**
- **ftell**
- **fwrite**
- **GET_ERROR_STATUS**
- **getc**
- **getchar**
- **gets**
- **getw**
- **hmEventInject**
- **hmEventLog**
- **initlocks**
- **ioctl**
- **iosRelinquish**
- **longjmp**
- **new**
- **open**
- **perror**
- **putc**
- **putchar**
- **puts**
- **putw**
- **read**
- **RECEIVE_QUEUING_MESSAGE**
- **RECEIVE_SAP_MESSAGE**
- **remove**
- **scanf**
- **SET_PARTITION_MODE**
- **START**
- **stdioFp**
- **stdioFpCreate**
- **STOP**
- **STOP_SELF**
- **SUSPEND**

- **taskDelay**
- **taskSafe**
- **taskSafeWait**
- **taskSuspend**
- **taskUnlock**
- **ungetc**
- **unlink**
- **vfprintf**

Managing APEX Processes

APEX processes are programming units contained within an APEX partition. Each process runs concurrently with other processes in the same partition.

A process consists of the following:

- the executable program
- data and stack areas
- program counter
- stack pointer
- priority deadline

Creating Processes

The **CREATE_PROCESS** service is used to create a process with certain attributes and allocate resources for it. Since the service can be called only during warm or cold start of a partition, creation attributes cannot be changed after a partition is initialized. Each process is created only once during the life of the partition. Also, all the processes in a partition must be defined in such a way that the necessary memory resources for each process can be determined at system build time. For information on configuring memory, see the *VxWorks 653 Configuration and Build Guide*.

For information on getting creation attributes dynamically, see Getting the Current Status of Processes below.

The following names are field names in a **PROCESS_ATTRIBUTE_TYPE** structure, which is an argument to the **CREATE_PROCESS** service.

BASE_PRIORITY

Process initial priority

DEADLINE

Type of deadline (**SOFT**, **HARD**, or no deadline). This indicates the correct remedial action to the health monitor.

ENTRY_POINT

Starting address of the process

NAME

String identifier for the process. It must be unique within the partition.

PERIOD

Delay between two activations (for periodic processes only)

STACK_SIZE

Size (in bytes) of the stack allocated to the process

TIME_CAPACITY

The elapsed time within which the process should complete running

Changing the Current Priority of Processes

Although the initial priority is set when the process is created (**BASE_PRIORITY**), the current priority can by changed dynamically through the **SET_PRIORITY** service.

For information on getting the current priority, see Getting the Current Status of Processes below.

Increasing Deadline Times

Deadline time is the absolute time by which the process should be complete. It starts as the return value of the **GET_TIME** service (current system time) plus the **TIME_CAPACITY** that is specified when the process is created. VxWorks Cert periodically evaluates deadline time to determine whether the process is satisfactorily completing its processing within the allotted time (time capacity). Deadline time can be increased by issuing the **REPLENISH** service (defined in the **apexTime** library).

For information on getting the value of deadline time, see Getting the Current Status of Processes below.

Getting the Current Status of Processes

The **GET_PROCESS_STATUS** service gets the current status of a process. The return value is of **PROCESS_STATUS_TYPE** type, which contains the fields listed in the following table.

Table 4 **APEX Process Status Information**

Field in PROCESS_STATUS_TYPE	Description
ATTRIBUTES	The creation attributes for the process See Creating Processes
CURRENT_PRIORITY	Current priority of the process See Changing the Current Priority of Processes
DEADLINE_TIME	Current deadline time for the process See Increasing Deadline Times
PROCESS_STATE	Current state of the process See Process State Transitions

Getting Process IDs

The **GET_MY_ID** service gets the process ID of the calling process.

The **GET_PROCESS_ID** service gets the process ID of the process with the specified name.

Getting and Using VxWorks Cert Task Information

Because APEX processes are implemented as VxWorks Cert tasks, they have VxWorks Cert task IDs. The following routines are available:

The **taskIdFromProcIdGet( )** routine gets the VxWorks Cert task ID for the specified APEX process ID.

The **procIdFromTaskIdGet( )** routine gets the APEX process ID for the specified VxWorks Cert task ID.

Types of Processes

There are two types of processes:

Periodic Processes

A periodic process is a process that is activated at regular times (defined by the **PERIOD** creation attribute). At activation time, the process becomes eligible for scheduling. When that happens, the state of the process changes to **RUNNING** or **READY** if it is preempted by a higher-priority process (periodic or not).

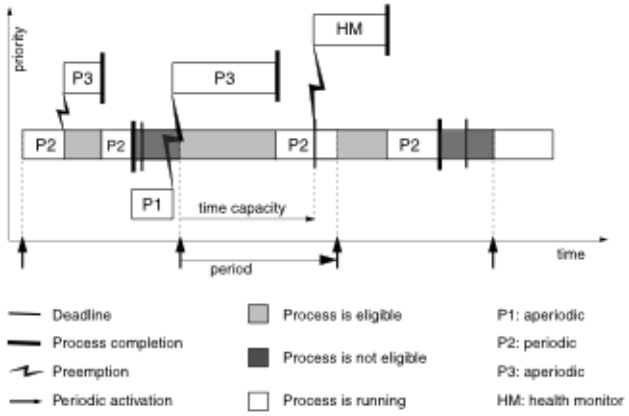
Aperiodic Processes

An aperiodic process is the same as a periodic process, but without an activation time. That is, an aperiodic process has a **PERIOD** creation attribute equal to **INFINITE_TIME_VALUE**.

Scheduling Processes

The following figure illustrates an example of process scheduling. Processes are scheduled according to the POSIX **SCHED_FIFO** method. In the example, P2 does not complete during the second time period. It is preempted by P3 most of the time, so it misses its deadline.

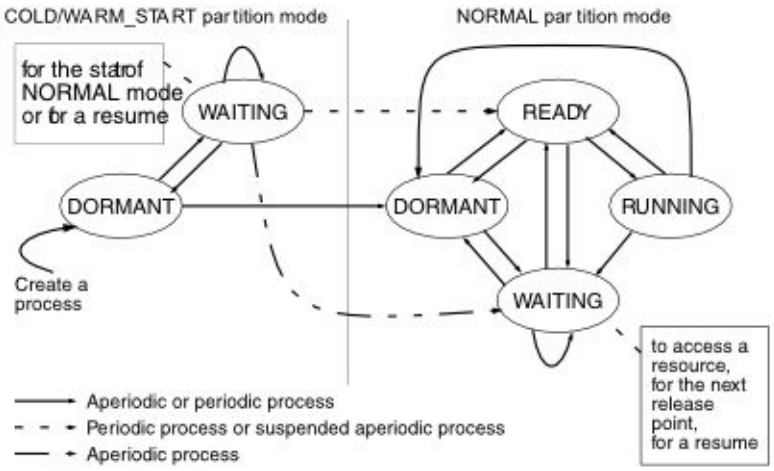
Figure 2: Example of Processes Scheduled with FIFO Scheduling



Process State Transitions

The following figure shows the relationships between the various states for APEX processes.

Figure 3: APEX Process State Transitions



DORMANT State

The **DORMANT** state indicates that the process is ineligible to receive resources. A process is in the **DORMANT** state before it is started and after it is terminated (that is, after a **STOP** service has been issued). Processes are created in the **DORMANT** state.

The **DORMANT** state moves to the following states:

READY

- When a process is started by another process while in **NORMAL** mode.

WAITING

- When a process is started during **INIT** mode.
- When a periodic process is started during **NORMAL** mode and is waiting for its next release point.
- When an aperiodic process is started with a delay during **NORMAL** mode.

DORMANT

- When a process is not started before the transition to **NORMAL** mode.

WAITING State

The **WAITING** state indicates that the process is not allowed to receive resources until a particular event occurs. The process is waiting for one or both of the following reasons:

- It is waiting on a resource, such as a semaphore or an event.
- It is suspended.

The **WAITING** state moves to the following states:

DORMANT

- When another process stops the process.
- When an error occurs and the health monitor stops the process.

READY

- When an unavailable resource becomes available and the process is not suspended.
- When the **RESUME** service is requested for the process and the process is not waiting for any resources.
- When the **TIMED_WAIT** service was requested, and the delay has expired.
- For a periodic process, when the time to activate the process (deadline time) is reached.
- For an aperiodic process started during **INIT** mode (which is started in the **WAITING** state), when the partition enters **NORMAL** mode.

WAITING

- When a process that is waiting to access a resource (delay, semaphore, period, event, message, and so on) is suspended.
- When a process that is both waiting to access a resource and suspended is resumed, or the resource becomes available, or the timeout expires.
- For a periodic process started during **INIT** mode (which is started in the **WAITING** state), when the VxWorks 653 module enters **NORMAL** mode.
- For an aperiodic process started using the **DELAYED_START** service in the **INIT** mode, when the partition enters **NORMAL** mode.

RUNNING State

The **RUNNING** state indicates that the process is running. Only one process can be running at a time. The previous state of a running process is always **READY**.

The **RUNNING** state moves to the following states:

DORMANT

- When the running process stops itself.
- When the health monitor stops the process because an error occurred.

READY

- When the running process requests a service delay with a delay time of zero. This is equivalent to round-robin scheduling of processes of the same priority.
- When another process with a higher priority enters the **READY** state.

WAITING

- When the running process suspends itself.
- When the running process attempts to access an unavailable resource (semaphore, buffer, event, blackboard, queuing port) with a non-zero timeout.
- When the running process requests a delay service (such as a timed wait or periodic wait) with a non-zero delay.

READY State

The **READY** state indicates that the process is eligible to be scheduled and is waiting to run. The process is not running for either or both of these reasons:

- A higher-priority process is running.
- Preemption is locked.

The **READY** state moves to the following states:

DORMANT

- When another process issues a **STOP** service on the process.
- When the health monitor stops the process because an error occurred.

RUNNING

- When the scheduler selects the process to run. This is the only way to enter the **RUNNING** state.

WAITING

- When another process issues a **SUSPEND** service on the running process.

Suspending and Resuming Processes

When a process is suspended, the process is not allowed to run, and its state is **WAITING** until another process resumes it. When a process waits on a resource such as a semaphore or an event, it can also be suspended. The services to suspend or resume processes are:

- **SUSPEND_SELF**

  If the current process is an aperiodic process, the service suspends it until **RESUME** is issued or the specified timeout expires.

  If preemption is disabled and the process being suspended is the one holding the preemption lock, **SUSPEND_SELF** returns **INVALID_MODE**.

- **SUSPEND**

  The service lets the current process suspend any aperiodic process, except itself, until another process resumes the suspended process. If the process is pending in a queue at the time it is suspended, it stays in the queue. When it is resumed, it continues pending unless it was removed from the queue before the end of its suspension. The process might have been removed if a particular condition occurred, a timeout expired, or the queue was reset.

  A process may suspend any other process asynchronously.

  If process B suspends an already suspended or self-suspended process A, the service has no effect.

  If preemption is disabled and the process being suspended is the one holding the preemption lock, **SUSPEND** returns **INVALID_MODE**.

- **RESUME**

  The service lets the current process resume another previously suspended process. The resumed process becomes ready if it is not waiting on a resource such as a delay, semaphore, period, event, or message. Since a periodic process cannot be suspended, it cannot be resumed.

Stopping and Starting Processes

The **STOP** service makes a process ineligible for processor resources (in other words, its state is **DORMANT**) until another process issues the **START** service, causing the first process to enter the **READY** state. After it is created, a process is also in a **DORMANT** state. The **DELAYED_START** service applies only to periodic processes and lets process schedules be phased. The services to stop and start processes are:

- **STOP_SELF**

The service lets the current process stop itself. If the current process is not the error handler process, the partition is placed in the unlocked condition. The service should not be called when the partition is in **WARM_START** or **COLD_START** modes. If it is, the behavior is not defined.

- **STOP**

  The service makes a process ineligible for processor resources until another process issues **START**. The **STOP** service lets the current process abort any process except itself. When a process aborts another process that is pending in a queue, the aborted process is removed from the queue.

- **START**

  The service initializes all attributes of a process to their default values and resets the process's run-time stack. If the partition is in **NORMAL** mode, the process's deadline expiration time and next release point are calculated. The service lets the current process start another process.

- **DELAYED_START**

  The service initializes all attributes of a process to their default values, resets the processes's run-time stack, and places the process in the **WAITING** state (that is, the specified process goes from **DORMANT** to **WAITING**). If the partition is in **NORMAL** mode, the process's release point is calculated with the specified delay time. In addition, the process's deadline expiration time is calculated. The service lets the current process start another process.

Controlling Preemption

Preemption locking or unlocking disables or enables process rescheduling in a partition. For details, see *Controlling Preemption in Partitions* in Managing APEX Partitions on page 7.

Managing Time in APEX Partitions

Time partitioning is a major characteristic of VxWorks 653 and all ARINC 653 systems.

System Clock Time

The system clock time gives the unique time of the system. Time is unique and independent of partition execution within a VxWorks 653 module. All time values or capacities are related to this unique time and are not relative to any partition execution.The timer provides the time of day, and it is used as a stamp or for anything that needs time or date information.

The **GET_TIME** service gets the system clock time.

Requesting Resources and Timeouts

When a process requests an APEX resource (such as a semaphore or an event), it can specify a timeout of one of the following types:

INFINITE_TIME_VALUE

Never expire. It is equivalent to wait forever.

ZERO_TIME_VALUE

Do not wait for the resource. If the resource is not available, return an error.

Finite value of timeout

The maximum amount of time to wait for a resource.

The timeout's units are of **SYSTEM_TIME_TYPE** type, which defines time in nanoseconds. Although expressed in nanoseconds, the time is rounded up to ticks, and the actual timeout is a multiple of the system clock period.

A timeout usually results in an early return from the service and a **TIMED_OUT** return code.

If a timeout expires outside the partition window, it is acted on at the beginning of the next partition window (as with deadlines; see Deadlines below).

Scheduling Processes

APEX time management services let partitions control their processes.

At the end of each processing cycle, a periodic process requests the **PERIODIC_WAIT** service to get a new deadline. The new deadline is calculated from the next periodic release point for that process. For all processes, the **TIMED_WAIT** service lets the process suspend itself for a minimum amount of elapsed time. After the wait time has elapsed, the process becomes available to be scheduled.

The **REPLENISH** service lets a process postpone its current deadline by the amount of time that has already passed.

Each process in a partition can specify an amount of elapsed time (called the time capacity) it is allowed to consume in order to satisfy its processing requirement. This time capacity is used to set a processing deadline time that VxWorks Cert periodically evaluates to determine whether the process is satisfactorily completing its processing within the allotted time.

Deadlines

Each process has associated with it a fixed time capacity, which represents the response time allotted to it for satisfying its processing requirements.

The deadline time (a variable process attribute) determines whether the process is satisfactorily completing its processing within its time capacity. Deadline time can be increased by issuing the **REPLENISH** service. The **PERIODIC_WAIT** service cancels the current deadline, and a new deadline is created at the next activation.

A deadline can expire outside the partition window, but it is acted on at the beginning of the next partition window (as with timeouts).

There are three types of deadlines:

- Hard deadlines

  If a process fails to meet a hard deadline within the specified time period, VxWorks Cert takes remedial action by raising a **DEADLINE_MISSED** event.

- Soft deadlines

  If a process fails to meet a soft deadline within the specified time period, typically, the failure is recorded and processing continues.

  Here also, a **DEADLINE_MISSED** event is raised. The event text allows you to distinguish whether the event is for a hard or soft deadline.

- No deadline

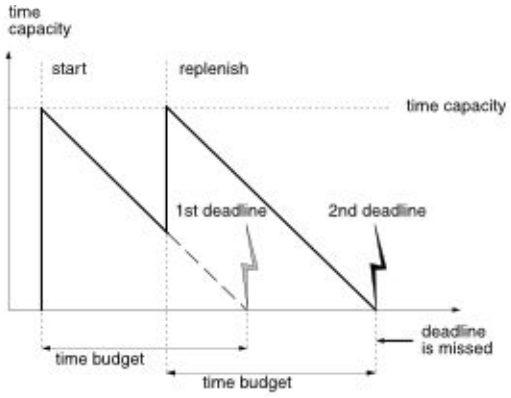
  No action is taken if a process fails to complete processing within the specified time period.

For a periodic process, the countdown on deadline time starts when the process's active period starts. Countdown is disabled when the process requests the **PERIODIC_WAIT** service. Deadline time is ended when the process is stopped or when it calls the **REPLENISH** or **PERIODIC_WAIT** services. Countdown is deactivated when the partition is in an operating mode other than **NORMAL**.

For an aperiodic process, the deadline time countdown starts when the process starts and the partition mode is **NORMAL**. The deadline time is rearmed with additional time equal to the time budget specified when the process requests a **REPLENISH** service (see the following figure). The process can specify the additional time. A periodic process deadline cannot be postponed beyond its next release point.

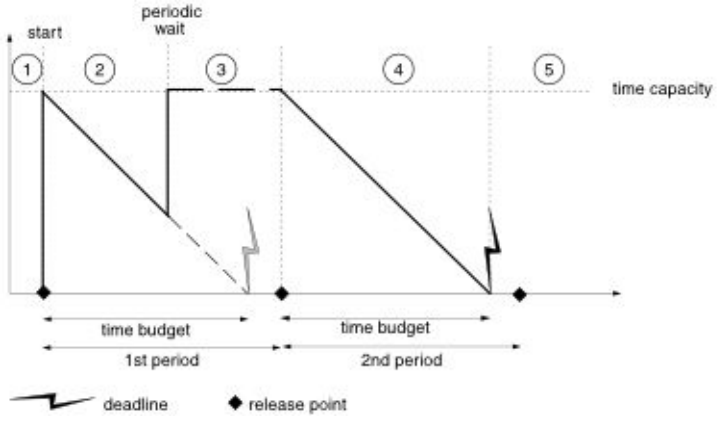
NOTE: To make the following figures easier to read, the time capacity (the time initially allotted to the process to complete its work) and the amount of time added by **REPLENISH** (specified by the **BUDGET_TIME** parameter) appear to be equal and constant. In reality, **BUDGET_TIME** can be larger or smaller than the time capacity.

Figure 4: Process with Replenish (Periodic or Aperiodic Processes)



The deadline ends when the process is stopped or when the partition state is not **NORMAL**.

Figure 5: Periodic Process Examples with PERIODIC_WAIT and Deadline

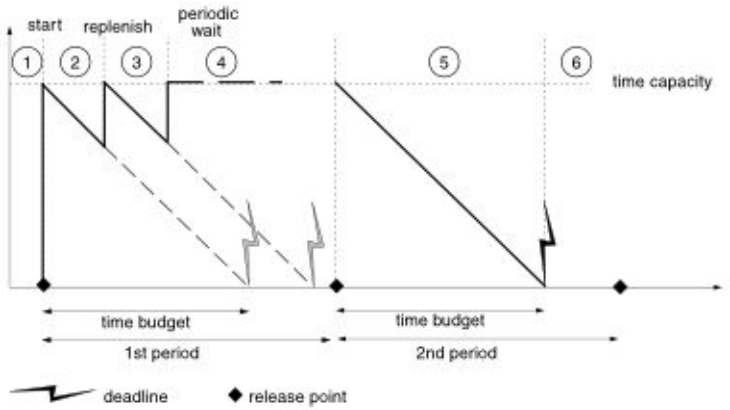


In the previous figure, the periodic process is in the following states:

1. **DORMANT**
2. **RUNNING** (or **READY** if another process has preempted it)

3. **WAITING**
4. **RUNNING** (or **READY** if another process has preempted it)
5. **READY** until the health monitor takes an action. The process has not completed within the deadline interval.

Figure 6: Periodic Process with REPLENISH and PERIODIC_WAIT



In the previous figure the periodic process is in the following states:

1. **DORMANT**
2. **RUNNING** (or **READY** if another process has preempted it)
3. **RUNNING** (or **READY** if another process has preempted it)
4. **WAITING**
5. **RUNNING** (or **READY** if another process has preempted it)
6. **READY** until the health monitor takes an action. The process has not completed within the deadline interval.

Release Points

The first release point for a periodic process is relative to the start of its partition's first window in the major time frame. Using a **DELAY** in the **DELAYED_START** service enables phasing of the process schedule. Subsequent release points are based on the previous release point and the process period.

Communicating Between Partitions

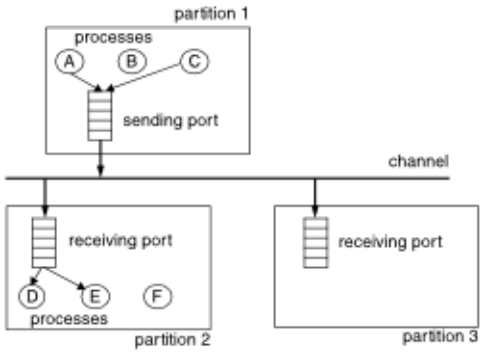
Interpartition communication includes all communication between two or more partitions in a VxWorks 653 module.

APEX partitions within a VxWorks 653 module communicate with each other by messages, ports, and channels.

A message can be sent from one source port to one or more destination ports. Processes read from these destination ports.

The configuration of this messaging system is defined when the VxWorks 653 module is configured. For more information, see the *VxWorks 653 Configuration and Build Guide*.

Figure 7: Sending Messages between Partitions



Limitations of APEX for Communicating between Partitions

Although the ARINC 653 standard specifies the following, APEX does not support them:

- multicast and client-server messages
- acknowledgement of messages

APEX Messages

APEX messages are contiguous blocks of data.

Although the ARINC 653 standard lets messages be decomposed into small blocks, communicated individually, and re-assembled before delivery, VxWorks 653 transmits full messages only. This avoids checking for message completeness and retransmitting segments.

Messages can be of fixed or variable lengths. Fixed length means a fixed size for every occurrence of a particular message. A variable-length message can vary in size. The sender specifies the length when the message is sent. To accommodate various message lengths, the messaging system allocates resources for the maximum length, which is defined in the ARINC 653 standard.

A message can be sent periodically or on demand (aperiodically). The messaging system operates independently of the content of the messages it transmits.

Any given message can be sent from a single partition only. In addition, once delivered, messages are destroyed. That is, it is not possible to request old versions of messages.

APEX Channels

A channel defines the following:

- logical link between one source port and one or more destination ports
- mode of transfer of the messages from the source to the destination
- characteristics of the messages to be sent

A message sent to one destination port is called a directed message. A message sent to multiple destination ports is called a broadcast message.

Each channel can be configured to operate in a specific mode. Two modes of transfer are defined: sampling mode and queuing mode. The messaging service returns an error if the port configuration (mode, direction) is not compatible with the request.

For information on how to configure ports and their associated channels, see the *VxWorks 653 Configuration and Build Guide*.

The consistency of the configuration is checked at build time and startup time. For a channel, the size of the sending port cannot exceed the size of any of the receiving ports.

Sampling Mode

In sampling mode, messages typically carry similar, but updated, data. No queuing is performed. A message remains in the source port until it is sent or overwritten. Messages arrive at the destination port or ports in the order in which they were sent. Each new message overwrites the previous one when it reaches the destination port and remains there until it is overwritten itself. Sampling mode supports variable-length messages.

Refresh Rate

Refresh rates applies to destination ports in sampling mode. The refresh rate indicates the maximum acceptable age of a valid message, from the time it was received at the port. It is specified when the port is created. When the message is read, a validity output parameter indicates whether the age of the message is consistent with the port's refresh rate.

Queuing Mode

In queuing mode, each new instance of a message may contain uniquely different data. Therefore, overwriting previous messages is not allowed during the transfer. Messages are queued in the source port until they are sent, and no message is lost (except in the case of a full message queue with the **DISCARD** protocol). Messages are stored in the receiver port until a process reads them. For information about the protocols required to manage message queues, see *Port Protocols* below. Queuing mode supports variable-length messages.

Ports

APEX ports let applications communicate with each other within a VxWorks 653 module. Safe IPC is used as the transport mechanism. For more information on Safe IPC, see the safeipc reference entries.

Port Protocols

The ARINC 653 standard does not specify port protocols, but VxWorks 653 provides the following:

BLOCK

A queuing message is sent to all the channel's destination ports. If any one is full, the message is queued in the source port in FIFO order.

When the source port is full and if a timeout was specified, sender processes are blocked during the **SEND_QUEUING_MESSAGE** service.

When a destination port is emptied, retransmission is attempted. Whether it succeeds depends on the state of the channel's other destination ports.

The main advantage of using the **BLOCK** protocol is that no messages are lost, as the ARINC 653 specification requires. The main drawback is that it introduces coupling between partitions. A non-responsive receiving partition blocks the entire channel, affecting the normal behavior of other receiving partitions.

Because the **BLOCK** attribute that is set on a single destination port can block the entire channel, it is considered a channel-wide attribute and is attached to the channel's source port.

DISCARD

If one of the channel's destination ports is full, the source port discards the message for that port. Therefore, if all the destination ports are full, the message might be lost. When a message is so discarded, the port's overflow flag is set to notify the application of the discarded (lost) message.

The **DISCARD** protocol avoids the problem caused by a faulty application that fails to read or empty its destination ports, thereby preventing other partitions from receiving messages.

Working with Queuing Messages

Create Queuing Ports

The **CREATE_QUEUING_PORT** service creates an empty port in queuing mode and returns a port ID. The **QUEUING_DISCIPLINE** attribute indicates whether blocked processes are queued in FIFO or priority order.

Sending Queuing Messages

The **SEND_QUEUING_MESSAGE** service sends a message to the specified queuing port. If there is sufficient space at the queuing port, the message is added to the end of the port's queue. If there is insufficient space, the sending process is blocked and added to the sending port's queue, according to the port's queuing discipline. The process stays on the queue until the specified timeout expires (if it is finite) or until space for the message becomes free at the queuing port.

Receiving Queuing Messages

The **RECEIVE_QUEUING_MESSAGE** service receives a message from the specified queuing port. If the queuing port is not empty, the message at the head of the port's queue is removed and returned to the caller. If the queuing port is empty, the process is blocked until the specified timeout or until a message arrives.

Getting Queuing Port Information

The **GET_QUEUING_PORT_ID** service gets the port ID for a specified queuing port name.

The **GET_QUEUING_PORT_STATUS** service gets the following information for a queuing port:

- direction
- number of messages at the port
- number of waiting processes
- size

Working with Sampling Messages

Creating Sampling Queues

The **CREATE_SAMPLING_PORT** service creates an empty sampling port and returns a port ID.

Writing Sampling Messages

The **WRITE_SAMPLING_MESSAGE** service writes a message at the specified sampling port, overwriting a previous message.

Reading Sampling Messages

The **READ_SAMPLING_MESSAGE** service reads a message at the specified sampling port and returns a validity parameter that indicates whether the age of the message is consistent with the port's refresh rate. The age is the difference between the value of the system clock when the message is written into the port and the value of the system clock when the messages is read at the destination port.

Getting Sampling Port Information

The **GET_SAMPLING_PORT_ID** service gets the port ID for a specified sampling port name.

The **GET_SAMPLING_PORT_STATUS** service gets the following for a sampling port:

- direction
- refresh rate
- size
- validity of the last message read by the specified sampling port

Communicating within APEX Partitions

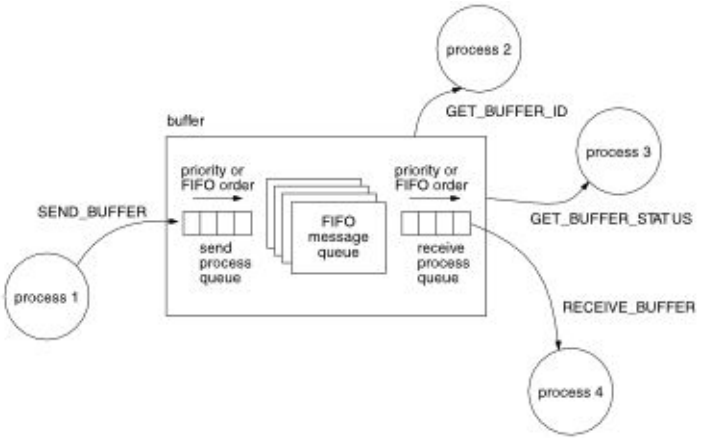
APEX provides the following APEX objects so that processes can communicate with each other within a partition: buffers, blackboards, semaphores, and events.

Communicating Using APEX Buffers

APEX buffers let processes communicate with each other within a partition. Buffers support a single message type between multiple source and destination processes. Communication is indirect: participating processes address the buffer rather than the opposing processes directly, thus providing a level of process independence.

Buffers store multiple messages in message queues and no messages are lost. The following figure summarizes how processes use a buffer to communicate.

Figure 8: Processes Using a Buffer to Communicate



Creating APEX Buffers

APEX buffers can be created only when a partition is being initialized, that is, when the partition mode is anything but **NORMAL**. Processes can create as many buffers as are supported by the memory that is pre-allocated for the partition's buffers.

The **CREATE_BUFFER** service creates an empty buffer with the following specified information:

- name, which must be unique within the partition
- maximum number of messages
- maximum message size
- discipline for queuing waiting processes (**FIFO** or **PRIORITY**)

The service returns a buffer ID.

Sending Messages to APEX Buffers

The **SEND_BUFFER** service sends a message to a specified buffer.

If the buffer is empty, the message is stored in FIFO order. If processes are waiting for messages, the first process is removed from the queue and put in the **READY** state.

If the buffer is full, the sending process is put in the **WAITING** state and put in the buffer's send queue according to the buffer's queuing discipline and the specified timeout value.

If the service times out, it returns **TIMED_OUT**.

Receiving Messages from APEX Buffers

The **RECEIVE_BUFFER** receives a message from a specified buffer.

If the buffer is not empty, the message is removed from the FIFO queue.

If the buffer is full and processes are waiting for messages, the first process is removed from the send queue and put in the **READY** state.

If the buffer is empty, the receiving process is put in the **WAITING** state and put in the receive queue according to the buffer's queuing discipline and the specified timeout value.

If the service times out, it returns **TIMED_OUT**.

Getting APEX Buffer Information

The **GET_BUFFER_ID** service gets the buffer ID of a specified buffer.

The **GET_BUFFER_STATUS** service gets the following information for the specified buffer:

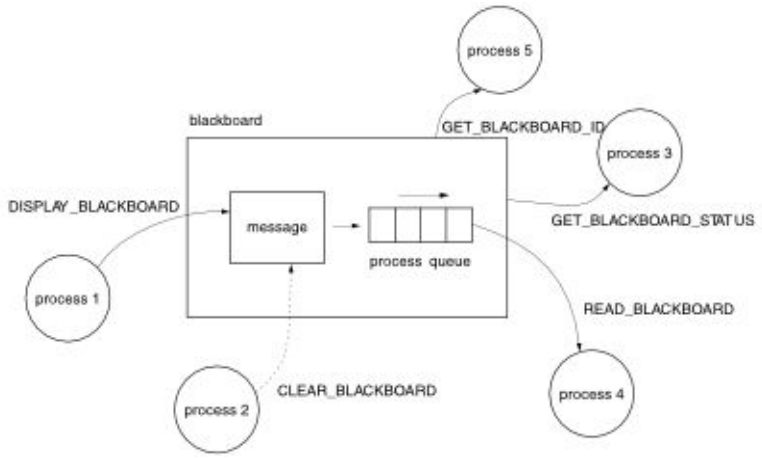
- current number of messages
- number of waiting processes
- maximum allowable number of messages
- maximum allowable message size

Communicating Using APEX Blackboards

APEX blackboards let processes communicate with each other within a partition. Blackboards support a single message type between multiple source and destination processes. Communication is indirect: participating processes address the blackboard rather than the opposing processes directly, thus providing a level of process independence.

The following figure summarizes how processes use a blackboard to communicate.

Figure 9: Processes Using a Blackboard to Communicate



Creating Blackboards

Blackboards can be created only when a partition is being initialized, that is, when the partition mode is anything but **NORMAL**. Processes can create as many blackboards as are supported by the memory that is pre-allocated for the partition's blackboards.

The **CREATE_BLACKBOARD** service creates an empty blackboard with the following specified information:

- name, which must be unique within the partition
- maximum number of messages

The service returns a blackboard ID.

Displaying Blackboard Messages

The **DISPLAY_BLACKBOARD** service writes a message on a blackboard and remove all waiting processes from the process queue, putting them in the **READY** state. The message remains on the blackboard.

Reading Blackboard Messages

If the specified blackboard is not empty, the **READ_BLACKBOARD** service reads the displayed message from it.

If the blackboard is empty, the reading process is put in the **WAITING** state according to the specified timeout value.

If the service times out, it returns **TIMED_OUT**.

Clearing Blackboards

The **CLEAR_BLACKBOARD** service clears the message from the specified blackboard. As a result, the blackboard becomes empty.

Getting Blackboard Information

The **GET_BLACKBOARD_ID** service gets the blackboard ID for the specified name.

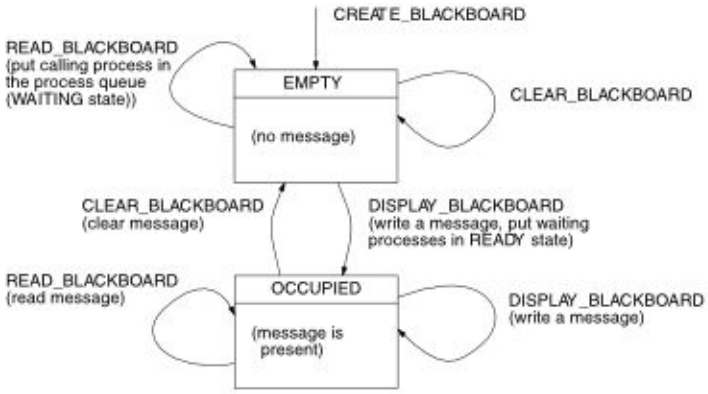
The **GET_BLACKBOARD_STATUS** service gets the following information for the specified blackboard:

- state (**EMPTY** or **OCCUPIED**)
- number of processes waiting for a message
- maximum allowable message size

State Transitions for Blackboards

The following figure shows state transitions for blackboards.

Figure 10: State Transitions for Blackboards



Communicating Using APEX Semaphores

APEX semaphores are counting semaphores. A process waits on a semaphore to gain access to a resource and then signals the semaphore when it is finished. A semaphore's current value indicates the number of times that it is currently available to be taken.

Creating APEX Semaphores

APEX semaphores can be created only when a partition is being initialized, that is, when the partition mode is anything but **NORMAL**. Processes can create as many APEX semaphores as are supported by the memory that is pre-allocated for the partition's APEX semaphores.

The **CREATE_SEMAPHORE** service creates a semaphore with the following specified information:

- name, which must be unique within the partition
- maximum value
- current value
- queuing discipline (**FIFO** or **PRIORITY**)

The service returns a semaphore ID.

Waiting for APEX Semaphores

If the specified semaphore's current value is not zero, the **WAIT_SEMAPHORE** service decrements the value, and the process continues to run.

If the current value is zero, the process is put in the **WAITING** state and queued according to the semaphore's queuing discipline and the specified timeout.

If the service times out, it returns **TIMED_OUT**.

Signalling APEX Semaphores

If there are no processes waiting for the specified semaphore, the **SIGNAL_SEMAPHORE** service increments the semaphore's current value.

If there are processes waiting for the semaphore, the service uses the semaphore's queuing discipline to determine which process will receive the signal and sets that process's state to **READY**.

Getting APEX Semaphore Information

The **GET_SEMAPHORE_ID** service gets the semaphore ID for the specified name.

The **GET_SEMAPHORE_STATUS** service gets the following information for the specified semaphore:

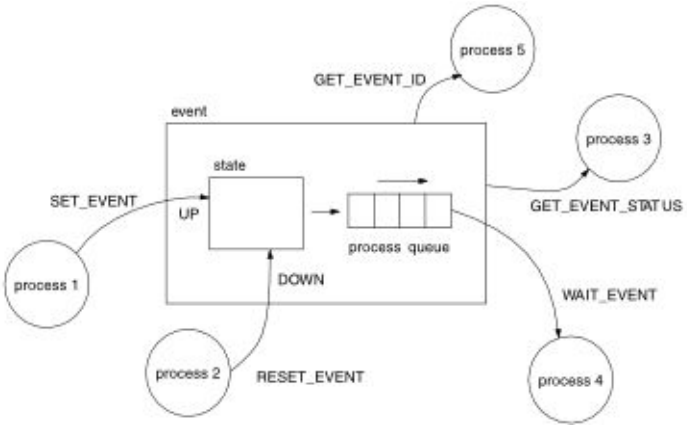
- current count
- number of processes waiting for the semaphore
- maximum value

Synchronizing Using APEX Events

APEX events let processes in a partition synchronize. Processes that are waiting for a condition are notified when the condition happens. An event can be in one of two states: **UP** or **DOWN**.

The following figure summarizes how processes use an APEX event to synchronize.

Figure 11: Synchronizing Using an APEX Event



NOTE: Processes should not count the occurrences of an event. Multiple **SET_EVENT** conditions that occur in a short period of time, or conditions that occur when no processes are waiting for the event, are coalesced into one **UP** state.

Event Queuing

Rescheduling of processes occurs when a process attempts to wait for an event that is down. The calling process is queued for a specified amount of time (the time can be infinite). If the event is not set (up) in that amount of time, VxWorks 653 automatically removes the process from the queue, sets the return code to **TIMED_OUT**, and puts the process back into the ready state.

Creating APEX Events

APEX events can be created only when a partition is being initialized, that is, when the partition mode is anything but **NORMAL**. Processes can create as many APEX events as are supported by the memory that is pre-allocated for the partition's APEX events.

The **CREATE_EVENT** service creates an event with a specified name, which must be unique within the partition. The service returns an event ID. The event starts in the **DOWN** state.

Setting and Resetting APEX Events

The **SET_EVENT** service sets the specified event to the **UP** state. All the processes waiting for the event are put into the **READY** state.

The **RESET_EVENT** service sets the specified event to the **DOWN** state.

Waiting for APEX Events

If the specified event is in the **DOWN** state, the **WAIT_EVENT** service moves the calling process from the **RUNNING** state to the **WAITING** state. If the event is in the **UP** state, the calling process continues to run.

Getting APEX Event Information

The **GET_EVENT_ID** service gets the event ID for the specified event.

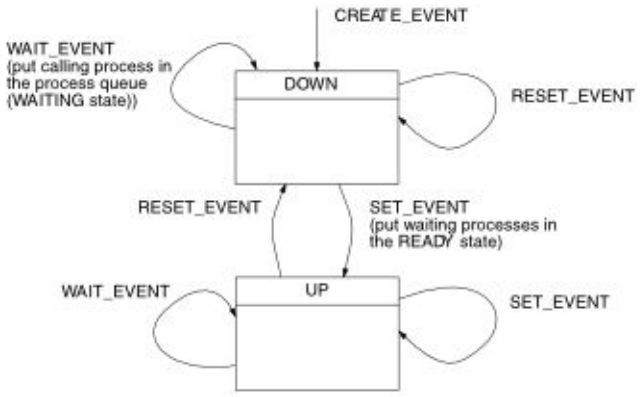
The **GET_EVENT_STATUS** service gets the following status information for the specified event:

- state (**UP** or **DOWN**)
- number of waiting processes

State Transitions for APEX Events

The following figure shows state transitions for APEX events.

Figure 12: State Transitions for APEX Events



Monitoring Health in APEX Partitions

APEX error services (in the **apexError** library) support process-level health monitoring as defined in the ARINC 653 standard.

When an APEX process raises an error, the partition's error handler process runs.

Raising Process-Level Errors

When an APEX partition detects an error, it issues the **RAISE_APPLICATION_ERROR** service with an error code and a fault message. The service causes the error handler process to run.

Depending on its nature and scope, an error raised at the process level with the **RAISE_APPLICATION_ERROR** service could propagate to the partition level, where it is processed.

APEX Errors

The following are process-level APEX error codes:

- **APPLICATION_ERROR**
- **DEADLINE_MISSED**
- **HARDWARE_FAULT**
- **ILLEGAL_REQUEST** (invalid or illegal OS call)
- **MEMORY_VIOLATION**
- **NUMERIC_ERROR**
- **POWER_FAIL**
- **STACK_OVERFLOW** (process stack overflow)

A faulty process can continue to run only in the cases of **APPLICATION_ERROR** or **DEADLINE_MISSED**.

Creating Error Handler Processes

An APEX application creates an error handler process for a partition by issuing the **CREATE_ERROR_HANDLER** service with the error handler entry point and stack size. The application supplies the error handler code.

The error handler is an aperiodic process that runs in the partition window with the highest priority of any process in the partition. It preempts any process regardless of its priority, even if preemption is disabled for the partition. It has no process ID and cannot be accessed by other processes within the partition. That is, other processes cannot suspend it, stop it, or change its priority.

An application developer writes the error handler. The error handled can do a selection of the following:

- Get information about the error (issue **GET_ERROR_STATUS**):

  - error code (see APEX Errors)
  - error address
  - process ID of the faulty process
  - fault message

    If more than one process is faulty, the error handler process must issue **GET_ERROR_STATUS** in a loop until there are no more processes in error (that is, until the service returns **NO_ACTION**).

- Get information about the failed process (issue **GET_PROCESS_STATUS**).
- Stop (issue **STOP**) or restart (issue **START**) the failed process.
- Restart the partition (issue **SET_PARTITION_MODE** with **WARM_START** or **COLD_START**).
- Shut down the partition (issue **SET_PARTITION_MODE** with **IDLE**).
- Escalate the error to the partition level (issue **RAISE_APPLICATION_ERROR**).
- Log the fault message with the health monitor (issue **REPORT_APPLICATION_MESSAGE**).
- Stop itself (issue **STOP_SELF**).

If code running in the context of the error handler calls **LOCK_PREEMPTION** or **UNLOCK_PREEMPTION**, no action is taken. This is because the error handler is already the highest-priority process and cannot be interrupted or blocked. It can transmit the error context to health monitoring via the **REPORT_APPLICATION_MESSAGE** service for maintenance purpose.

An error handler process cannot do the following:

- Correct an error. For example, it cannot limit a value in the case of overflow.
- Call blocking services.

Errors that occur while the error handler process runs are partition-level errors.

3

Developing C++ Applications

Developing C++ Applications Introduction

You can develop C++ applications that run in VxWorks Cert partitions. A subset of this lets safety-critical applications be certified to Level A of the RTCA DO-178B/C avionics software guidelines.

For information on the GNU C++ toolchain, see *Using the GNU Compiler Collection*.

NOTE: C++ is not supported in the MOS.

Adding C++ Support to VxWorks Cert Partitions

By including certain components, you can provide a VxWorks Cert partition with C++ support.

To provide a VxWorks Cert partition with C++ support, include **FOLDER_CPLUS** in the kernel. If you use **PROFILE_653**, it is added for you by the system.

For information on how to include components and folders, refer to the *VxWorks 653 Configuration and Build Guide*.

Supported Features

Within the VxWorks cert environment there are numerous C++ features that are supported.

The C++ features supported for the cert environment include:

- Classes, including constructors and destructors
  - Constructors that allocate memory can be invoked until memory allocation is disabled.
  - Global C++ destructors are not called by the Wind River C++ runtime.
  - Non-global destructors are invoked as they normally would be and each line of code written for the destructor is executed. However, any code that would normally free up memory (as it would in the non-cert case) is bypassed.
- Inheritance, including multiple inheritance
- Templates
  - You can write your own templates and provide any supporting functions that are required. However, support for the Standard Template Library (STL) is not included for kernel applications.
- Namespaces
  - For example, the namespace **std** (see **yvals.h**) can be used.
- Dynamic dispatch (polymorphism)
- Operator overloading
- Virtual functions (including pure virtual functions)
  - In kernel mode, if a pure virtual function is called and there is no available definition, **lastRites( )** is called.
- C++ style cast operators (but no dynamic casting)
- The **new** operator
  - It is available until memory allocation is disabled.
- The **delete** operator
  - In kernel mode, it triggers **lastRites( )**.
  - It will not free up memory, but destructors are still invoked as they normally would be.

The C++ cert subset runtime library is provided as relocatable ELF files that must be linked with the kernel.

The following compiler flags are required when compiling C++ in any cert environment:

- **-fno-builtin** (also required for C language development)
- **-fno-exceptions**
- **-fno-rtti**
- **-fcheck-new**

Unsupported Features

Within the VxWorks cert environment there are some C++ features that are not supported.

The following C++ features are not supported for the cert environment:

- Exception handling, including try, catch, and throw keywords and related functionality
  - Attempting to use exception handling will result in compilation errors.
- Runtime type identification (RTTI) including any related dynamic type casting and type ID
  - Attempting to use RTTI will result in compilation errors.
- Runtime library support, including Standard Template Library (STL) support

4

Health Monitoring

Health Monitoring Introduction

Health monitoring provides a framework to raise and handle events, which can be alarms or messages, in a system.

Alarms are injected to represent faults in the system, and handlers perform health recovery actions.

Adding HM Support to VxWorks Cert Partitions

By including certain components, you can provide a VxWorks Cert partition with health monitoring (HM) support.

To provide a VxWorks Cert partition with HM support, include the **INCLUDE_HM** component in the kernel. If you use **PROFILE_653**, it is added for you by the system.

For information on how to include components, refer to the *VxWorks 653 Configuration and Build Guide*.

Basic Health Monitor Concepts

There are basic concepts as regards the health monitor that developers should be aware of within the VxWorks cert environment.

This section describes the following basic health monitor concepts:

- events (alarms and messages)
- health monitor hierarchy
- alarm injection
- thresholds

Health Monitor Events

An event is the base unit of injection within the health monitor. An event can be an alarm or a message.

Health Monitor Alarms

An alarm is an event that is the software representation of a fault that needs attention. It could have a positive or negative effect. Examples include hardware-generated exceptions, error paths in the code, and crossed thresholds. Alarms have information associated with them.

For details, see Table 5 on page 40.

Health Monitor Messages

A message is an event that has a code of **HM_MSG**. If the message is sent from within a partition, the partition health monitor handles it. If it is sent from outside the partition, the module health monitor handles it. A default handler is provided that logs the message. However, a system integrator can replace the handler with another by means of the XML configuration file. Messages, like alarms, have information associated with them.

For details, see Table 5 on page 40.

Health Monitor Hierarchy

There are four levels of health monitoring:

- module health monitor
- multi-partition health monitor
- partition health monitor
- process health monitor

The partition health monitor and the module health monitor are driven by static tables that relate event codes to their appropriate handlers. There is one module health monitor table for the VxWorks 653 module. There is a partition health monitor table for each partition. The tables are loaded as part of the configuration loading for the VxWorks 653 module and partitions.

Messages have a hard-coded dispatch level. That is, the system health monitor table cannot be used to configure their dispatch level. At initialization, the hard-coded dispatch rules override anything in the XML configuration file that pertains to health monitor messages.

There is the potential for a process health monitor (also called the error handler process) for each partition. The application must create the process health monitor by calling **hmErrorHandlerCreate( )** or, for ARINC 653 applications, by issuing **CREATE_ERROR_HANDLER**. The routines create a highest-priority task in the partition OS with which to run the process health monitor handler.

After an event is injected, the system health monitor table determines how to dispatch the event. Because dispatching is hard-coded, events are automatically dispatched to the process, partition, or module level, depending on where they were injected.

The system health monitor table relates the event code and system status at injection time to a dispatch level, which can be one of the following:

- no level
- process level
- partition level
- module level

The system integrator creates the system health monitor table in the XML configuration file for the VxWorks 653 module. For information on specifying the table, see the *VxWorks 653 Configuration and Build Guide*. The table is read as part of the configuration tables of the MOS.

In terms of memory, the process health monitor (error handler process) is within the partition OS.

The partition health monitor runs as a MOS task (its stack is in the MOS kernel domain) with a higher priority than the MOS task that is running the associated partition OS (also higher than any worker tasks). But, it is scheduled during its associated partition's window.

The module health monitor runs as a highest-priority task in the MOS and is the only task at this priority. Its priority is higher than the partition health monitor tasks.

Event Structure (HM_EVENT)

The following table describes the fields of the structure (**HM_EVENT**) that defines a health monitor event.

Table 5 **HM_EVENT Structure**

Field	Description
code	The code associated with the event. If the code is **HM_MSG**, the event is a message.
subCode	The subcode associated with the event. If a subcode is not needed, the field is 0. For injecting events, applications must use **HM_SUB_CODE_USER** (defined in **hmTypes.h**) + *offset*. If the event has been reformatted, the field has the previous value of code. See *Reformatting Events* below.
historicalCode	If the event has not been reformatted, the field is 0. If the event has been reformatted, the field has the previous value of **subCode**. See *Reformatting Events* below.
level	The level at which the event was initially dispatched.
timeStamp	The time (in microseconds) when the event was injected.
sysStatus	The status of the system when the event was injected. For details, see *System Status and Modes* below.
addInfo	Additional information specified by the injector.
addr	The address where the injection was made.
partNumber	The partition number, indicating from which partition the event was injected. If the event was not injected from a partition, the field is 0.
taskName	A **NULL**-terminated string representing the name of the task that injected the event. If the event was injected from an interrupt context, the string is **INTERRUPT**.
taskId	The task ID of the task that injected the event. If the event was injected from an interrupt context, the field is 0.
msgLen	The length (in bytes) of the message body (**msg**). In the case of an exception, this is the sum of the sizes of "`EXC_INFO\0`" and the **EXC_INFO** structure.
msg	The message body, also known as the event payload and message payload. If the event is the result of an exception, **msg** contains text

Field	Description
	and data: the string "EXC_INFO\0" followed by the **EXC_INFO** data structure. If the event is the result of a second reformatting, see *Reformatting Events* below.

System Status and Modes

When an event is injected, the health monitor facility determines and sets the value of the systemStatus field in the **HM_EVENT** structure and passes it to the system health monitor, which uses it to determine the level to which to dispatch the event.

The **systemStatus** is a bitmap that can have one of the following values:

- **HM_MODULE_MODE**
- **HM_PARTITION_MODE**
- **HM_PROCESS_MODE**

Each injection of a health monitor event results in partitions or processes being preempted. The mechanism is not explicit. It results from the fact that the event is dispatched to a higher-priority task for handling, so that the partition OS scheduler preempts the current injecting task.

System Status for MOS Context

For code running in the MOS context, system status can be the logical OR of the following:

- **HM_MODULE_HM_STATUS**
- **HM_MODULE_INIT_STATUS**
- **HM_PARTITION_HM_STATUS**
- **HM_PARTITION_SWITCH_STATUS**
- **HM_SYS_FUNC_STATUS**
- **HM_SYSCALL_STATUS**

System Status for Partition OS Context

For code running in the partition OS context, system status can be the logical OR of the following:

- **HM_PARTITION_INIT_STATUS**

  Even though this status corresponds to code running in the partition OS, the mode of the system is still partition mode, because the partition OS is not yet prepared to handle events.

- **HM_PROCESS_EXEC_STATUS**
- **HM_PROCESS_MGMT_STATUS**

Module Mode (**HM_MODULE_MODE**)

Injections made from **HM_MODULE_MODE** are dispatched to the module level only. All partitions are preempted until the module-level handler handles the alarm. The handler can control the duration of preemption. But, since the situation that caused the alarm probably needs to be corrected before it is safe for partitions to continue running, the duration would

not be an issue. If this is not the case, the alarm probably could and should have been handled by the partition health monitor.

HM_MODULE_MODE is equal to the logical OR of the following status values:

- **HM_MODULE_HM_STATUS**—module health monitor task
- **HM_MODULE_INIT_STATUS**—module initialization
- **HM_PARTITION_SWITCH_STATUS**—rescheduling as a result of a partition switch
- **HM_SYS_FUNC_STATUS**—not a health monitor task or a partition-related task; that is, an ISR
- **HM_UNKNOWN_STATUS**—unable to determine the system status

Partition Mode (**HM_PARTITION_MODE**)

Injections made from **HM_PARTITION_MODE** are dispatched to the partition or module levels. The current partition is preempted, and the handler runs.

HM_PARTITION_MODE is equal to the logical OR of the following status values:

- **HM_PARTITION_HM_STATUS**—partition health monitor task
- **HM_PARTITION_INIT_STATUS**—partition OS initialization
- **HM_SYSCALL_STATUS**—in a MOS task that is related to a partition, but not the partition health monitor task

Process Mode (**HM_PROCESS_MODE**)

Injections made from **HM_PROCESS_MODE** are dispatched to the process, partition, or module levels. The current task in the partition is preempted until the alarm is handled.

HM_PROCESS_MODE is equal to the logical OR of the following status values:

- **HM_PROCESS_EXEC_STATUS**—running a partition OS task
- **HM_PROCESS_MGMT_STATUS**—in the partition OS kernel or partition OS interrupt state

Injecting Alarms

The MOS or an application injects an alarm by calling **hmEventInject( )** or **HM_EVENT_INJECT( )** with a code other than **HM_MSG**. ARINC 653 applications must issue the **RAISE_APPLICATION_ERROR** service with the **APPLICATION_ERROR** code.

Information about the alarm injection is collected from the viewpoint of where the injecting routine is called, which is not necessarily where the fault occurred. For instance, when **HM_EVENT_INJECT( )** is called, the address is that of the program counter when **HM_EVENT_INJECT( )** was called.

HM_EVENT_INJECT( ) is a macro to **hmEventInject( )** that fills in the *addr* and *taskId* parameters to the program counter and the current task ID at the time that **HM_EVENT_INJECT( )** is called.

The following routines can be used when calling **hmEventInject( )**:

- **taskPcGet( )**—Specifies the program counter of a non-running task
- **vxCurrentPcGet( )**—Specifies the current program counter

The alarm goes first to the system health monitor table, which decides at what level to dispatch. Alarm Injection shows the logic of the alarm injection. Alarm Dispatch shows the subsequent dispatching to the appropriate level for handling.

Figure 13: Alarm Injection

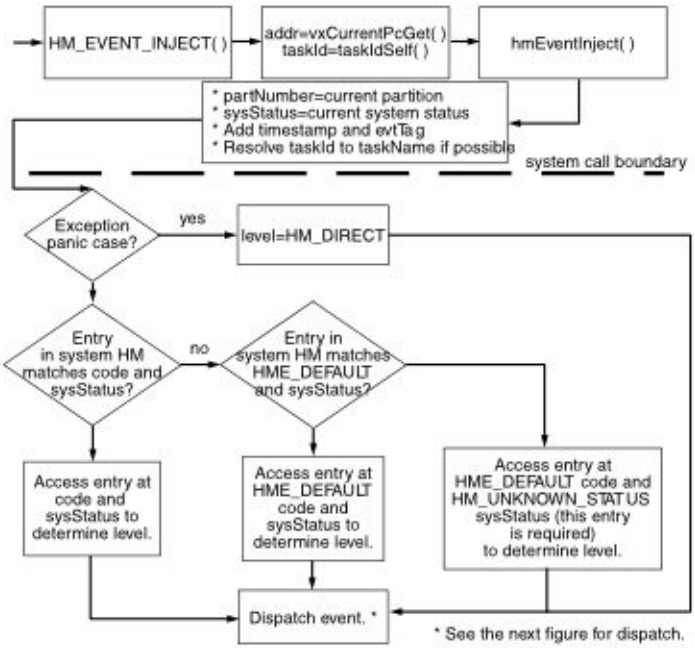
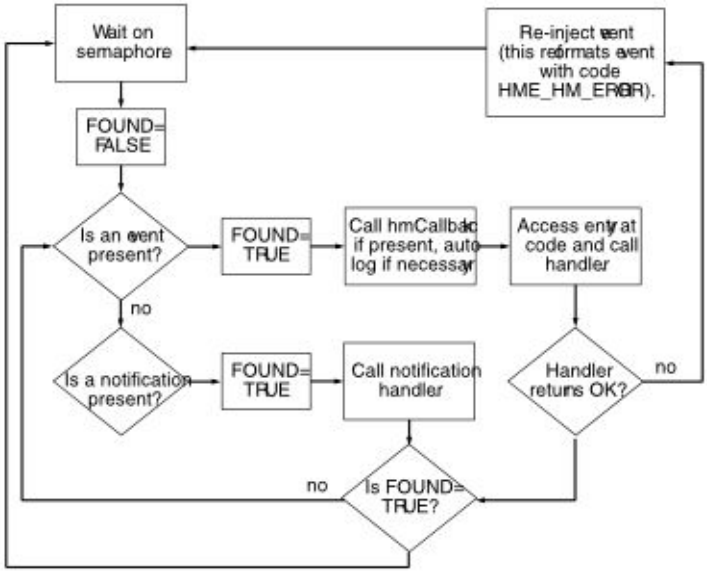


Figure 14: Alarm Dispatch



Dispatching Rules

This section outlines the rules for dispatching events. In the descriptions, interrupt context includes ISRs, watchdog routines, and kernel hooks (for example, partition switch hooks).

No Process Health Monitor Installed

Events dispatched to the process health monitor when the process health monitor is not installed are dispatched unchanged to the partition health monitor.

Alarms Injected in Exception or Interrupt Context

Alarms injected from exception or interrupt context are treated like alarms injected from task level. This means that the handling of events when injected from an ISR is deferred to task context. Except in the case of an exception panic, events injected due to an unhandled exception have their information (such as task information and system status) set according to what was happening at injection time.

In the case of an exception panic, the module-level handler is called directly. An exception panic can be any of the following:

- an exception in kernel state
- an exception from interrupt level
- a nested exception
- an exception in the root task

Events Injected from Tasks in the Partition OS

Events injected from a task in the partition OS that has a priority equal to the process health monitor (error handler) task, which ordinarily are dispatched to the process level, are dispatched unchanged to the partition level. This includes injection from the process health monitor task. The exception is for the **APPLICATION_ERROR** event code, which can be injected from the error handler process and handled by it.

Events Injected from Tasks Outside the Partition OS

- Priority Greater than or Equal to the Partition Health Monitor Task

  Events injected from a task outside the partition OS that has a priority equal to or greater than the partition health monitor task, which ordinarily are dispatched to the partition level, are dispatched unchanged to the module level. This includes injection from the partition health monitor task.

- Priority Equal to the Module Health Monitor Task

  Events injected from a task outside the partition OS that has a priority equal to the module health monitor task, have their handlers called directly and synchronously. This applies to injection from within a module health monitor handler only.

Full Health Monitor Queues

If dispatch is not possible because the partition health monitor queue is full, the event is reformatted with the **HME_HM_ERROR** code and dispatched for handling at the module level according to the rules for injecting from the partition health monitor task.

If dispatch is not possible because the process health monitor queue is full, the event is reformatted with the **HME_HM_ERROR** code and dispatched for handling at the partition level according to the rules for injecting from the process health monitor task.

If dispatch is not possible because the module health monitor queue is full, the module-level **HME_HM_ERROR** handler is called directly and synchronously according to the rules for injecting from the module health monitor task.

Task-Lock Condition Exists

If an event is injected while a task-lock condition exists, the following rules are followed:

- Injected from the Task Context

  If an event is injected from the task context while a task-lock condition exists, the lock is broken and the error handler process runs. However, before breaking the lock, the

partition OS raises the task's priority to one less than the error handler's priority. This strategy is an attempt to have the task regain its task lock. The preempted task then runs immediately after the error handler, at which point the partition OS restores the task-lock count and original priority. However, if an application uses task priorities 0 or 1 (contrary to the ARINC 653 specification), there is no guarantee that the preempted task is the first to run after the error handler.

- Injected from the Interrupt Context

  If an event is injected from the interrupt context while a task-lock condition exists and **watchDogDuration** (as specified in the partition XML configuration file) is 0, locks (preemption and task locks) are broken immediately and the error handler process runs. The error handler process restores the locks and instructs the partition OS to run the preempted task first (unless the task has stopped).

  If **watchDogDuration** is **INFINITE_TIME**, the error handler process runs when the task unlocks itself.

  If **watchDogDuration** is between **0** and **INFINITE_TIME**, any APEX locks (the result of the application issuing the **LOCK_PREEMPTION** service) are broken immediately. If there are any other task locks remaining (the result of the kernel calling **taskLock( )**), the watchdog is started. When the watchdog expires, task locks are broken, and the error handler process runs. The error handler process restores the locks and instructs the partition OS to run the preempted task first (unless the task has stopped).

Handlers Cannot Handle the Alarm

Handlers that cannot handle an alarm must inject an alarm of their own or return **ERROR** so that the health monitor can reformat the event with the **HME_HM_ERROR** code.

However, returning **ERROR** does not apply at the process health monitor level. At this level, the application must inject another event from the process health monitor, which the health monitor reformats with the **HME_HM_ERROR** code. For rules pertaining to injecting from the process health monitor task, see Events Injected from Tasks Outside the Partition OS above.

All system health monitor tables should include a handler for **HME_HM_ERROR**.

The return code from calling an **HM_DIRECT**-level handler is not checked, because no further escalation is possible. If an **HME_HM_ERROR** module-level handler is not declared (or any handler for that matter), the **HME_DEFAULT** handler is called.

Reformatting Events

When the health monitor facility reformats an event, the event gets a new code. The old code moves to the **subCode**, and the old **subCode** moves to the **historicalCode**. If this is a second reformatting, **historicalCode** information would be lost. In this case, an event is injected with the following information:

- code of **HME_DATA_LOSS** (**subCode** of 0, **historicalCode** of 0)
- **msg** containing the old code, **subCode**, and **historicalCode** (retrievable by casting **msg** as an **HM_DATA_LOSS** data structure)

Dismissing Alarms

An alarm is dismissed under either of the following circumstances:

- In the partition health monitor table or module health monitor table, for a given code, a NULL handler (**CFG_NO_HANDLER**) is set.
- In the system health monitor table, for a given code and **systemStatus** combination, the dispatch level is set to **HM_NO_LVL**.

Dispatching and Logging Messages

A message event differs from an alarm event as follows:

- Alarms are dispatched according to the system health monitor table. Messages are not dispatched by this mechanism. If a message is injected within the partition, it is dispatched to the partition health monitor. If it is injected outside the partition, it is dispatched to the module health monitor. Each level has a default handler that logs the message to the partition or module log. The system integrator can replace the default handler through the XML configuration file.
- Alarms are logged only if automatic logging is enabled or if a handler explicitly logs the alarm by calling **hmEventLog( )**. The system integrator must specify the handler in the XML configuration file.

Health Monitor Thresholds

Thresholds apply to the following:

- notification queues
- logs
- event queues

Notification Queue Threshold

The notification queue threshold equals the depth of the notification queue. An event is injected when the notification queue overflows. If overflow notification is disabled and if an event is injected because of overflow, the event is dispatched to the partition health monitor or module health monitor with which the notification queue is associated.

Log Threshold

The log threshold defines if and when an event should be injected when the log has a certain number of entries.

Event Queue Threshold

The event queue threshold defines if and when an event should be injected when the event queue has a certain number of entries.

Error Handler Queue Threshold

The error handler queue threshold defines the event threshold of the error handler. It is analogous to the event queue threshold that applies to the partition health monitor or module health monitor.

Health Monitor Actions

When an application runs, the health monitor facility: escalates alarms, logs events, notifies other partitions, and issues callbacks.

In addition to detecting and reporting its own faults, an application needs to respond to the actions of the health monitor.

Escalating Alarms

Alarms are not automatically escalated, because it is the system integrator who knows the level and handler that best services each alarm. The system integrator configures the system health monitor table in the XML configuration file.

If the specified handler from the partition health monitor table or module health monitor table cannot handle the alarm, it should return **ERROR** or inject an alarm of its own.

If a handler returns **ERROR**, this indicates that the alarm was not handled correctly. In this situation, the health monitor facility reformats the alarm, using the first alarm information, but with code equal to **HME_HM_ERROR** (see *Reformatting Events* in Basic Health Monitor Concepts on page 38).

As a result of the above, the system integrator can choose whether to handle alarms that result from alarms not being handled and, if so, which ones.

NOTE: Only the health monitor facility can inject alarms with the **HME_HM_ERROR** code. This is enforced so that a rogue task cannot directly inject these types of alarms, possibly forcing escalation and affecting the protection guarantees between partitions.

Logging Events

When logging is enabled, if an event is injected from within a partition (**sysStatus** is **HM_PROCESS_MODE** or **HM_PARTITION_MODE**), the event is logged to the partition health monitor log.

When logging is enabled, if an event is injected from outside the partition (**sysStatus** is **HM_MODULE_MODE**), the event is logged to the module health monitor log.

When logging is not enabled, a handler must call **hmEventLog( )**, which might generate an additional system call to log the alarm, depending on where the current event processing is occurring.

Application code can log messages to the health monitor log by calling **hmEventInject( )** with the **HM_MSG** code (assuming the default handler behavior).

There is one log for each partition and one log for the VxWorks 653 module. By default, the logs are stored in volatile memory. If logs need to survive module or system restart, handlers need to be provided that write the logs to non-volatile memory.

Each log is a circular buffer of configurable size. As a result, for log sizes greater than zero, the request to log an event is never denied. For a log size of n, only the n most-recent entries are in the log. Old information could be overwritten.

Logs can be accessed by calling **hmLogEntriesGet( )** with the partition ID and the number of entries to retrieve. Partition ID 0 accesses the module log. The log can be read in FIFO or LIFO order. Entries that have been read can be preserved in the log or purged. An offset can be specified to retrieve the log in segments. Read-purge with LIFO-order reading is not permitted, because it would result in a discontinuous log. The *HM_EVENT Structure* table in Basic Health Monitor Concepts on page 38 lists the information in each entry of each log.

Detecting and Reporting Application Errors

Applications are responsible for detecting their own errors and reporting (injecting) associated alarms.

An application can fail in such a way that it cannot correctly report the failure or cannot report a failure at all. System integrators may need to account for this possibility when they design the overall system.

Reporting for ARINC 653 Applications

For an application to conform to the ARINC 653 specification, it must use the APEX API. That is, to inject an alarm, it must issue the **RAISE_APPLICATION_ERROR** service and must report only the **APPLICATION_ERROR** ARINC 653-defined error. Other errors must be identified by potentially non-portable use of the service's message parameter.

The message that the **RAISE_APPLICATION_ERROR** service passes is read with the **GET_ERROR_STATUS** service. If the partition's error handler is created, it is then started to take the recovery action for the process that raises the error code. If the error handler is not created, the error is considered a partition-level error.

ARINC 653 Errors and Their Health Monitor Alarm Code Equivalents shows all the ARINC 653-defined errors, their numeric values, and equivalent health monitor alarm codes. If the application uses this service to inject any of the health monitor alarm codes or health monitor extended codes, the events are lost. The system integrator is responsible for preventing this.

ARINC 653 applications that want to handle a health monitor event code can first determine its equivalent ARINC 653 error by issuing the **GET_ERROR_STATUS** service.

If an ARINC 653 application wants to inject and handle health monitor alarm codes within the partition OS context, it would need to call **HM_EVENT_INJECT( )**, **hmEventInject( )**, or **hmErrorHandlerEventGet( )**. Since this does not comply with the ARINC 653 specification, the partition or module health monitor must handle them.

ARINC 653 Errors and Health Monitor Equivalents

Table 6 **ARINC 653 Errors and Their Health Monitor Alarm Code Equivalents**

ARINC 653 Error (Value)	Health Monitor Alarm Code	Examples
APPLICATION_ERROR (1)	**HME_APPLICATION_ERROR**	Errors raised by application processes
DEADLINE_MISSED (0)	**HME_DEADLINE_MISSED**	Process deadline violations
HARDWARE_FAULT (6)	**HME_HARDWARE_FAULT**	Memory-parity errors, I/O-access errors
ILLEGAL_REQUEST (3)	**HME_ILLEGAL_REQUEST**	

ARINC 653 Error (Value)	Health Monitor Alarm Code	Examples
		Illegal OS request by a process
MEMORY_VIOLATION (5)	**HME_MEMORY_VIOLATION**	Memory-protection errors, supervisor privilege violations
NUMERIC_ERROR (2)	**HME_NUMERIC_ERROR**	Overflow errors, divide by zero, floating-point errors
POWER_FAIL (7)	**HME_POWER_FAIL**	Notification of power interruption so that, for example, application-specific state data can be saved
STACK_OVERFLOW (4)	**HME_STACK_OVERFLOW**	Process stack overflow

Reporting for Non-ARINC 653 Applications

Applications that do not need to conform to the ARINC 653 specification can inject alarms by calling **HM_EVENT_INJECT( )** or **hmEventInject( )**. For more information about these routines, see *Injecting Alarms* in Basic Health Monitor Concepts on page 38.

A partition that uses the APEX layer and wants to inject alarms within the partition OS context can inject an alarm by issuing **RAISE_APPLICATION_ERROR** with the **APPLICATION_ERROR** ARINC 653-defined code.

Initializing the Health Monitor

The MOS is responsible for initializing the health monitor. This is accomplished in stages.

The health monitor facility is initialized in the following stages:

1. The MOS initializes the module health monitor after it enables support for protection domains and before it initializes the ARINC 653 schedule.
2. As the MOS creates each partition, it initializes the associated partition health monitor after it assigns the partition to the proper window, but before it activates the partition.
3. If the application requested it, the MOS creates the error handler for the application.

Getting Health Monitor Information at Run-time

Configuration information is specified for the partition and module health monitors in the XML configuration file. User-supplied code can get this information by using **sicvLib** reference entries with the appropriate field selector.

The field selectors for reference entries are as follows:

- **HM_ATTRIBUTE_MASK**
- **HM_CALLBACK**
- **HM_ENTRY_COUNT**
- **HM_ERROR_HANDLER_QUEUE_THRESHOLD**
- **HM_EVENT_CODE**
- **HM_EVENT_FILTER_MASK**
- **HM_HANDLER**
- **HM_LOG_ENTRIES_THRESHOLD**
- **HM_MAX_ERROR_HANDLER_QUEUE_DEPTH**
- **HM_MAX_LOG_ENTRIES**
- **HM_MAX_QUEUE_DEPTH**
- **HM_NOTIF_MAX_QUEUE_DEPTH**
- **HM_NOTIFICATION_HANDLER**
- **HM_QUEUE_THRESHOLD**
- **HM_STACK_SIZE**
- **HM_TRUSTED_PARTITION_MASK**

Defining the Health Monitor Handler Table

The health monitor handler table, which is defined in the MOS BSP, in the file **hmActions.xml**, must define all handlers that are specified in the health monitor configuration.

At initialization time, handler names in the table are resolved to function pointers. For more information, see the *VxWorks 653 Configuration and Build Guide*.

Guidelines for Writing Handlers

Handlers should not make blocking calls without first making sure the alarm's injector cannot run until the health concern is fully handled. The health monitor facility assumes the handler is called synchronously within the context of the task or interrupt that injects the alarm. The facility

dispatches the alarm to the appropriate level. In addition, it assumes the health monitor task at that level preempts the current context (the one that injected the alarm) or, if the alarm is injected from interrupt context, the handler is intentionally deferred. Thus, if the handler is pended due to a blocking call, the injecting context (if the injector is a task and not an interrupt handler) might be scheduled to run without having fully handled the health concern that occurred in that task. In such a situation, it might be desirable to suspend the offending task before issuing a blocking call in the context of the handler.

Other Facilities That Inject Alarms

In addition to the health monitor, other facilities that inject alarms include the MOS, partition OS, and APEX layer.

The MOS injects alarms when conditions cause the system to restart.

The partition OS injects alarms under various conditions, such as when conditions cause the partition to restart.

The APEX layer injects alarms according to the ARINC 653 specification; for example, when a process misses its deadline.

Where possible and where appropriate, faults are mapped to a health monitor equivalent of an ARINC 653-defined code. This is especially true when handling the fault is possible or appropriate at the process level, such as in the case when applications generate exceptions.

HM Public Information

Learn about the public health monitor information and where it is located.

For information about a library and its routines, see their reference entries.

Table 7 **Health Monitor Public Information**

Type of Information	Location	Details
Library header files	***installDir*/vxworks-cert-6.6.*x*/target/h/hmLib.h**	Header file for the POS
Constants and data structures	***installDir*/vxworks-cert-6.6.*x*/target/h/hmTypes.h**	• System status fields and modes • Dispatch levels • ARINC 653-defined event codes • Wind River-defined event codes

Type of Information	Location	Details
		• Dispatch levels • Event subcodes
APIs	**hmLib** **hmErrorHandlerLib** **hmShow** **hmVbi**	

5

MIPC

MIPC Introduction

The Multi-OS Interprocess Communication (MIPC) facility provides *socket-like* routines that correspond to standard socket APIs.

In VxWorks 653, Safe IPC is used as the transport mechanism for MIPC.

Adding MIPC to VxWorks Partitions

To provide a VxWorks Cert or non-cert partition with MIPC over Safe IPC support, include **INCLUDE_MIPC_HV_SAFE** in the kernel.

For information on how to include components and folders, refer to the *VxWorks 653 Configuration and Build Guide*.

MIPC requires two Safe IPC channels configured bi-directionally between the two communicating partitions. They may be configured as desired. The transport layer can be configured to operate in either of the following modes:

- the data movement is performed by the MOS (or **@Transport**: **MODULE**). The partition issues a hypercall to send and receive messages on the port. The control structures are only accessible to the MOS, so this is always safer. This is the default.
- the data transport is performed by the partitions. The control structures and message buffers are in shared memory accessible to the partitions. The partitions can send and receive messages without a hypercall. It also means **@Transport**: **PARTITION** is less safe, since a partition has the ability to put a channel in an inconsistent state and cause a failure in the peer. **PARTITION** is never appropriate between partitions at different criticality levels. This requires fewer hypercalls and may provide an increase in efficiency and performance.

Additionally, the effects of the remainder of the channel configuration are mostly transparent to the MIPC application:

- The recommended default is **@FlowControl**:**INTERRUPT** with **@WhenFull**: **BLOCK**.
- **@FlowControl**:**POLL** with **@WhenFull**:**DISCARD** will increase the latency of messages. In partitions of mixed criticality, where the higher criticality partition does not want to be interrupted use **POLL**. **INTERRUPT** is not supported with **POLL**. MIPC messages are not lost when the required **DISCARD** is used for the IPC port, the application will pend or return **EWOULDBLOCK** depending on the socket configuration.

mipc_sockaddr Socket Address Structure

The **mipc** library defined its socket address based on the socket it is bound to as well as the socket address family.

The **mipc** library defines its socket address structure as follows:

```
struct mipc_sockaddr
{
    unsigned short family;
    unsigned short node;
    unsigned short port;
};
```

where:

- *family* is the socket address family: **MIPC_AF**.
- *node* is the number of the node that the socket is bound to. When using the SafeIPC back-end the node is the index of the SafeIPC port. SafeIPC ports are distinct from, and should not be confused with the MIPC port number. The function **mipc_getnodebyname( )** can be used to obtain the node number in both cases.
- *port* is the number of the port that the socket is bound to.

For use with the **mipc_bind( )** routine, the **MIPC_PORT_ANY** macro can be used to define the *port* field (see *MIPC_PORT_ANY Macro for mipc_sockaddr Structure with mipc_bind( )* in MIPC Socket-Like Routines on page 55).

For more information on the structures and constants, such as error codes, see: ***installDir*/vxworks-cert-6.6.*x*/target/h/mipcHv/mipc.h**.

MIPC Socket-Like Routines

The MIPC socket-like API provides functions for opening and closing sockets, creating ports for services and binding them to sockets, and for sending and receiving data on a socket.

This section presents the routines that make up the MIPC socket-like API. The routines corresponding to standard socket routines are named with the **mipc_** prefix. For example, **mipc_bind( )** corresponds to **bind( )**.

NOTE: Do not call **mipc** library routines at interrupt level. Doing so may result in unpredictable behavior.

Application Considerations

In writing applications that use the API, the following points need to be considered:

- The MIPC library does not provide a mechanism for preventing two applications or two threads within an application from accessing a socket at the same time.

  As a result, applications need to access sockets in a single-threaded manner and ensure that there is no conflict with other applications attempting to access a socket.

- A MIPC socket can only be bound to a single port.

  In this document, when the term *port* is used, it can also refer to the socket bound to the port.

- Ports 0-31 are reserved for Wind River's use (but reservation is not enforced programmatically).

- A node, in the context of MIPC programming, is an instance of an operating system running in a CPU; node numbers do not always correspond to CPU numbers.

MIPC Socket-Like Routines

The MIPC socket-like API is summarized in the following table.

Table 8 **MIPC Socket API**

Socket Routine	Description
mipc_accept( )	Accept a request for a connection to a MIPC socket.
mipc_addr_verify( )	Verify that a MIPC socket exists at a given address.
mipc_bind( )	Bind an address to a MIPC socket.
mipc_close( )	Close a MIPC socket.
mipc_connect( )	Request a connection to a MIPC socket.
mipc_getmtu( )	Get the MTU of a MIPC channel.

Socket Routine	Description
mipc_getnodebyname( )	Get the MIPC node number associated with a name.
mipc_getpeername( )	Get the address of a peer MIPC socket.
mipc_getsockname( )	Get the address of a MIPC socket as a **mipc_sockaddr** structure.
mipc_getsockopt( )	Get the value of an option associated with a MIPC socket. See Socket Options below.
mipc_listen( )	Enable a MIPC socket to receive connection requests.
mipc_recv( )	Receive data from a MIPC socket (returns only the incoming message, but without any information about the sender). Designed primarily for use with **STREAM** and **SEQPACKET** sockets as they are connection-based (the message can only come from one possible source).
mipc_recvfrom( )	Receive data from a MIPC socket (returns the message, along with the socket address of the sender of the incoming message). Designed primarily for with **DGRAM** and **RDM** sockets as they are connectionless, and the sender's socket address is needed if the application wants to send back a reply using **mipc_sendto( )**.
mipc_send( )	Send a message to a MIPC socket.
mipc_sendto( )	Send a message to a specified MIPC socket.
mipc_setsockopt( )	Set the value of an option associated with a MIPC socket. See Socket Options below.
mipc_shutdown( )	Shut down communication by a MIPC socket.
mipc_socket( )	Create a MIPC socket.

For detailed information on individual routines, see the *VxWorks 653 POS MIPC API Reference*.

MIPC_PORT_ANY Macro for mipc_sockaddr Structure with **mipc_bind( )**

The **MIPC_PORT_ANY** macro tells MIPC to choose an available port number to bind to. To get the port number, use the **mipc_getsockname( )** routine.

Socket Options

The following socket options are supported:

MIPC_SO_NBIO

Specifies if the socket performs non-blocking I/O. The value 1 signifies that I/O operations always return immediately, while the value 0 signifies that I/O operations do not return until they either complete or time out.

MIPC_SO_SEND_POLL_RATE

Specifies the number of ticks to sleep between iterations when polling on a full send channel. This is zero by default.

MIPC_SO_CONN_TIMEOUT

Specifies the maximum time to wait for completion of a connection attempt initiated by the socket, in ticks. The value **MIPC_TIMEOUT_FOREVER** (default) signifies no limit.

MIPC_SO_SEND_TIMEOUT

Specifies the maximum time to wait for completion of a send operation initiated by the socket, in ticks. The value **MIPC_TIMEOUT_FOREVER** (default) signifies no limit.

MIPC_SO_RECEIVE_TIMEOUT

Specifies the maximum time to wait for completion of a receive operation initiated by the socket, in ticks. The value **MIPC_TIMEOUT_FOREVER** (default) signifies no limit.

MIPC_SO_RECV_CALLBACK

Specifies a routine to be invoked prior to queuing a message on the socket.

The routine must have the following signature:

```
typedef int (*MIPC_RECV_CALLBACK)
    (int sd,
    void * buf,
    size_t buflen,
    unsigned short node,
    unsigned short port);
```

The routine must return one of the following values:

- **MIPC_RECV_QUEUE_MESSAGE**: queue the message to the socket
- **MIPC_RECV_RELEASE_MESSAGE**: do not queue the message to the socket

The routine executes in the context of a MIPC system thread and should refrain from calling routines that may block.

A **NULL** value at the location pointed to by **optval** will remove an installed callback routine.

MIPC_SO_RECV_QUEUED_CALLBACK

Specifies a routine to be invoked after queuing a message on a socket.

The routine must have the following signature:

```
typedef int (*MIPC_RECV_QUEUED_CALLBACK) (int sd);
```

The routine executes in the context of a MIPC system thread and should refrain from calling routines that may block.

A **NULL** value at the location pointed to by optval will remove an installed callback routine.

For more details, see the *VxWorks 653 POS MIPC API Reference*.