MicroTCA™

Standard Process Model Design Guide

Guidelines for designing multi-threaded software for MTCA.4 systems

MTCA_DG.3 R1.0

August 30, 2017

PICMG®

Open Modular Computing Specifications

μTCA®

Contents

Figures

Tables

Caveat

This design guide is not a specification. It contains additional detail information but does not replace any applicable PICMG specifications.

For complete requirements on the design of MTCA.4 or MTCA.4.1 compliant boards and systems, refer to the full specification – do not use this design guide as the only reference for any design decisions.

1 Preface

1.1 About This Document

This guideline defines the functions and Application Programming Interfaces (APIs) for the *Standard Process Model (SPM)* developed for the MTCA.4 effort within PICMG. It is applicable to systems developed in conjunction with the standards released as part of that effort, and more generally to systems developed for instrumentation and machine control applications.

This guideline defines a standard operating model and Application Programming Interface (API) for code development to facilitate module re-use and portability. It is recommended, but not required, that applications developed for use with MTCA.4 or MATCA.4.1 systems make use of these guidelines to the greatest reasonable extent.

1.2 Intended Audience

This design guide is intended for software engineers and programmers designing software for use with MTCA.4 or MTCA.4.1 systems.

1.3 No Special Word Usage

Unlike a PICMG specification, which assigns special meanings to certain words such as "shall", "should" and "may", there is no such usage in this document. That is because this document is not a specification; it is a non-normative design guide.

1.4 No Statements of Compliance

As this document is not a specification but a set of guidelines, there should not be any statements of compliance made with reference to this document.

1.5 Correctness Disclaimer

The code examples given in this document are believed to be correct but no guarantee is given. In most cases the examples come from designs that have been built and tested.

1.6 Name and Logo Usage

The PCI Industrial Computer Manufacturers Group's policies regarding the use of its logos and trademarks are as follows:

Permission to use the PICMG organization logo is automatically granted to designated members only as stipulated on the most recent Membership Privileges document (available at www.picmg.org) during the period of time for which their membership dues are paid. Nonmembers must not use the PICMG organization logo.

The PICMG organization logo must be printed in black or color as shown in the files available for download from the member's side of the Web site. Logos with or without the "Open Modular Computing Specifications" banner can be used. Nothing may be added or deleted from the PICMG logo.

The PICMG® name and logo are registered trademarks of The PICMG®. Registered trademarks must be followed by the ® symbol and the following statement must appear in all published literature and advertising material in which the logo appears:

PICMG and the PICMG logo are registered trademarks of the PCI Industrial Computer Manufacturers Group.

1.7 Intellectual Property

The Consortium draws attention to the fact that implementing recommendations made in this document could involve the use of one or more patent claims ("IPR"). The Consortium takes no position concerning the evidence, validity, or scope of this IPR.

Attention is also drawn to the possibility that some of the elements of this specification could be the subject of unidentified IPR. The Consortium is not responsible for identifying any or all such IPR.

No representation is made as to the availability of any license rights for use of any IPR that might be required to implement the recommendations of this Guide. This document conforms to the Specification Development and does not contain any known intellectual property that is not available for licensing under Reasonable and Nondiscriminatory terms. In the course of Membership Review the following disclosures were made:

Necessary Claims (referring to mandatory or recommended features):

- No disclosures in this category were made during Member Review

Unnecessary Claims (referring to optional features or non-normative elements):

- No disclosures in this category were made during Member Review

Third Party Disclosures (Note that third party IPR submissions do not contain any claim of willingness to license the IPR.):

1.8 Copyright Notice

1.9 Acronyms and Abbreviations Used

Table 1-1: Acronyms and Abbreviations Used

Acronym	Definition
API	Application Programming Interface
POSIX	Portable Operatng System Interface for Unix
SPM	Standard Process Model
xTCA	ATCA and/or µTCA

Acronym	**Definition**
µTCA	Micro-TCA

1.10 Signal Table Terminology

For purposes of describing the Standard Process Model and its environment, the following terminology is applicable:

Table 1-2: Model Terminology

Term	**Definition**
	No special terms are defined

2 Introduction

2.1 Overview

The Advanced Telecommunication Computing Architecture (ATCA) specification and related standards maintained by the PCI Industrial Computer Manufacturers Group (PICMG) define an infrastructure for development of distributed data processing systems for the telecommunications industry. At the urging of the experimental physics community a set of committees were formed under the auspices of PICMG to extend the ATCA family of standards for use in embedded instrumentation and machine control applications, with particular emphasis on scientific applications. These committees, known collectively as the MTCA.4 or MTCA.4.1 committees, were chartered to adapt and/or extend the existing ATCA family specifications to accommodate sensor and control I/O, including: support for analog signals; timing, synchronization, and interlock mechanisms required for instrumentation and control operations; and low latency data distribution protocols to support time-critical data processing and control constraints.

As part of the standardization effort the committees also defined common development architectures to encourage hardware and software component interoperability and portability among the various scientific centers. To that end, guidelines for various aspects of software development of MTCA.4 or MTCA.4.1 systems were developed and published.

This guideline defines the function and usage for a Standard Process Model (SPM) for use in MTCA.4 or MTCA.4.1 software applications.

2.2 Nomenclature

Recommendations are designated within this document by the words **shall**, **will, should”, is,** and **are.** These terms are interchangeable and usage is driven entirely by context and stylistic considerations.

Although this is a guideline, rather than a standard, its intent is to define mechanisms and practices that facilitate interoperability of software modules across different projects and facilities. As such, it is useful to provide language that allows a discussion among implementers about the degree to which their applications do or do not conform to the recommendations presented here, and therefore the degree to which they can expect that goal of interoperability to be achieved. To that end, four levels of recommendation may be identified:

- **Mandatory**: These recommendations must be implemented to be fully conformant with this guideline.

- **Required:** These recommendations must be implemented to be fully conformant with this guideline, but may be excluded from a particular implementation provided the release notes for the implementation specifically identify them as areas of non-conformance. An implementation with exclusions may be considered “conditionally conformant”.

- **Desired:** These recommendations comprise design goals, performance targets, and “nice to haves” that are desirable but not necessary for full conformance with this guideline. Application developers should not depend on availability of these items on all implementations.

- **Guidance**: These recommendations are provided for guidance and clarification to the designer regarding the intent of other specific recommendations, expected guideline usage, external interfaces and constraints, and preferred or potential design approaches, technologies, and practices. These items should be taken into account during design and deviations may be noted within the standard design documentation, but they are not binding for design.

Required, Desired, and Guidance recommendations must be specifically designated as such within the text of the specification. Any recommendation without such a designation is presumed to be *Mandatory*.

2.3 Background

2.3.1 Context and Rationale

The *Standard Process Model (SPM)* defined in this guideline is intended to apply to applications developed in conjunction with the MTCA.4 or MTCA.4.1 family of standards from PICMG. These applications typically involve large networks of distributed computing elements, sensors, actuators, and signal generators forming integrated control and data acquisition/analysis systems. Although such systems will often make use of large general-purpose computing platforms for system monitoring/control and offline analysis, a large fraction of software components in such systems operate within embedded environments, remote from the high-level control/analysis systems and with soft and hard real-time requirements.

Further, the operating environment for such systems is anticipated to be dynamic in terms of both hardware and software configuration as experiments evolve, and to involve a high degree of sharing, both of expertise and of actual hardware and software modules, among the various laboratories participating in the work.

Both the dynamic nature of the environment and the degree of sharing between facilities leads to a strong desire for a software development infrastructure that facilitates rapid prototyping of new components, efficient code development and validation, and code portability. That desire has been formalized by the PICMG MTCA.4 or MTCA.4.1 Software and Protocols committee as a set of guidelines for creating standard I/O interfaces, communication protocols, and other commonly used functional blocks and APIs.

This standard defines the process model which supports those other functions and APIs. The purpose of the Standard Process Model (SPM) is to provide a standardized and portable way for software applications to manage scheduling for various threads of operation, to move data among them, and to move data in and out of those threads through the physical and logical boundaries of the machine on which they operate.

Having such a standard scheduling infrastructure is generally useful when designing applications that might need to be ported to various processor/OS platforms for use in different facilities, or for designing libraries to be adapted to disparate applications across projects and organizations. It is, in particular, necessary to support the specification and design of standard APIs to device and communication channels which must perform hardware I/O operations and must be able to asynchronously notify software modules when data is available for processing and/or when particular hardware requires attention.

Historically, those I/O and notification mechanisms – and the scheduling policies and interlocks surrounding them – have been specific to a particular operating system. In any real implementation those operating-system-specific mechanisms will still be necessary; but, to facilitate standard device APIs and general code portability, the SPM provides a layer between the operating system and the application with a standard API at the application layer. If fully and properly implemented, porting of code from one platform to another would, therefore, require only an operating-system-specific adaptation of the SPM itself with no modifications to application software modules.

2.3.1 *Guidance:* Technology and Operations Summary

A summary of the various technology and operational aspects of a generalized process model are provided for reference in Appendix A.

2.3.2 *Guidance:* Functional Requirements Summary

Figure 3-1 shows the relationship between a Standard Process Model and the Operating System/Application in a typical software environment hierarchy.

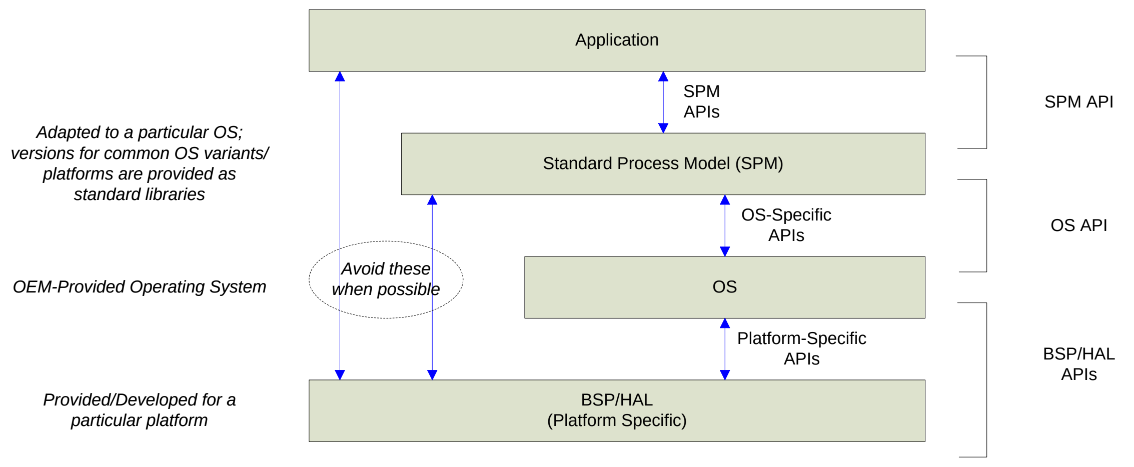

Application
SPM APIs
SPM API
Adapted to a particular OS; versions for common OS variants/ platforms are provided as standard libraries
Standard Process Model (SPM)
OS-Specific APIs
OS API
Avoid these when possible
OEM-Provided Operating System
OS
Platform-Specific APIs
BSP/HAL APIs
Provided/Developed for a particular platform
BSP/HAL (Platform Specific)

Figure 3-1: Process Model Hierarchy

To support management of and interaction with multiple devices and communication channels in a platform-independent way requires the following general categories of SPM features:

- A standard mechanism for creating and scheduling independent threads of control within a single software application to deal with different devices and communication/data streams.
- Standard mechanisms for interlocking thread access to common-use resources like devices and communication channels.
- Standard mechanisms for notifying waiting threads when a resource has become available, when a communication channel has data to deliver, and/or when a particular hardware device requires software attention.
- Standard mechanisms and models for accessing communication channels and I/O devices.

In addition, to facilitate the coordination of activities among the various threads of an application the SPM should provide a set of utility features, including:

- Standard mechanisms, which include a means of asynchronous notifications, for moving data and signaling events between application threads and for synchronizing the activities of various application threads.
- A common time-base and standard event- and interval-timer functions to facilitate temporal synchronization.

Finally, for embedded and hardware-oriented applications, which typically have some degree of real-time constraint on latency and deadlines, the mechanisms used to manage the activation and deactivation of threads should support priority-based scheduling and provide a degree of scheduling determinism.

2.3.3 The POSIX Standard

The POSIX standard was developed as a platform to facilitate code portability and operating-system independence by defining a common set of operating system functions and APIs that could be implemented across platforms as a layer between the operating system and the application. Although it was initially intended specifically to unify application APIs across various versions of UNIX it has found a wider application, with POSIX-compliant versions of many real-time kernels now available and with implementations of POSIX subsets now available as third-party libraries for popular operating systems like Windows.

As originally conceived, and based closely on UNIX, the POSIX standard provided for threads only at the process level, and provided no real-time features. However, in the late 1990s the standard was updated to include support for threads and for various real-time support functions.

The current standard is, if not ubiquitous, at least common and has all the features required to support the requirements summarized above. Further, it has been adopted and maintained as IEEE standard 1003.1, and has been through several revisions reflecting lessons learned from real-world applications.

In light of that, and in keeping with the xTCA for Physics committees' goals of adhering to existing standards where they exist and are applicable, we propose to adopt a subset of the POSIX real-time API as the basis for the SPM module for xTCA physics applications

3 Guidelines: Standard Process Model (SPM)

The SPM shall comprise a subset of the full POSIX standard plus a set of POSIX extensions. SPM extensions shall be defined in such a way that they may be implemented using elements from the standard POSIX API, and will therefore maintain compliance with POSIX via POSIX-compliant extension libraries. The SPM is defined in detail in section 3.2.

Guidance: Since SPM extensions may be provided by a POSIX-conformant library any operating system which is POSIX conformant may be used directly as a valid implementation of the SPM guideline.

3.1 POSIX Module Applicability

Guidance: The following POSIX modules and options are applicable to the Base SPM:

Base Module Option Groups

POSIX_SINGLE_PROCESS	baseline single-process management
POSIX_MULTI_PROCESS	(optional) baseline multi-process management
POSIX_SIGNALS	baseline signal support
POSIX_DEVICE_IO	general file/stream I/O
POSIX_FILE_LOCKING	thread-safe file locking
POSIX_NETWORKING	baseline sockets support
POSIX_DEVICE_SPECIFIC	standard terminal support
POSIX_DEVICE_SPECIFIC_R	thread-safe terminal support

Optional Modules

Threads	THREAD support
Realtime	Realtime extensions

For detailed information about POSIX functions and features see the POSIX standard, IEEE Std. 1003.1, 2004 Edition.

3.2 Functions and APIs

The SDM functions and APIs comprise a subset of the POSIX standard augmented by a set of useful extensions. The purpose of the extensions, in addition to simplifying the use of some POSIX features, is to create a consistent model within the SPM for managing the suspension/resumption of threads as the result of interactions with various interlock, synchronization, and communication entities.

For historical reasons, the POSIX REAL-TIME and THREAD support APIs were added as extensions to a Process-based scheduling environment with no particular real-time support.

Where those extensions created new requirements for asynchronous communication and thread suspension/activation facilities they were added in a manner consistent with a traditional threading model described in section 4.4 (Appendix A), which appears synchronous with respect to the thread – the thread identifies when it will await a specific event, is suspended until the event occurs, and then resumes its operation at the point where it stopped to wait. Any asynchronicity in such a structure results from the fact that various threads are operating, including establishing their wait points and processing received events, asynchronously with respect to each other; the act of waiting for the condition or event automatically synchronizes the thread with the event itself. Hence a threaded development model utilizes discrete threads, rather than some mechanism like a callback function, to manage asynchronicity.

POSIX, however, is not completely consistent in the use of this model for synchronization entities. Elements of the POSIX standard – in particular timers and device I/O – pre-dated the addition of the REAL-TIME and THREAD extensions, and were therefore designed to operate in a single-threaded environment in which asynchronous conditions had to be handled in some other fashion than allowing the single program thread to suspend itself. Hence, to manage asynchronicity they utilize explicit asynchronous callback mechanisms like signal catchers and asynchronous I/O termination functions rather than synchronous waits within threads. When the REAL-TIME and THREADS extensions were added to POSIX the standards committee opted to preserve the old mechanisms in deference to backward compatibility rather than to update them to conform to the new threaded model.

Hence some of the extensions to the POSIX standard proposed in this guideline for the SPM are provided specifically to map those legacy entities into functionality more consistent with the threading model described in section 4.4.

Guidance: In all cases the SPM requires availability of ‘timed-wait’ versions of interlock/synchronization/message functions that can suspend thread operation.

Usage Recommendations:

Although the POSIX standard allows and supports blocking indefinitely while waiting for synchronization events, it is recommended that threads prevent the potential for lockup conditions by utilizing synchronization APIs that provide a timeout on the wait for the synchronization condition.

It is recommended that applications use a threaded development model with synchronous event notifications rather than the various asynchronous communication mechanisms, including SIGNALS, provided by POSIX.

3.2.1 Configuration Management

The SPM shall make direct use of the following POSIX functions for determining and managing the run-time configuration of the POSIX components, including which POSIX options are supported in a given implementation:

sysconf()	Reports the state of various internal limits and configuration options, including whether or not particular optional POSIX functionality is supported in the current implementation. Non-standard extensions can be added to the reporting from this function to allow for application- or system-dependent extensions to be added.
pathconf(),fpathconf()	Reports various configurable path names within the current system.

The following sysconf() parameter values must indicate the POSIX feature is supported for all SPM implementations.

_SC_THREADS	Implementation supports threads
_SC_THREAD_PRIORITY_SCHEDULING	Implementation supports priority scheduling of threads
_SC_TIMERS	Implementation supports timers
_SC_SEMAPHORES	Implementation supports semaphores
_SC_SHARED_MEMORY_OBJECTS	Implementation supports shared memory
_SC_MESSAGE_PASSING	Implementation supports message passing

_SC_REALTIME_SIGNALS	Implementation supports real-time signaling

The following *sysconf()* parameter values must indicate the POSIX feature is supported if the SPM implementation supports multiple processes.

_SC_PRIORITY_SCHEDULING	Implementation supports priority scheduling of processes

The following sysconf() parameter values must be available for SPM configuration and operation.

_SC_CLK_TCK	System clock ticks/second
_SC_THREAD_THREADS_MAX	Maximum # of threads per process
_SC_TIMER_MAX	Maximum # of timers per process
_SC_SEM_VALUE_MAX	Maximum value a semaphore may have
_SC_SEM_NSEMS_MAX	Maximum # of semaphores per process
_SC_SIGQUEUE_MAX	Maximum # of queued signals
_SC_RTSIG_MAX	Maximum # of real-time signals
_SC_MQ_OPEN_MAX	Maximum # of open message queues

3.2.2 Process Management

The SPM shall provide for a single process operating on the system. Basic process support shall be provided by the POSIX_SINGLE_PROCESS option. Since, by definition, the single process represents the startup context for the application no specific functional support for process management is required for single-process operation.

Desired: The SPM shall provide support for multiple processes operating on the system.

Guidance: Processes are managed by use of a “Process ID”, which is an anonymous handle that identifies the Process within the context of the system.

Guidance: POSIX manages Processes as a hierarchy, with “Parent” processes having the ability to start, monitor, and stop “Child” processes as necessary for the application. Processes exist in their own virtual machine, with barriers to prevent them interfering with each other and with accesses to hardware resources mediated by the underlying operating system. Namespaces and resource handles are, in general, private to each process, although there are mechanisms to allow Parent processes to share resource handles with their Children.

Guidance: See section 3.2.3 for a discussion of scheduling policies and the associated requirements for thread design.

If The SPM provides support for multiple processes, it shall incorporate the optional component option ‘POSIX_MULTI_PROCESS’ and provide the following functions:

posix_spawn(),posix_spawnp()	Create a new Process from a file or ‘C’ function reference.
fork()/vfork()	Create a new Process that is a copy of the calling Process
exit()	Terminate the current Process
atexit()	Register a function to be executed when a Process terminates through a call to exit().
wait()/waitpid()	Wait for a child Process to terminate and return its status information
getpid()	Report the Process ID for the current Process
sched_setscheduler()/sched_getscheduler()	Set/get the scheduler policy for the current Process
sched_setparam()/sched_getparam()	Set/get the parameters to configure the scheduler policy for the current Process
times()	Get the elapsed time since startup for the Process

Usage Recommendations:

It is recommended that systems with real-time response requirements utilize a single application Process supporting multiple Threads, rather than Threads assigned to different Processes, to simplify analyses of thread determinism and to minimize process context switching overhead.

3.2.3 Thread Management

The SPM shall provide support for multiple threads operating within a process. Basic thread support is provided by the optional POSIX Threads and Realtime modules.

Guidance: The POSIX process/thread scheduler supports at least 3 scheduling policies, designated FIFO, Round-Robin, and Sporadic. The details of the policies may be found in the POSIX specification; in general all policies allow the highest-priority thread to run in preference to lower priority threads but differ in how they manage multiple threads at the same priority. The FIFO policy selects a thread from the pool of threads at the same priority based strictly on the order in which they became ready to run, and each thread is allowed to run until it suspends itself for some reason; the Round-Robin policy allocates each thread at the same priority a time-slice to ensure no thread can monopolize the processor; the Sporadic policy also implements a time-slice, but adjusts thread priorities up and down within some narrow range to balance the execution time among threads. Each Process/Thread is assigned a scheduling policy and the configuration parameters for that policy independently, such that there can be different threads or groups of threads operating with different policies.

Guidance: If more than one thread priority is used then all threads must be designed to share the processor at some point within their control loop, either explicitly or by waiting for a communication or synchronization object, regardless of the scheduler policy selected. If a thread does not share the processor then no lower-priority threads will be allowed to execute even if a time-slicing scheduler policy is selected.

At a minimum the SPM shall provide for thread operation using the SCHED_FIFO scheduling policy; it is desired, but not required, that the SPM also support the SCHED_RR (round-robin) scheduling policy.

Guidance: Resources for a POSIX Thread are allocated automatically when the Thread is created, but release of the resources depends on how the Thread is to be terminated. If a Thread is to be managed by another Thread, and must therefore return status to the managing Thread after termination, then it is said to be "joinable" and resources are released by the managing Thread using a call to 'pthread_join()', which extracts information about the managed Thread's termination status before releasing the resources. Since the "joinable" Thread does not release its resources when it terminates, the "joining" operation does not need to happen before that point; the managing Thread can "join" the managed Thread after managed Thread termination. If a Thread operates independently and does not need to report status to some other Thread it is said to be 'detached' and will release its own resources upon termination without providing status information externally. A detached Thread may not be "joined".

Guidance: POSIX provides a facility for installing ‘cleanup’ handlers which will be executed automatically when a Thread terminates. Cleanup handlers are executed on a last-in/first-out basis; that is, the most recently installed handler is executed first.

Guidance: Threads are managed by use of a “Thread ID”, which is an anonymous handle that identifies the Thread within the context of its Process.

Guidance: For purposes of scheduling, threads exist with a scheduling “scope” which defines the set of peers against which thread prioritization is managed. For ‘PROCESS’ scope the threads compete in priority order with all other threads within their Process but not with threads from other processes; resolution of scheduling conflicts across processes is determined by the process scheduler and the relative process priorities. Hence, a high priority thread in one process may be outprioritized by a low priority thread from another process with a higher process priority. For ‘SYSTEM’ scope the threads compete in priority order with all other threads from all processes, such that priority ordering at the thread level is preserved.

At a minimum the SPM shall support PTHREAD_SCOPE_PROCESS scope for thread scheduling, and shall support thread operation by direct incorporation of the following minimum set of functions:

pthread_create()	Create a Thread
pthread_exit()	Terminate the current Thread
pthread_join()	Suspend the current Thread until the specified Thread has terminated; Thread resources are released upon Thread termination as a result of the call to pthread_join().
pthread_detach()	Configure the specified Thread to release its resources upon termination.
pthread_cancel()	Request termination of the specified Thread. Termination will be deferred until the Thread reaches a ‘cancellation point’.
pthread_testcancel()	Create a Thread ‘cancellation point’.
pthread_setcancelstate()/pthread_setcanceltype()	Configure the cancelability state (ENABLE or DISABLE) and/or cancelability type (DEFERRED or

	ASYNCHRONOUS) for the current Thread.
pthread_once()	Execute a specified initialization routine exactly once within the current process. After the first execution subsequent calls will result in noops.
pthread_self()	Report the thread ID for the current Thread.
sched_get_priority_min()/sched_get_priority_max()	Report the minimum/maximum valid scheduling priorities.
pthread_setschedprio()/pthread_getschedprio()	Set/get the priority for the current Thread
pthread_setschedparam()/pthread_getschedparam()	Set/get the configurable schedule policy parameters for the current Thread
sched_yield()	Suspend the current Thread to allow other Threads at the same priority an opportunity to run.
pthread_cleanup_push()/pthread_cleanup_pop()	Install/uninstall cleanup handlers to be executed when a Thread exits due to cancellation or self-termination.
pthread_attr_init()/pthread_attr_destroy()	Initialize/release a Thread attribute object
pthread_attr_setstacksize()/pthread_attr_getstacksize()	Set/get the stack size for a Thread in the Thread attribute object
pthread_attr_setdetachstate()/pthread_attr_getdetachstate()	Set/get the detach state (DETACHED or JOINABLE) for a Thread in the Thread attribute object
pthread_attr_setschedpolicy()/pthread_attr_getschedpolicy()	Set/get the scheduler policy for a Thread in the Thread attribute object
pthread_attr_setschedparam()/pthread_attr_getschedparam()	

Set/get the scheduler policy configuration parameters for a Thread in the Thread attribute object

Guidance: In the interest of simplicity and portability some of the more advanced and esoteric features of POSIX Threads have been omitted from this list of minimum functionality.

Guidance: Thread configuration, including stack size and priority, is specified by the 'attribute' argument passed during the call to pthread_create(). The attributes are configured using the pthread_attr_xxxx() family of functions. Note that POSIX specifically defines attributes using an anonymous object handle and accessor functions, rather than through a structure, to minimize maintenance issues when features are added to POSIX through the attribute object.

Guidance: The designers of POSIX opted to manage Thread attributes through accessor functions acting on anonymous handles to attribute objects rather than directly through a public data structure. The reason for that choice was to make it easy to provide backward compatibility between applications designed for older POSIX versions and new POSIX releases that extend the attribute functionality – the older applications can be linked directly to the newer POSIX library without recompilation and/or data inconsistencies. However, that choice makes the configuration of threads with non-default attributes awkward.

To simplify creation of threads with common attribute configurations the SPM shall provide the following extension to the standard POSIX implementation:

```
int pthread_spm_create ( pthread_t *restrict pThreadId,
                         int nPriority,
                         unsigned int uStackSize,
                         int bDetached,
                         void *(*)(void *) pfnThreadFunction,
                         void *restrict pContext );
```

Guidance: Implementation of the simplified 'create' function will encapsulate initialization of the Thread attribute object using the function arguments, and use that Thread attribute object to create the Thread using the standard POSIX Thread creation function. The scheduler policy for the Thread shall be set automatically to SCHED_FIFO.

The SPM shall provide the following extensions to the standard POSIX implementation to accommodate underlying operating systems on which the thread subsystem must be initialized and/or shut down by the application:

```
int pthread_spm_init ( void );
int pthread_spm_cleanup ( void );
```

Guidance: For implementations that do not require explicit thread subsystem initialization/cleanup those functions will be noops.

Usage Recommendations:

To maximize portability, it is recommended that processes initialize the thread library at startup with a call to 'pthread_spm_init()' and clean up the thread library at shutdown with a call to 'pthread_spm_cleanup()', regardless of whether the implementation specifically requires it.

It is recommended that if a system with real-time response requirements utilizes multiple application processes then all real-time threads operate at 'SYSTEM' scope. That maintains the determinism of thread scheduling across processes.

It is recommended that the cleanup handler facility (pthread_cleanup_push()) be used to release resources and quiesce hardware used by the thread function. That ensures that resources will be released and hardware will be left in a quiescent state even if the thread exits due to an external 'cancel' request.

It is recommended that threaded applications be designed to operate using the 'SCHED_FIFO' scheduling policy and not to depend on the time-slicer to ensure threads are allotted adequate execution time. This minimizes context switching overhead since threads at equal priority will generally yield the processor once for each pass through their loop rather than at multiple arbitrary points determined by the system time-slice. In general, if this results in a thread 'hogging' the processor, that is either because it is wasting time polling something when it could be suspended, because it is mis-prioritized, or because it represents a compute-intensive task that is overloading the available processing resources.

3.2.4 Resource Interlocks

The SPM shall make direct use of the following POSIX resource interlock entities and functions to manage access to resources:

Desired: Spin-Lock (Advanced Real-Time Threads): A spin-lock is the lightest-weight interlock; it allows only a single thread to have access to the protected resource at a time. The requesting thread polls the spin-lock until it is available. On a multi-processor system this can eliminate a task switch (compared with a mutex, defined below) if the lock is held for only a short interval by a thread running on another processor; however, time spent polling the spin-lock is effectively lost as usable processor time. On a single-processor system, or if the thread holding the lock is executing on the same processor as the requesting thread, the lock cannot be released without intervention of a task switch, so the spin-lock provides no advantage over a mutex (see below) and has the potential to create a deadlock if time-slicing is not enabled or if the thread currently holding the SpinLock is at a lower priority than the one requesting it.

The SPM shall provide support for Thread-level spinlocks that are accessible within a single process.

Desired: the SPM shall provide support for Process-level spinlocks that are accessible across processes.

Guidance: POSIX provides cross-process functionality for spinlocks only if the application has created the spinlock with the 'pshared' argument set to TRUE and allocated memory for the spinlock within a shared memory space.

Spinlocks shall be accessed using the following functions:

pthread_spin_init()	Create a spin-lock
pthread_spin_destroy()	Destroy a spin-lock
pthread_spin_lock()	Request access to a spin-lock; continue polling until the lock is available.
pthread_spin_trylock()	Request access to a spin-lock; return immediately, with or without the lock.
pthread_spin_unlock()	Release access to a spin-lock.

Mutex (Threads): A mutex (mutual-exclusion lock) allows only a single thread to have access to the protected resource at a time. The requesting thread suspends (relinquishes the processor) until the lock is available.

The SPM shall provide support for Thread-level mutexes that are accessible within a single process.

Desired: The SPM shall provide support for Process-level mutexes that are accessible across processes.

Guidance: POSIX provides cross-process functionality for mutexes only if the application has created the mutex with the ‘PTHREAD_PROCESS_SHARED’ attribute and allocated memory for the mutex within a shared memory space.

Guidance: POSIX mutexes support priority boost to prevent a priority inversion. There are three boost protocols available: ‘NONE’ (no priority boost); ‘INHERIT’ (inherit the priority from the highest priority waiting thread); and ‘PROTECT’ (set the priority based on the specified priority ceiling).

At a minimum, the SPM shall support either the ‘INHERIT’ or ‘PROTECT’ ‘protocol’ attribute for mutexes.

Mutexes shall be accessed using the following functions:

pthread_mutex_init()	Create a mutex
pthread_mutex_destroy()	Destroy a mutex
pthread_mutex_lock()	Request access to a mutex; suspends until the lock is available.
pthread_mutex_timedlock()	Requests access to a mutex; suspends until the lock is available or the timeout expires, whichever happens first.
pthread_mutex_trylock()	Request access to a mutex; return immediately, with or without the mutex.
pthread_mutex_unlock()	Release access to a mutex.
pthread_mutexattr_init()/pthread_mutexattr_destroy()	Initialize/release a mutex attribute object.
pthread_mutexattr_settype()/pthread_mutexattr_gettype()	Set/get the mutex type in the mutex attribute object.
pthread_mutexattr_setprioceiling()/pthread_mutexattr_getprioceiling()	

Set/get the mutex priority ceiling in the mutex attribute object.

pthread_mutexattr_setprotocol()/pthread_mutexattr_getprotocol()

Set/get the mutex priority boost protocol in the mutex attribute object.

pthread_mutexattr_setpshared()/pthread_mutexattr_getpshared()

Set/get cross-process sharing flag in the mutex attribute object.

Guidance: The designers of POSIX opted to manage mutex attributes through accessor functions acting on anonymous handles to attribute objects rather than directly through a public data structure. The reason for that choice was to make it easy to provide backward compatibility between applications designed for older POSIX versions and new POSIX releases that extend the attribute functionality – the older applications can be linked directly to the newer POSIX library without recompilation and/or data inconsistencies. However, that choice makes the configuration of mutexes with non-default attributes awkward.

To simplify creation of mutexes with common attribute configurations the SPM shall provide the following extension to the standard POSIX implementation:

```
int pthread_spm_mutex_init ( pthread_mutex_t *restrict pMutex,
                             int bCrossProcess );
```

Guidance: Implementation of the simplified 'init' function will encapsulate initialization of the mutex attribute object using the function arguments, and use that mutex attribute object to create the mutex using the standard POSIX mutex creation function. The priority boost protocol for the mutex shall be set to 'PTHREAD_PRIO_INHERIT', indicating that the boost priority shall be inherited from the highest-priority thread pending on the mutex; since the 'INHERIT' protocol does not use the priority ceiling, the priority ceiling will be set to the system default value.

Usage Recommendations:

It is recommended that mutexes encapsulated within library functions – and hence beyond the direct control of the application programmer – be created either with the 'type' attribute RECURSIVE' or with the 'type' attribute 'ERRORCHECK' to protect against deadlocks due to re-locking by the same thread.

It is recommended that mutexes used in constructing access control hierarchies (e.g. Device-Level -> Channel-Level -> Register-Level) be created with the 'type' attribute 'RECURSIVE' since such hierarchies are prone to experiencing recursive lock requests.

Per the POSIX documentation it is recommended that mutexes used in conjunction with Condition Variables (section 4.5, Appendix A) not be created with 'type' attribute 'RECURSIVE' since the internal structure of the Condition Variable does not guarantee a single unlock request for each lock request.

Neither POSIX nor the SPM define a specific, global 'Critical Section' locking mechanism. It is recommended that applications use mutexes to create resource-specific 'Critical Section' locks as necessary.

Read/Write Lock (Threads): A read/write lock allows multiple threads at once to 'read' an object – that is, to access it in a fashion that does not change its state – but only a single thread at once to 'write' an object – that is, to access it in a fashion that changes its state. A thread may get 'read' access to the object as long as no thread has 'write' access; a thread may get 'write' access when no thread has either 'read' or 'write' access. The requesting thread suspends (relinquishes the processor) until the lock ('read' or 'write') is available.

The SPM shall provide support for Thread-level read/write locks that are accessible within a single process.

Desired: The SPM shall provide support for Process-level read/write locks that are accessible across processes.

Guidance: POSIX provides cross-process functionality for read/write locks only if the application has created the read/write lock with the 'PTHREAD_PROCESS_SHARED' attribute and allocated memory for the read/write lock within a shared memory space.

pthread_rwlock_init()	Create a read/write lock
pthread_rwlock_destroy()	Destroy a read/write lock
pthread_rwlock_rdlock()	Request 'read' access to a read/write lock; suspends until the lock is available.
pthread_rwlock_timedrdlock()	Request 'read' access to a read/write lock; suspends until the lock is available or the timeout expires, whichever happens first.

pthread_rwlock_tryrdlock()	Request 'read' access to a read/write lock; return immediately, with or without the lock.
pthread_rwlock_wrlock()	Request 'write' access to a read/write lock; suspends until the lock is available.
pthread_rwlock_timedwrlock()	Request 'write' access to a read/write lock; suspends until the lock is available or the timeout expires, whichever happens first.
pthread_rwlock_tryrdlock()	Request 'write' access to a read/write lock; return immediately, with or without the lock.
pthread_rwlock_unlock()	Release access to a read/write lock.
pthread_rwlockattr_init()/pthread_rwlockattr_destroy()	Initialize/release a read/write lock attribute object.
pthread_rwlockattr_setpshared()/pthread_rwlockattr_getpshared()	Set/get cross-process sharing flag in the read/write lock attribute object.

Guidance: The designers of POSIX opted to manage read/write lock attributes through accessor functions acting on anonymous handles to attribute objects rather than directly through a public data structure. The reason for that choice was to make it easy to provide backward compatibility between applications designed for older POSIX versions and new POSIX releases that extend the attribute functionality – the older applications can be linked directly to the newer POSIX library without recompilation and/or data inconsistencies. However, that choice makes the configuration of read/write locks with non-default attributes awkward.

To simplify creation of read/write locks with common attribute configurations the SPM shall provide the following extension to the standard POSIX implementation:

```
int pthread_spm_rwlock_init ( pthread_rwlock_t *restrict pLock,
                              int bCrossProcess );
```

Guidance: Implementation of the simplified 'init' function will encapsulate initialization of the read/write lock attribute object using the function arguments, and use that read/write lock attribute object to create the read/write lock using the standard POSIX read/write lock creation function.

Usage Recommendations:

No specific recommendations are provided for the use of read/write locks.

Semaphore (Realtime): A semaphore allows some fixed number of threads at once access to some resource, or to a pool of resources. The requesting thread suspends (relinquishes the processor) until the semaphore is available. A semaphore is available when its associated count is non-zero, and unavailable when its associated count is zero; gaining access to the semaphore decrements the count, while releasing access increments the count. A semaphore is a process-level object that resolves resource conflicts across processes. Hence a semaphore may be created with a globally-unique name and accessed using that name from any process.

Guidance: A semaphore initialized to a count of 1 is functionally equivalent to a mutex and may be used for that purpose, with the exception that it does not support priority boost to protect against priority inversions.

The SPM shall provide support for Thread-level semaphores that are accessible within a single process.

Desired: The SPM shall provide support for Process-level semaphores that are accessible across processes.

Guidance: POSIX provides cross-process functionality for semaphores only if the application has created the semaphore with the 'pshared' argument set to TRUE and allocated memory for the semaphore within a shared memory space.

sem_init()	Create an unnamed semaphore
sem_destroy()	Destroy an unnamed semaphore
sem_open()	Create a named semaphore or connect to a previously-created named semaphore
sem_close()	Disconnect from a named semaphore and release its resources.
sem_unlink()	Disconnect from a named semaphore, but keep its resources allocated and preserve its state.
sem_wait()	Request access to a semaphore; suspends until the semaphore is available.

sem_timedwait()	Request access to a semaphore; suspends until the semaphore is available or the timeout expires, whichever happens first.
sem_trywait()	Request access to a semaphore; return immediately, with or without the semaphore.
sem_post()	Release access to a semaphore lock.
sem_getvalue()	Reports the current semaphore count.

Usage Recommendations:

It is recommended that mutexes be used in preference to semaphores for simple mutual-exclusion functionality, since mutexes offer a priority boost to prevent priority inversions.

3.2.5 Synchronization

The SPM shall make direct use of the following POSIX synchronization entities and functions to manage thread and I/O synchronization:

Guidance: Timer (REALTIME): POSIX supports timers that are accessible to all threads within a single process. Timers are associated with a specific system clock, identified by a clock ID defined in the header <time.h>.

Guidance: POSIX provides a system-wide clock that is guaranteed to be monotonic (cannot be set by the application, and therefore will never jump backward in time) designated by the clock ID CLOCK_MONOTONIC. POSIX implementations supporting the REALTIME extensions must also provide at least one clock, designated CLOCK_REALTIME, used by the REALTIME components. Other clocks may be provided for a given implementation, but are not mandated for POSIX conformance.

Guidance: The signaling mechanism associated with a timer is an asynchronous Signal (see below) configured with a callback function, rather than one of the thread-aware synchronization objects. For that reason, this guideline defines an Extended Timer function (see below) which uses thread-aware synchronization objects for notification. The standard POSIX Timer functions are described here because they must be present to support the Extended Timer.

The SPM shall permit use of standard POSIX Timers through the following functions:

sleep()	Suspends the current thread for the specified number of seconds; depending on the implementation, may be incompatible with the use of the POSIX alarm() function.
nanosleep()	Suspends the current thread for the specified number of nanoseconds. Uses CLOCK_REALTIME as its timing source, and its resolution is therefore limited to the resolution provided by that clock, which is in all cases much larger than the nanosecond resolution specified for the delay.
clock_getres()	Reports the timing resolution of a specified source clock
timer_create()	Create a timer
timer_destroy()	Destroy a timer
timer_settime()	Configure a timer
timer_gettime()	Report the time remaining on a timer until the next timeout.
timer_getoverruns()	Report the total number of timer timeouts that have occurred since the last execution of the timeout handler.

Guidance: Some implementations use the POSIX alarm() function as the basis for sleep() and in those implementations the two functions may conflict with each other. For that reason the SPM does not assume the alarm() function is available for use and does not incorporate it as part of the standard SPM API.

Usage Recommendations:

It is recommended that applications use the Extended Timer functions defined below in preference to the raw POSIX timer functions.

It is recommended that applications use timer functions, rather than the POSIX alarm() function, to activate threads at specific times or on a timed interval.

It is recommended that timers be configured to use CLOCK_REALTIME as the clock ID to make them consistent with timing functions in other components from the REALTIME subsystem.

Signals (REALTIME): A Signal is an asynchronous signaling mechanism that typically utilizes a callback function to receive notification of an event. The SPM shall provide support for POSIX signals using the following functions:

sigaction()	Configure the action to take in response to a specified Signal
sigwait()	Wait for a specified Signal and handle it; suspends until the Signal is received. Returns only the Signal number.
sigwaitinfo()	Wait for a specified Signal and handle it; suspends until the Signal is received. Returns Signal information defined by the real-time Signal extensions.
sigtimedwait()	Wait for a specified Signal and handle it; suspends until the Signal is received or the timeout expires, whichever happens first. Returns Signal information defined by the real-time Signal extensions.
sigsuspend()	Wait for a specified Signal to be handled by an installed callback function; suspends until the Signal has been received and handled.
sigpending()	Report all the Signals currently pending for delivery to the current Thread.
sigqueue()	Queue a Signal for delivery to a specified Process; allows passing of a datum with the Signal.
kill()	Queue a Signal for delivery to a specified Process; no datum accompanies the Signal.
pthread_kill()	Queue a Signal for delivery to a specified Thread; no datum accompanies the Signal.
raise()	Queue a Signal for delivery to the current Thread; no datum accompanies the Signal.

sigprocmask()	Set and/or get the current Process Signal mask
pthread_sigmask()	Set and/or get the current Thread Signal mask

Guidance: The mechanisms by which threads interact with signals are different than the mechanisms by which they interact with other synchronization objects like mutexes and semaphores. Ordinarily, those other mechanisms are preferred but some POSIX objects -- specifically timers, asynchronous I/O channels, and sockets – use Signals as their notification mechanism so support for Signals is included as a requirement for the SPM.

Usage Recommendations:

It is recommended that applications use other synchronization mechanisms in lieu of signals wherever possible.

Condition (THREADS): A Condition, or Condition Variable, is a signaling mechanism used to indicate that some logical state has occurred. Since the 'condition' being signaled is represented by some logical test on application data, it is possible to end up with a race condition if one thread is waiting for the condition to be true at the same time as another thread is changing the condition. To prevent that, a Condition Variable is always associated with a Mutex that is used to lock access to the data while it is being manipulated. The Condition Variable itself merely provides a channel for asynchronous notification that some event has occurred; the Mutex provides thread-safe access to an associated data item, and the application code must assess the data to determine whether the condition does or does not exist.

pthread_cond_init()	Create a Condition Variable
pthread_cond_destroy()	Destroy a Condition Variable
pthread_cond_wait()	Wait on a Condition Variable; suspends until the condition is signaled.
pthread_cond_timedwait()	Wait on a Condition Variable; suspends until the condition is signaled or the timeout expires, whichever happens first.
pthread_cond_signal()	Signal that a condition is true on a Condition Variable; used to signal the condition to a single waiting thread.
pthread_cond_broadcast()	Signal that a condition is true on a Condition Variable; used to signal the condition to all waiting threads.

pthread_condattr_init()/pthread_condattr_destroy()

Initialize/release a condition variable attribute object.

pthread_condattr_setpshared()/pthread_condattr_getpshared()

Set/get cross-process sharing flag in the condition variable attribute object.

Guidance: For condition variables POSIX allows for configuration of the clock used to time waits (pthread_cond_timedwait()). Since the default clock ('system clock') is the same as that used for timing waits on other synchronization objects, we do not require that the SPM support use of that configuration attribute.

The designers of POSIX opted to manage condition variable attributes through accessor functions acting on anonymous handles to attribute objects rather than directly through a public data structure. The reason for that choice was to make it easy to provide backward compatibility between applications designed for older POSIX versions and new POSIX releases that extend the attribute functionality – the older applications can be linked directly to the newer POSIX library without recompilation and/or data inconsistencies. However, that choice makes the configuration of condition variables with non-default attributes awkward.

To simplify creation of condition variables with common attribute configurations the SPM shall provide the following extension to the standard POSIX implementation:

```
int pthread_spm_cond_init ( pthread_rwlock_t *restrict pLock,
                            int bCrossProcess );
```

Guidance: Implementation of the simplified 'init' function will encapsulate initialization of the read/write lock attribute object using the function arguments, and use that read/write lock attribute object to create the read/write lock using the standard POSIX read/write lock creation function.

Guidance: In addition to requiring creation of a mutex for use with a condition variable, the POSIX condition variable API also places requirements on the application with respect to when the application must lock/unlock the mutex.

To simplify the creation and use of condition variables, and because condition variables are always used in conjunction with a mutex, the SPM shall provide an extension to the POSIX implementation that defines an SPM-specific version of a condition variable (SPM Condition Variable) which encapsulates the mutex and the condition variable into a single object. The following functions shall be provided to manage SPM-specific condition variables:

pthread_spm_cond_init()	Create a Condition Variable
pthread_spm_cond_destroy()	Destroy a Condition Variable
pthread_spm_cond_wait()	Wait on a Condition Variable; suspends until the condition is signaled.
pthread_spm_cond_timedwait()	Wait on a Condition Variable; suspends until the condition is signaled or the timeout expires, whichever happens first.
pthread_spm_cond_lock()/pthread_spm_cond_timedlock()/ pthread_spm_cond_trylock()/pthread_spm_cond_unlock()	Lock/unlock the Mutex associated with the Condition Variable; used to gain exclusive access to the data managed by the condition variable.
pthread_spm_cond_signal()	Signal that a condition is true on a Condition Variable; used to signal the condition to a single waiting thread.
pthread_spm_cond_broadcast()	Signal that a condition is true on a Condition Variable; used to signal the condition to all waiting threads.

Usage Recommendations:

It is recommended that applications use the SPM-specific version of the condition variable in preference to the standard POSIX version.

The SPM shall provide extended functionality to manage thread and I/O synchronization as follows:

Extended Timer: The SPM shall provide extended functionality to support a configurable timer that utilizes the Event mechanism (defined below), rather than a Signal, as its timeout notification method.

Guidance: The function names and forms for the Extended Timer API are defined specifically to maintain consistency with the names and forms provided for standard POSIX timer components.

spm_timer_create()	Create a timer
spm_timer_destroy()	Destroy a timer
spm_timer_settime()	Configure a timer
spm_timer_gettime()	Report the time remaining on a timer until the next timeout.
spm_timer_getoverruns()	Report the total number of timer timeouts that have occurred since the last execution of the timeout handler.

Guidance: To maintain full POSIX conformance it is recommended that the Extended Timer functions be implemented using the standard POSIX Timer functionality. This will likely require creation of a background timer thread to catch signals from the POSIX timer.

Usage Recommendations:

It is recommended that applications use the Extended Timer rather than the standard POSIX timer for thread synchronization.

Flag Group: The SPM shall provide extended functionality to support Flags. A Flag is a Boolean value, represented by a single bit within an integer variable, called the Flag Group; hence multiple Flags may be represented using the same underlying Flag Group, and Flag functions can act on multiple Flags at once provided they reside within the same Flag Group. A thread may suspend while waiting for Flag(s) to be in a specified state; other threads thread may set the Flag(s) to the specified state, thereby causing the pending thread to be alerted.

Guidance: The function names and forms for the Flag Group API are defined specifically to maintain consistency with the names and forms provided for standard POSIX synchronization/communication components.

Guidance: To maintain full POSIX conformance it is recommended that Flags be implemented using a standard POSIX Condition Variable managing the Flag Group.

pthread_spm_flag_init()	Create a Flag Group
pthread_spm_flag_destroy()	Destroy a Flag Group
pthread_spm_flag_wait()	Wait for a specified Flag state or pattern of Flag states; suspends until the condition is satisfied.
pthread_spm_flag_timedwait()	Wait for a specified Flag state or pattern of Flag states; suspends until the condition is satisfied or the timeout expires, whichever happens first.
pthread_spm_flag_set()	Set a Flag state or pattern of Flag states; results in pending threads being signaled if the new state matches the requested pattern.

Usage Recommendations:

It is recommended that a Flag Group containing a single flag bit be used to provide functionality equivalent to the 'event' element provided in some real-time environments.

3.2.6 Inter-Thread Communication

The SPM shall make direct use of the following POSIX inter-thread communication entities and functions to manage thread and I/O communication:

Shared Memory: *Shared Memory* is a memory region that is accessible across Processes. Since Process address spaces are virtual, memory references are normally valid only within the Process in which they were allocated. Shared Memory regions may be mapped into the private address spaces of multiple processes; once they have been mapped they may be accessed by pointer in the same way any other memory region would be. To facilitate mapping them into different Process address spaces, Shared Memory regions are managed as if they were files, identified by a system-unique name and accessed for purposes of configuration via a file handle and using a subset of the POSIX file manipulation functions. Like files, once they are created named Shared Memory regions will persist until they are 'unlinked' even if no processes are currently mapped to them.

Guidance: POSIX does not require shared memory regions to persist across system restarts.

Guidance: Per standard POSIX usage, synchronization and interlocking objects, like mutexes and condition variables, can be made accessible across Processes by placing them into shared memory blocks.

The SPM shall provide support for POSIX Shared Memory using the following functions:

shm_open()	Create/connect to a Shared Memory region
close()	Close a connection to a Shared Memory region; this is a standard file manipulation function.
shm_unlink()	Destroy a Shared Memory region and make its name available for assignment to a new Shared Memory region.
ftruncate()	Set the size of a Shared Memory region; this is a standard file manipulation function.
mmap()	Map a Shared Memory region into the address space of the current Process
munmap()	Remove a Shared Memory region from the address space of the current Process.

To simplify creation of Shared Memory regions, the SPM shall provide the following extension to the standard POSIX functionality:

```
int spm_shm_open ( const char strName[], int oflag, mode_t mode, off_t length );
```

Guidance: The extended ‘open’ function combines the creation/connection to the Shared Memory region and the sizing of the region into a single function call.

Usage Recommendations:

No specific recommendations are provided for the use of Shared Memory.

Message Queue: A POSIX Message Queue is a channel through which messages can flow between Threads and Processes. The maximum message size and the maximum number of messages that can be queued into the channel are configured when the Message Queue is created; individual messages may be of different sizes provided no message exceeds the maximum message size. Messages within the Message Queue are prioritized, and messages are always delivered to the far end of the channel in priority order; within each priority messages are delivered on a first-in/first-out basis. To facilitate mapping them into different Process address spaces, Message Queues are identified and managed in a manner similar to that used for files, identified by a system-unique name.

Guidance: Message queues were designed for inter-process communication, with all the overhead that implies. In some implementations Message Queues may be implemented using mapped files or other file-system constructs, which imposes the same upper bound on the number of open Message Queues are there is on the number of open files.

The SPM shall provide support for POSIX Message Queues using the following functions:

mq_open()	Connect to a Message Queue
mq_close()	Disconnect from a Message Queue.
mq_unlink()	Destroy a Message Queue and make its name available for assignment to a Message Queue.
mq_receive()	Receive a message from a MessageQueue; suspends until a message is available.
mq_timedreceive()	Receive a message from a MessageQueue; suspends until a message is available or the timeout expires, whichever happens first.
mq_send()	Place a message into a Message Queue; suspends until there is room for the message in the channel.
mq_timedsend()	Place a message into a Message Queue; suspends until there is room for the message in the channel or the timeout expires, whichever happens first.
mq_setattr()/mq_getattr()	Set/get attributes of the Message Channel.

Usage Recommendations:

Guidance: It is recommended that Data Queues (defined below) be used in preference to Message Queues if the functionality required is the transfer of fixed-size elements through a simple FIFO.

The SPM shall provide extended functionality to facilitate inter-thread communication as follows:

Data Queue: A Data Queue is a thread-aware FIFO that can be used to pass data between Threads. The size of an individual element and the number of elements in the Data Queue FIFO is configured at initialization and remains constant for the lifetime of the Data Queue.

Guidance: Data Queues are somewhat simpler than POSIX Message Queues, inasmuch as they operate as a traditional FIFO: they are constrained to operate with a fixed element size, do not discriminate queued elements by priority, and operate entirely within a memory block without kernel interaction. Hence, they are designed to be extremely efficient for inter-thread communication within a Process. Internally, Data Queues should use a Mutex (as necessary) to protect access to the Data Queue object and a Condition Variable to send notifications to any pending threads when the internal FIFO becomes empty or full.

The SPM shall provide support for Thread-level Data Queues that are accessible within a single process.

Desired: The SPM shall provide support for Process-level Data Queues that are accessible across processes.

Guidance: As with other POSIX thread synchronization/communication objects, cross-process functionality for Data Queues is provided only if the application has created the Data Queue with the ‘bCrossProcess’ argument set to TRUE and allocated memory for the Data Queue within a shared memory space. NOTE that any synchronization/interlock objects (like mutexes) embedded within the Data Queue must also have been created for use across processes.

Guidance: Because the amount of memory required to hold a Data Queue depends on the element size and the number of elements, the size of a Data Queue object in memory is not fixed and must be determined using the ‘spm_dq_objectbytes()’ macro. To allow its use for initialization, that macro must be defined in such a way that it resolves to a valid constant at the time of compilation.

Guidance: The function names and forms for the Data Queue API are defined specifically to maintain consistency with the names and forms provided for standard POSIX synchronization/communication components.

spm_dq_objectbytes()	Report the size of the memory block required to hold a Data Queue object.
spm_dq_init()	Create a Data Queue
spm_dq_destroy()	Release a Data Queue.
spm_dq_elementbytes()/spm_dq_elements()	Report the element size and the number of elements in the Data Queue FIFO.
spm_dq_isfull()/spm_dq_isempty()/spm_dq_entries()	Report the number of entries currently in the Data Queue FIFO, either directly (‘entries’) or indirectly (‘empty’ or ‘full’).
spm_dq_waitwhileempty()/spm_dq_waitwhilefull()/ spm_dq_timedwaitwhileempty()/spm_dq_timedwaitwhilefull()	Suspend the current thread while the Data Queue is empty or full.
spm_dq_reset()	Reset the Data Queue; removes all entries.
spm_dq_dequeue()/spm_dq_timeddequeue()	Extract an entry from the head of the Data Queue FIFO.
spm_dq_enqueue()/spm_dq_timedenqueue()	Insert an entry at the tail of the Data Queue FIFO.
spm_dq_requeue()	Insert an entry at the head of the Data Queue FIFO.
spm_dq_read()	Extract multiple entries from the head end of the Data Queue FIFO.
spm_dq_write()	Insert multiple entries at the tail end of the Data Queue FIFO.

spm_dq_headofqueue()	Report the entry at the head of the Data Queue FIFO without removing it.
spm_dq_examine()	Report multiple entries from the head end of the Data Queue FIFO without removing them.

Usage Recommendations:

It is recommended that a Data Queue of length 1 be used to provide functionality equivalent to the ‘mailbox’ provided in some real-time environments.

3.2.7 I/O Support

I/O support is beyond the specific scope of this guideline and is the subject of a separate guideline describing a “Standard Device Model” (SDM). To the extent that common support for I/O operations within the POSIX environment is properly integrated with the thread- and process-management mechanisms, applications using this Standard Process Model may also use the I/O support mechanisms provided within the POSIX environment. However, in general, it is recommended that I/O accesses conform to the SDM guideline.

Usage Recommendations:

Guidance: In a threaded development model different threads, operating concurrently, execute different aspects of an application, and I/O operations may be isolated within threads that are not responsible for other aspects of system operation. Hence, in general, within a threaded development model there is very little need or justification to use an asynchronous I/O model in which I/O requests return immediately while I/O operations signal completion at some later time using a callback function or a signal. For that reason, support for asynchronous I/O has not been specified as part of the standard SPM framework.

It is recommended that SPM applications avoid use of asynchronous I/O operations in favor of isolating synchronous I/O operations within their own Threads.

3.2.8 Interrupt Support

Support for and interaction with interrupts and interrupt handlers is beyond the specific scope of this guideline. As a rule, applications should interact with interrupts through the intermediary mechanism of I/O calls into and out of device drivers and/or through the mechanisms defined in the separate “Standard Device Model” guideline.

4 Appendix A: Technology and Operations Summary

4.1 Threads of Control

In the context of a software system, a thread of control, or a thread, is a self-contained sequence of related operations that is logically separable from other sequences of operations. One may consider an analogy to the “thread of a conversation”, meaning the bits of a broader conversation, pertaining narrowly to a specific topic and excluding discussions of other topics, that may be separated from the rest of the conversation and assembled into a single, coherent, and focused chain of thought.

A thread is generally differentiated from a “function” (or other similar concepts) in that a thread is persistent, representing an ongoing operating sequence rather than a discrete event or calculation or activity – that is, a “function” is invoked by some controlling entity to perform its action and then returns control to the caller; whereas, a thread represents an extended sequence of actions that are repeated continuously, along with the controlling elements to manage that repetition. Hence, the control structure of a thread is entirely self-contained – it is not “invoked” by some other module but operates (at least in a logical sense) as an independent and enduring entity.

Further, a thread may need to coordinate its operations with other threads at certain points in its operating sequence but, aside from that specific and bounded set of synchronization points, its operations may be considered to be independent of the operations performed by other threads.

Conceptually, different threads operate concurrently, rather than sequentially – one thread may write data to a disk at the same time that another thread is reading keystrokes from a keyboard at the same time that another thread is performing some time-consuming calculation.

By way of analogy, consider a typical electronic system which comprises a network of interconnected components – passive components, like resistors or capacitors, and complex integrated circuits like processors, memories, peripheral controllers, and the like; or, at a higher level, full-featured functional blocks, like processor and disk subsystems, on discrete circuit boards or even in discrete enclosures or equipment racks. Each component performs some function that represents a limited subset of the overall system functionality; and components interact with each other in extremely constrained ways, via the signals passing through the component boundaries and utilizing data busses or communication channels or dedicated wired connections.

Each component in such a system may be considered as implementing a thread of the overall hardware functionality – a self-contained and ongoing sequence of operations that is logically (and, in the case of hardware, physically) separable from other operations. A microprocessor and a disk controller each perform their own complex functions that extend over time and do not depend in any direct way how the other component performs its functions; and their only interaction is in the form of a handoff: "Here is a piece of data that must be written to storage; go deal with it, while I get on with my other business, and tell me when you are done."

In a hardware system these physically distinct "threads" literally perform their functions concurrently – the microprocessor can be doing a calculation while the disk controller is moving data from its buffer onto the disk, with no interaction or interference unless one requires a service from the other. Components synchronize their activities with other components through a limited set of specific physical interfaces and events – say the microprocessor issues an I/O command which causes data to be placed onto the system bus and then latched into the disk controller's input register.

Although it is common to think of electronic system designs, especially designs for data acquisition and processing, in terms of flows of data from one component to the next – to think of them in terms of a sequence of operations propagating through the hardware components over time – that is not the way the hardware actually works. Hardware is inherently parallel and inherently modular. That fact allows sections of a hardware design – a particular integrated circuit or a complete circuit board or an entire equipment rack – to be designed based solely on its limited functional requirements and interface constraints, largely in isolation from the design of other components; and it allows such components to be re-used as building blocks across multiple applications and designs.

Similarly, software designed using the concept of concurrent threads performing their functions simultaneously, rather than being conceived in terms of a cascade of activities that follow a single piece of data through the system, may achieve the same kinds of modularity. However, such modular threaded design typically requires support from the operating environment beyond that required to execute a single, integrated sequence of actions.

As with concurrent electronic components, concurrent software threads must have a mechanism by which synchronization and data signals pass through their boundaries and activate or inhibit their functions. Electronic components, and threads running on their own dedicated processors, can "poll" their signal interfaces (internal shared data structures or physical I/O channels) to determine when new synchronization signals or data have arrived.

However, when multiple threads are sharing a single processor then the processor itself – and its available execution time – may represent a scarce resource; and it is often undesirable to squander that resource – to waste processor time – on a continuous polling of synchronization and data channels in an effort to detect infrequent changes. Rather, there should be some mechanism – the scheduler equivalent to a hardware interrupt – by which threads that are in an idle state, awaiting some event, can be suspended from actively using the processor until that event occurs. That is, there should be some mechanism for suspending thread scheduling until a notification arrives indicating that it may continue its operations, somewhat analogous to a latching signal that indicates a valid data value has been presented to the input of an integrated circuit. Along with the scheduler itself, these suspension/notification mechanisms are the primary support required from an operating environment to support concurrent programming models.

4.2 Scheduling and Prioritization

In a system with multiple physical processors different software threads, like different hardware components, may literally run simultaneously, each assigned to its own processor. However, in a single-processor system -- or on any system with more operating threads than processors – some number of different threads must share a processor and cannot literally "run at the same time". In those cases the operations of different threads must be interleaved in some fashion to give the illusion of operating simultaneously.

```
enter thread 1
      .
      .
exit thread 1          enter thread 2
                             .
                             .
                       exit thread 2          enter thread 3
                                                    .
                                                    .
enter thread 1                                exit thread 3
```

Figure 4-1: Thread Scheduling Example

Figure 5-1 shows an example of that process, with time progressing from top to bottom and each column representing a different thread performing its operations. At any instant in time (at any vertical position in the figure) only one thread is doing anything. But over the entire time interval from top to bottom in the figure all three threads have done something; and, therefore, viewed over that time-scale the three threads appear to be all operating at the same time. This is equivalent to the process of Time-Division Multiplexing (TDM) on a communications channel, in which each logical channel is assigned a small time-slot within a larger overall frame transmitted on the physical channel. And it is the process that allows multiple users to share a single workstation or main-frame computer through different terminals, or that allows several different programs to be running on the desktop of a Windows computer at the same time.

The interleaving of threads to give this illusion of simultaneity is done by a scheduler, a system service which arbitrates among the various threads requesting time from the processor, and the operation of the scheduler is called thread scheduling (or simply scheduling).

The simplest form of scheduling, a *control loop*, is built into the structure of a software program. Imagine that the three columns in Figure 5-1 were shifted horizontally until they all aligned into a single column. The result would be a single thread, the control loop, in which the programmer explicitly interleaved the operations of the three original threads (now sub-threads of the control loop) into a single larger sequence. Using a method like this, one could conceptualize a software application in terms of separate logical threads but implement it as a single well-structured physical thread. This has the advantage that it requires no support from an operating system, and it creates a program that is not only fully deterministic – its sequence of execution is completely predictable from the structure of the code and knowledge of its inputs – but determined a priori by design. And it is structurally simple and easy to understand.

However, for an application of any significant size or complexity it can also become very quickly unwieldy, especially if the various logical threads must perform their functions on different time-scales and/or if they must respond in real-time to asynchronous events generated outside the system or by other logical threads within the system. A thread embedded within a control loop must wait until processing reaches its position in the sequence before it can respond to an event; and the time it takes to do so is entirely dependent on the structure built into the loop and on the activities of any threads that appear ahead of it in that structure.

Further, a thread that requires a lot of time to execute its functions can tie up all the other threads for long periods, thus tarnishing the illusion that all threads are running simultaneously over reasonably short time-scales. It is possible to restore that illusion by breaking the logical threads into smaller pieces, that will execute faster, and jumping back and forth between them. But with a thread broken apart and interleaved with other threads it ceases, in any meaningful way, to be modular.

Finally, within a control loop the only mechanism available for threads to detect events to which they must respond is to poll for them – the loop can't suspend itself and wait for a notification because that would suspend all threads at once – which can be unnecessarily time-consuming if the events occur only rarely.

If interleaving the execution of threads to approximate concurrency is not built into the structure of the program by using a control loop then some mechanism outside the program structure must perform that function. That mechanism requires two components: an algorithm to decide which of many threads requesting execution is assigned to the processor at any given time; and a means of storing the context[1] of a thread that is being unloaded from the processor and restoring the context of a thread which is being loaded into the processor. As stated above, the process of determining which thread to execute is called scheduling; the loading and unloading of the context is a context switch.

A context switch can happen for a variety of reasons: because the current thread has voluntarily yielded the processor to allow another thread a chance to run; because the current thread requires access to some unavailable resource or needs some data which has not yet arrived before it can proceed any farther; or because the scheduler has determined, based on some other criterion, that some other thread has a better claim on the processor than the current thread does. All those considerations are taken into account in a scheduling algorithm.

The intricacies of the various algorithms used for scheduling is beyond the scope of this guideline. In broad strokes there are two general classes of scheduling techniques that can be employed:

- *Cooperative:* A thread runs until it explicitly 'shares' the processor by a call to the scheduler. In this method each thread must be designed to be well-behaved, with specific points during its execution when it agrees to allow other threads to run. A Control Loop is a special case of a cooperative scheduler in which the thread switch points are coded into the loop itself. Note that in a cooperative scheduler, as with a control loop, a poorly-designed or badly-behaving thread can hold the processor indefinitely and prevent other threads from functioning.

[1] The "context" of a thread is the environment in which it is operating (see section 2.3.3.3). It includes, in particular, the values currently loaded into all the microprocessor's operating registers and all the local data values that the thread has allocated for its own use. Those must be stored so that when the thread again resumes operation its data is all exactly as it was before the thread was interrupted, notwithstanding what the other threads that ran in the interim were doing to change them.

- *Pre-Emptive:* A thread may be suspended by the scheduler asynchronously (at any arbitrary point within the flow of its internal operations) rather than only at specific share-points. This implies that no thread can tie up the processor by design (though it can do so as an artifact of the algorithm by which threads are chosen for execution). Since a thread can be suspended at any point during its operation, the "context" that must be saved during the switch is more complex than it might be for a co-operative task switch. When a thread shares the processor co-operatively it knows, a priori, that another thread is going to run and that, as a result, it is possible some of its data may not be valid when it resumes operation. When a thread is suspended asynchronously, and then resumes its operation where it left off, it has no indication that another thread has run – from its perspective it stepped from one operation to the next one within its sequence; the fact that another thread did something else between those two steps is invisible.

Within those two broad classes there are two general types of algorithm that can be used to determine which thread should next be assigned to the processor:

- *Time-Sharing*, Time-Sliced or Round-Robin: Threads are scheduled in a fixed sequence; when one thread is suspended the next one in the sequence is activated. During some bounded scheduling cycle all threads are awarded some slice of time in which to execute. This is an 'egalitarian' method inasmuch as no thread is allowed to prevent another thread from running. If the underlying scheduling method is cooperative then scheduled threads run until their next share point and the timing of the scheduling is determined by the internal thread dynamics. If the underlying scheduling method is pre-emptive then each thread will be awarded a "slice" of time and will be asynchronously suspended when its time expires, so the timing of scheduling is determined by the scheduler. Some variants of the algorithm allow threads to be 'weighted' according to how much processor time they will get – say one thread gets two time slices for every one time slice that another thread gets – but every thread will get its share of the time over the course of the scheduling cycle.

- *Prioritized:* Each thread is assigned a priority, and the scheduler always allows higher-priority threads to execute in preference to lower-priority threads when both are ready to run at the same time. In a cooperative scheduler the higher priority thread is not activated until the currently-executing thread reaches its share-point, so the prioritization determines the order in which threads are scheduled but not the timing. A pre-emptive prioritized scheduler will allow a higher-priority thread to run as soon as it is ready – it will asynchronously suspend a lower-priority thread in favor of the higher-priority thread – so the prioritization also affects the timing of the scheduling process. Since higher-priority threads will always be scheduled in preference to lower-priority threads, threads running in a prioritized scheduler (cooperative or pre-emptive) must be properly designed to suspend themselves – for instance, while they are awaiting new data – if they are not to prevent lower priority threads from ever getting a share of the processor time.

There are many variants and combinations of these basic types – for example, prioritized time-slicers which allocate each thread at the same priority a slice of processor time and then suspend it in favor of another thread at the same priority. And there are mathematical models for determining the processor loading, maximum latency, and deadline performance of threads operating within a particular type of scheduler. In general we will not attempt to detail all those variations here, except to note that the fundamental requirement for any scheduling algorithm that is designed to support real-time latency and deadline requirements is that it provide for pre-emptive scheduling and thread prioritization. Given such a scheduler, a designer can guarantee a deterministic scheduling outcome and variants on the Rate Monotonic Scheduling model can be used to analyze thread performance and, more importantly, to verify and guarantee that all the threads of an application will meet their real-time deadlines.

4.3 Threads and Processes

Within a software operating environment a thread, as defined above, is explicitly supported as a logical element which is typically called a Thread or a Task. Some systems only support a single thread of operation – a control loop – but any system designed to support concurrent processing models will support the operation of multiple concurrent threads in one form or another. To allow concurrent threads to operate each Thread must define and maintain its own local "context", which is the minimum set of information that defines its current state. That typically means its own copy of the set of hardware registers within the processor and a private stack for temporary variable storage.

Many software operating environments define some or all of their thread support in terms of Processes, rather than in terms of Threads or Tasks. A Process is an organizational concept, rather than a functional one: it is a logical (or virtual) machine context[2] in

[2] A "virtual" object in a software system is a logical model of a real-world object like a display or an I/O device or an entire computer (a "machine"). The virtual object is defined entirely by the functions it can perform and the interfaces it presents to other software components, not by its internal operating structure.

A good example of this is the windows provided by the Microsoft Windows, MacOS, X-Windows, and other windowing operating systems or environments, which may be thought of as "virtual displays" for the programs which use them. Each program is assigned a window (or several windows) in which it presents its output information. To the program, the window appears to be (because it behaves like) a dedicated display device: the program can write information to the window, no other program can get access to the window or overwrite what has been written to it, and the program does not need to know about or take into account other windows that might be used by other applications. But the display provided by the window is "virtual" because it has no physical existence in the way that a flat panel or CRT monitor does. Rather, the window is a logical model of a display that has been mapped onto some portion of the physical display and is sharing it with other windows.

which one or more Threads may operate, defining memory and/or I/O address spaces, hardware access mechanisms, and other system services through which Threads can interact with the physical machine on which they are operating.

To prevent Processes from inadvertently interfering with one another, a typical general computing environment makes the address spaces, hardware access mechanisms, and system service APIs private within a Process – that is, Threads operating within the context of one Process have no access to the address spaces and resource interfaces used by Threads operating in the context of another Process. These protections are designed to prevent problems when different users may run different applications simultaneously – when no one user has complete control over what Threads may or may not be running in the system at any given time. In fact, the fundamental purpose of a Process is to present Threads a "virtual machine" on which to run – to present a model of the operating environment that hides, to the extent possible, the existence and effects of other Processes. Hence the "context" provided by a Process is an entire virtual machine model.

Although the protection mechanisms which isolate Processes – private address spaces and I/O handles and the like – are considered necessary in a general computing environment they are not a strict requirement for Processes. In particular, operating systems designed for embedded applications, in which the system designer asserts complete control over what Processes and/or Threads are extant in the system at any given time, may not provide those protective mechanisms.

We should note here that *a Process does not execute any operations* -- it is merely a logical construct, a virtual model for a machine on which Threads can operate. In a system which does not support multiple Threads within a Process – that is, in many older and non-"real-time" operating systems – each Process is automatically – and transparently – associated with a single operating Thread which executes the Process operations. Documentation for such systems rarely mentions "threads" and treats the Process rather than its associated Thread as the concurrent execution unit. Hence Processes are historically and commonly thought of as operating units rather than as organizational units. But that usage obscures the subtleties underlying threading necessary to a thorough understanding of more complex multi-threaded systems. In a similar fashion, many operating systems designed for use in embedded systems claim explicitly to support "Threads" but not "Processes". What such claims really mean is that they support exactly one (implicit) global Process which defines the context in which all Threads operate.

In a similar fashion, the "virtual machine" represented by a Process is a logical model of a computer, with processing capacity, memory, and I/O channels. That model is mapped onto the physical computer and is sharing it with other Processes.

For purposes of the context switch a Thread is considered a *"lightweight"* object, while a Process is considered a *"heavyweight"* object. The two designations are descriptive not quantitative, meant to communicate the relative complexities – and therefore the relative execution times – of the context switching operations. If we consider what the "context" is in the two different situations we can see why the designations are as they are.

When switching from one Thread to another within a Process, the context that must be saved/restored is fairly minimal: it typically comprises nothing more than the contents of the internal processor registers, including the instruction pointer for the code to be executed and the pointer to the stack for local variable storage. The entire context switch can take place within the local Process address space and without interaction with the operating system.

The context of a Process, on the other hand, models an entire virtual machine. When switching from a Thread in one Process to a Thread in another Process, not only the Thread context but the entire context associated with presenting the virtual machine must be swapped. Especially in systems which erect strict barriers between Processes, this will require resort to calls into the protected operating system kernel, and probably a significantly larger set of data to be stored into and recalled from memory.

The same general concept applies to interactions between Threads and the various interlocking, synchronization, and messaging elements defined in later sections. An element that operates entirely within a single Process can operate entirely within the local Process address space and without resort to the operating system kernel; an element that crosses a Process boundary requires the operating system to translate across the private address spaces and resource references – to move the data associated with the element from one virtual machine to another. This is an inherently slower operation.

For these reasons, applications with real-time requirements and/or demanding processing loads are best designed using Threads within a single Process to minimize cross-Process interactions.

It is worth noting in this context that, as a rule, systems which support multiple processes – and particularly systems designed for general use and which, therefore, support simultaneous processes from different users – will utilize some form of time-slicing between processes to ensure that badly-behaved processes cannot monopolize the processor time. This complicates – and sometimes subverts – the constraints required to guarantee real-time response to stimuli and to guarantee deterministic operation, especially if the designer cannot know a priori what other processes (and associated processing loads) other users may install.

4.4 The Threaded Architecture Model

In a "threaded" software architecture the functions of an application are decomposed into discrete and self-contained operations that must be executed on an ongoing basis, and each such operation is assigned to its own thread.

For instance, a simple application may wait for keyboard input, take some action based on the input, and report some status that results from that action. A traditional single-threaded application would do each of those in turn within a single loop; as long as no input was received there would be no action required and nothing to report, but once an input was received it would move on to acting upon and reporting the input and no more inputs could be accepted until both the actions and the reporting were completed.

```
while ( !bStopThread ) {
    <do things>
        .
        .

    If ( 0 != pthread_<entity>_timedwait ( <entity>, timeout ) ) {
        /* timed out without detecting the condition */
        .
        .
    }
    else {
        /* detected the condition */
        <do things>
        .

    }

    <do things>
        .
        .
}
```

Figure 4-2: Basic Thread Structure

A threaded application, in contrast, might assign a thread to each of those activities – detecting keyboard input, taking action, and reporting results – and all of those could be happening in parallel, with one thread detecting and queuing up new keyboard inputs while a second was still processing the last one, and with some new action starting even while the last report was still being written to the display. A scheduler would determine which thread was allowed to operate at any given time, including interleaving their operations as necessary, without guidance from the threads themselves.

The basic structure of a thread is illustrated by the 'C' code snippet shown in Figure 5-2. The thread runs in a continuous loop (until it is told to stop); on each pass through the loop it checks to see if some condition – a new key input or completion of an action – has been detected; it either performs some action or not depending on whether that condition was detected; and when it tests for detection of the condition it waits – it suspends itself to allow other threads to run – until it either detects the condition or has waited too long for it.

There are variants on this basic structure – the condition test may pend indefinitely, for example, rather than terminating after some fixed amount of time; or the thread may depend on a time-sliced scheduler to suspend its activities; or it may have additional places within its loop where is suspends waiting for resources to be available or for other conditions to pertain – but this is the basic template for thread operations.

Note that the architectural model governing this structure is explicitly "event-driven" – that is, each thread executes its function in response to an "event" – the detection of the condition for which it is waiting – and is otherwise quiescent. In practice most applications, threaded or otherwise, are ultimately driven by "events", so that is perhaps not a distinction that resonates. But because each thread is responsible for only a small piece of the overall application – for reading the keyboard, or for writing a report to the display, or for performing some calculation – the "event-driven" nature of the structure is much more explicit than in a single-threaded application in which whether or not to execute a particular function is typically determined by stepping through a series of conditionals rather than by explicitly waiting for that condition to "arrive" at the wait point. In this architectural model it is easy to see how those "events" are relayed from module to module through the system and how each individual module either responds to events or generates new ones.

Note, also, that because each thread is responsible only for one small piece of the overall application there is generally no need for a thread to be interrupted to deal with some asynchronous event that is unrelated to its limited function. A properly-designed threaded application would typically provide a discrete thread to handle each such asynchronous event; and each individual thread would synchronize itself to its own event by waiting for it at the "wait" point within its loop. Hence, threaded programming architectures do not generally require or rely upon explicit "asynchronous" event handlers, like "callback" functions that interrupt thread operation, to manage asynchronous system behavior.

4.5 Interlocks

When multiple concurrent threads can access common resources, like a physical I/O channel or a shared memory area, it is possible for their actions to interfere with each other. For instance, when a thread writes a multi-byte message through a communication channel the message flow is spread out over some amount of time as the physical channel handles the I/O and transmission operations. If a second thread attempts to write its own multi-byte message to the same channel during the same time interval, the two messages are likely to end up interleaved on the channel and incomprehensible to the receiver.

To resolve these types of problems the operating environment must provide mechanisms for interlocking access to shared resources, allowing access to only one thread at a time. To continue the electronic design analogy, an interlock is conceptually similar to a bus arbitrator that resolves conflicts over bus usage among multiple processing units, or (in a loose sense) to the collision detection or avoidance mechanisms required on network channels to ensure only one station is actually passing a valid message through the network at any given time.

As with threads which are waiting for data, threads which are interlocked from access to a resource they require should suspend their operation until they are notified that the resource has been released, and such suspension/notification functionality should be integral to the interlock mechanism.

4.6 Synchronization

With multiple threads operating concurrently to execute the various functions of an application it will be inevitable that they will sometimes need to coordinate their efforts in some fashion or to synchronize their operations to real-world events. Some threads will need to perform a task repeatedly at a fixed interval; others must process some data every time it becomes available, or respond to an external signal or to an internal state change. As with interlocks these other synchronization mechanisms must cooperate with the scheduler to allow temporary suspension of threads that are awaiting synchronization, and notification of when the synchronization has occurred so threads may resume their operations. In addition, to perform internal coordination, threads require a mechanism for asserting synchronization events targeted to other threads as well as for receiving them.

4.7 Messaging Channels

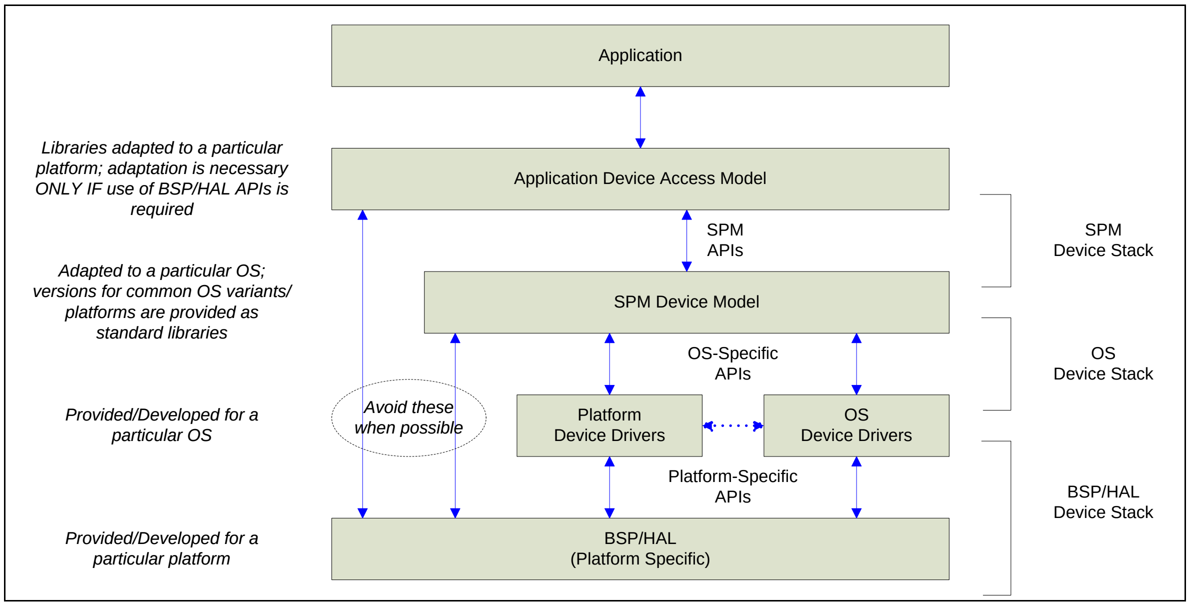
In addition to raw synchronization, events threads will often need to share data and require notification when that data has been updated. In particular, data processing systems tend to operate on discretized flows of data that may be processed through one or more threads, and require mechanisms for moving that data flow through the chain of threads. Even "continuous" streams of data will be processed in packets of some discrete size (even if that size is a single byte or a single bit), so such mechanisms typically take the form of a channel through which packetized messages can flow. As with interlocks and synchronization objects, message channels must cooperate with the scheduler to allow threads awaiting data to be suspended when data is not available and notified when data has arrived.

4.8 Hardware Access and I/O

Although standard models for access to various hardware devices and communication channels is the subject of other guidelines, those functions require support from the SPM inasmuch as events derived from such external interfaces are often a primary source of asynchronous data and notifications which control Thread operation; and access to such external interfaces is often a primary subject of Thread control interlocks.

Hence, support for interlocks, event notifications, and messaging channels within the APIs for standard device and communication channel models is a primary justification for development and use of the SPM. Further, any device I/O functions provided by the operating environment and used by those device and communication models must cooperate with the SPM scheduler and its associated interlocks and synchronization facilities. Hence, one component of the SPM specification will be the range of such I/O functions available and the mechanisms by which they interact with the threading system.

A general hierarchical model of device I/O is shown in Figure 5-3. The SPM device model is integrated with the SPM scheduler to allow for thread suspension and notification as a result of I/O activities. The "Device Access Model" represents any additional device abstractions and/or access functionality that might be provided between the SPM and the application. It is beyond the scope of this guideline.



Application
Libraries adapted to a particular platform; adaptation is necessary ONLY IF use of BSP/HAL APIs is required
Application Device Access Model
SPM APIs
SPM Device Stack
Adapted to a particular OS; versions for common OS variants/ platforms are provided as standard libraries
SPM Device Model
OS-Specific APIs
OS Device Stack
Provided/Developed for a particular OS
Avoid these when possible
Platform Device Drivers
OS Device Drivers
Platform-Specific APIs
BSP/HAL Device Stack
Provided/Developed for a particular platform
BSP/HAL (Platform Specific)

Figure 4-3: Device I/O Hierarchy

4.9 Interrupt Service Routines

An Interrupt Service Routine (ISR) is a specialized thread of control typically associated with some asynchronous hardware or operating system event – an interrupt – and managed by a pre-emptive prioritized hardware scheduler directly supported by specific processor instructions. That is, interrupts are scheduled by a dedicated scheduler separate from the one used for general thread scheduling; and the entire software operating environment associated with the operating system or kernel (including its scheduler and all the threads managed by it) may be considered, for analytical purposes, to be the 'default' or 'idle' thread associated with the interrupt scheduler – that is, it may be considered to be the lowest-priority thread that runs when no higher-priority threads (interrupts) are ready.

Because an interrupt is managed directly in hardware it bypasses the constraints of the software scheduler, interlocks, and other controls on Thread scheduling; that is, because they are managed directly in hardware ISRs automatically attain higher priority than Threads and will pre-empt them when an interrupt occurs. Because of this, ISRs are used to respond to events that require extremely low latency; but, because they can indefinitely suspend the operation of any Thread, independent of its priority, ISRs should typically perform only the minimal amount of processing required to manage the interrupt. Further, in general ISRs are assigned high priority precisely because the operations they perform require low latency between the assertion of the interrupt signal and the software response to that signal. For that reason, ISRs will generally be designed to ignore interlocks provided by the software scheduler (which could suspend them indefinitely and defeat the goal of low latency response). Hence, Threads and ISRs should not be required to access the same resources unless such accesses are controlled by some mechanisms beyond the scope of interlocks provided by the scheduling environment.

As a rule, ISRs are tied so closely to the specific hardware on which they operate that a standardized API for managing them is not practical. For that reason we exclude ISRs from the SPM function and API; they must be dealt with as part of the Hardware Access Layer (HAL) provided for a given platform and will typically be adapted for the specific hardware at the level of device drivers.

That said, however, the SPM should provide functional APIs for specific synchronization and message passing operations that may be invoked from within an ISR to allow the ISR to pass synchronization events and data to threads.

When analyzing a system to ensure that threads meet their deadlines and that processor loading is not prohibitive, ISRs and their execution time, modeled as highest-priority threads, should be taken into account.

4.10 Critical Sections

Occasionally a thread must perform some monolithic operation during which it needs to guarantee that it will not be forced to yield the processor – say a multi-step interaction with a hardware device in which the transaction must be completed within some bounded amount of time once it has begun. In such cases the thread must have a mechanism for disabling the scheduling of other threads during the operation.

In a cooperative scheduler, of course, a thread can ensure it is not interrupted merely by not signaling a “share”. Similarly, the most straightforward method to protect such operations in a prioritized, non-time-sliced scheduler would be to place them into threads assigned a maximum priority. In that case a thread that started such an operation could not be re-scheduled because there would be no other threads in the system which could assert a higher priority and claim the processor.

However, in any system which utilizes time-slicing the scheduler will routinely suspend threads in favor of other threads operating at the same priority when the time-slice expires. In that case, either all such non-interruptible operations must be placed into a single highest-priority thread or the scheduler must provide a mechanism by which a thread can block scheduling during the operation; and, in a scheduler which does not support thread prioritization, such a blocking mechanism is the only way to protect those operations.

A monolithic operation that cannot be interrupted by a context switch is called a Critical Section, and the mechanism used to protect it is a Critical Section Lock. By definition, there must be a single, global critical section lock that applies to all threads in all processes – otherwise a thread from another process could cause a re-schedule and the operation would not be protected. A thread protects its critical section by asserting the lock before entering the section and releasing the lock upon exiting the section.

Note that the critical section lock effectively disables all thread scheduling temporarily. For obvious reasons, then, the period of time during which such a lock is held – and the use of such locks in general –should be minimized.

Note also that, in general, if a critical section should not be interrupted by a context switch within the software scheduler, it should also not be interrupted by a context switch associated with an interrupt service routine – the effect on the timing of the critical section could be the same in either case. Hence, in the general case assertion of a critical section lock should also disable hardware interrupts.

In specific cases that may or may not be necessary, since interrupt service routines will normally be designed to minimize their execution times, and hence to minimize the amount of time by which they can delay the operation of a critical section. However, designers should note that critical sections protected using thread priority rather than a critical section lock may nonetheless require disabling of interrupts if interrupt service routines can delay execution of the critical section by enough to cause problems.

Note, finally, that, as a matter of practice, accesses to memory spaces or I/O channels that must be treated as monolithic to prevent the corruption or misinterpretation of data (rather than to control access timing) should be protected using some form of interlock rather than using a critical section lock. That ensures that only threads which actually access the memory or I/O channel will be affected by the protection mechanism; all other threads and interrupts can continue to be scheduled as appropriate.

4.11 Callback Functions

To this point discussions of notification mechanisms have described a process of thread suspension and resumption through the action of the scheduler and with the cooperation of the affected threads. Typically a thread requests some synchronization object – an interlock (for resource access) or a time-tick (for temporal synchronization) or a message (for data flow) – and is suspended at that point in its operation unless/until the object is available, at which time it resumes its operation at the same point. In that sense, although in general threads may be suspended by the actions of higher-priority threads (or by a time-sliced scheduler) asynchronously with respect to their internal operations, in the specific case of these notification mechanisms scheduling is fully synchronous from the perspective of the thread: it was initiated and mediated by the explicit request for the synchronization object.

Some operating environments provide a second type of notification mechanism, a Callback, that creates a temporary thread – conceptually similar to an interrupt service routine – in response to an asynchronous notification event. The temporary thread operations are typically contained within a Callback Function, which is installed by the thread requiring the notification and executed by the notification manager in its own private context and asynchronously to the operation of the thread which installed it. In other words, a Callback Function is merely a function that is installed to be executed when some asynchronous event occurs; in that sense it is conceptually similar to an ISR, except that it is typically invoked by a device driver (or some system-related thread) rather than directly by the hardware-based interrupt scheduler. The purpose of the callback function is to handle the condition or data which led to the notification, which may include invoking some mechanism local to the thread itself to signal to the thread that the notification has occurred.

Note that, because a Callback executes asynchronously, it is generally constrained in the same ways as are ISRs, with restrictions on how they use the scheduler and operating system functions to mediate resource accesses.

4.12 Priority Inversion

The use of interlocks and critical sections to restrict scheduling can, if not properly managed, cause high-priority threads to become blocked by low-priority threads. For instance, if a low-priority thread (priority=L) is holding the interlock for a particular resource and a high-priority thread (priority=H) requests the same resource, the high-priority thread will be suspended until the resource becomes available – which will not occur until the low-priority thread releases it. But any thread with a priority higher than L can prevent the low-priority thread from executing and, thereby, prevent it from releasing the interlock. And, hence, the high-priority thread will be disabled until all threads at priorities higher than L – and, of particular concern, all threads at priorities higher than L but lower than H – have relinquished the processor and allowed the low-priority thread to complete its use of the resource. In effect, the high-priority thread has been blocked by lower-priority threads, threads that have priorities between its own and that of the thread which holds the lock. This situation is called a *Priority Inversion.*

Note that, because the low-priority thread can be blocked by other threads for an indefinite period of time, an a priori restriction on the amount of processing that the low-priority thread can do while holding the interlock does not solve the priority inversion problem. The only general way to prevent priority inversions is to avoid them entirely (by not allowing threads at different priorities to access the same lock) or to provide a mechanism for dynamic priority assignment by which the priority of a low-priority thread holding an interlock is boosted temporarily to the same priority as the highest-priority thread waiting for the interlock (a thread holding a critical section lock is, by definition, operating at the highest possible priority since it has disabled all scheduling, so a boost is not necessary). This process is known as *Priority Boost.* The effect of the priority boost is to ensure that only threads with priorities higher than that of any thread awaiting the interlock can delay the release of the interlock.

4.13 Timeouts

In a perfectly designed and perfectly operating system, threads waiting for data or synchronization events would always be activated when their primary functions were required and would never need to be activated at any other times – that is, there would be no need to handle contingencies outside the scope of the nominal data/control flows. In any practical system, however, a designer must assume that there will be times when things don't happen according to the intent of the design, either because of errors in the design itself or because of some failure external to the software. For instance, a hardware device that generates an interrupt on a periodic basis can fail; as a result, the interrupt will not be asserted and a thread that is to be scheduled periodically based on that interrupt will not be scheduled; and, as a result, some system function that is supposed to occur periodically will not be executed; and, as a result, other threads which depend on that function having been executed may also fail to perform their tasks; and, worst, since the failure is that *nothing happened*, no one may know that things have gone awry.

In general, when a thread is suspended awaiting a specific notification event from the scheduler it is not simultaneously listening for other notification events (some operating system do provide for that possibility, but not all do) and so there is no general way to notify such threads that some error has occurred which will prevent the notification they are waiting for from ever arriving. In effect, those threads become “hung” in the waiting state.

Under some circumstances this can happen even when nothing has gone wrong. A thread that is to execute based on a notification that occurs once each hour may spend the entire hour between notifications suspended. If some other event – say a user-generated request for the system to shut down – happens while the thread is suspended, the next opportunity to communicate that request to the thread is on its next time notice, which may be an hour from now.

It is useful, then, to have some mechanism to limit the amount of time that a thread should remain suspended, to give it an opportunity to process exceptions and/or asynchronous state-change requests in between long-interval or randomly-occurring notifications. One might imagine some complicated structure of notification mechanisms that would allow this, but the simplest way is to impose an upper bound – a *Timeout* – on the amount of time it is allowed to wait. That way, after some maximum amount of time the “wait” function will return with a status code indicating that it timed out (and did not return data) and the thread will wake up, note the timeout, deal with anything else that has come up (for instance, read information from some secondary notification channel), and go back to waiting again.

For this reason, all SPM notification functions – that is, all functions which suspend thread operation pending notification of some event – should support timeout functionality.

4.14 A Simple Example

Figure 5-4 illustrates the use of a multi-threaded application to create a single-channel hybrid analog/digital control loop. In the figure, circles represent threads, rectangles, represent data access blocks, skinny arrows represent ordinary data flows, and fat arrows represent scheduler synchronization elements.

The implementation shown is, perhaps, more complex than this application strictly demands, and the use of particular threading and communication elements was chosen to be illustrative and not necessarily optimal. However, this example is simple enough to be easily understood, and could provide a framework for adding more complexity as the application requirements expand – say if it had to handle multiple independent control loops operating on different time bases and/or high-speed analysis and triggering channels (not to mention more complex user controls and communications) within the same framework.

The basic control flow is as follows: an analog feedback channel is sampled at a high rate using an A/D converter; the raw *Feedback Samples* are filtered to reduce noise and decimated in time to produce a *Feedback Signal* at some reduced sample rate; and a control algorithm is applied to create a *Control Sample*, which is written to an analog control channel through a D/A converter. In addition to the basic control functions, the raw and filtered data is written to a disk for archiving and post-operation analysis, and the filter and control algorithm parameters may be changed during operation in response to messages received through a serial port.

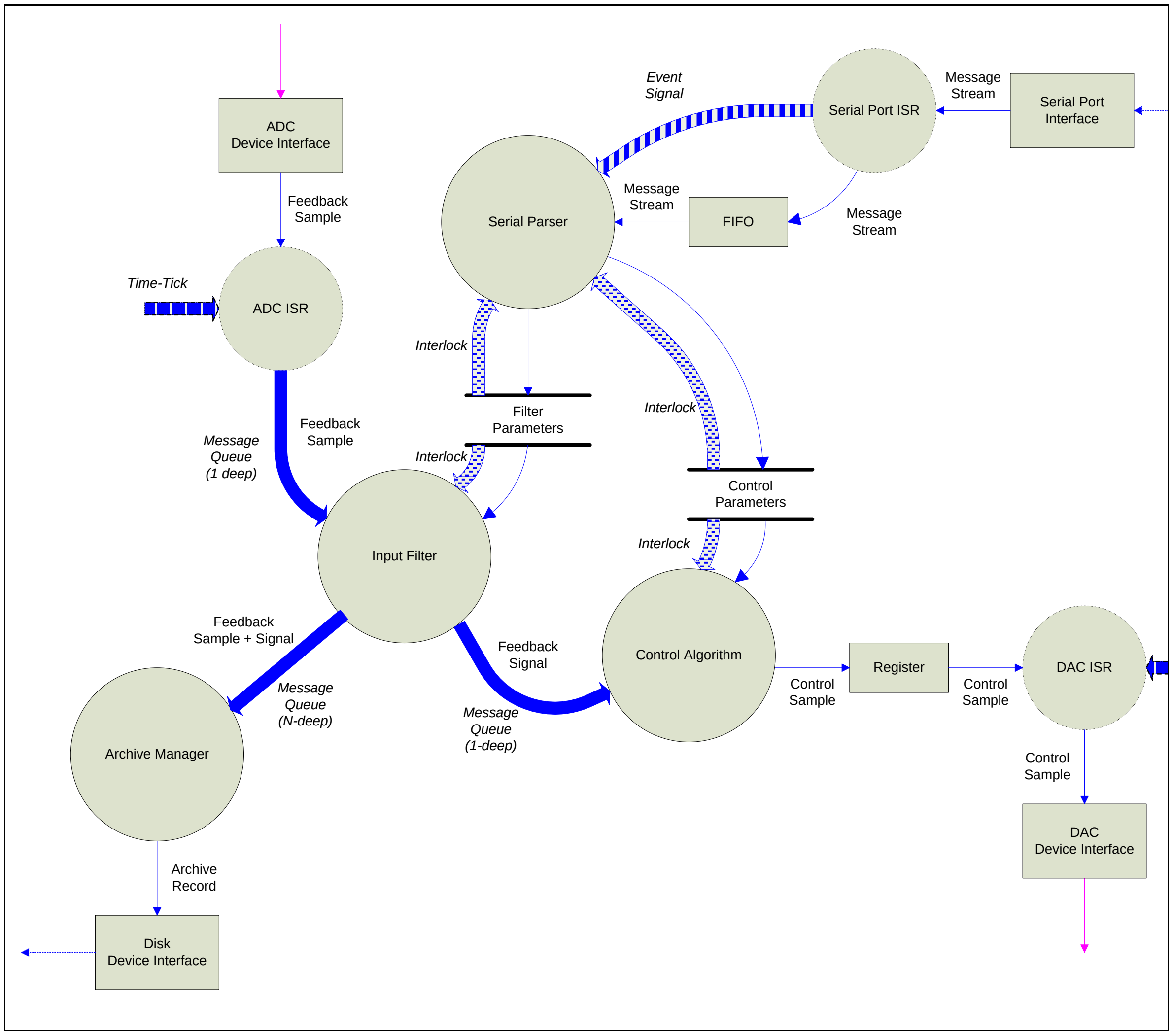


Figure 4-4: Simple Thread Example

Each of the major functions – sampling, filtering, controlling, archiving, and communications – is assigned to an individual thread, which may be designed independently of the other threads based on functional and input/output specifications, and which depends for its operation only on the input and output channels assigned to it.

Hence any thread may be tested as an independent entity (by using a test fixture to push data into its receive channels and to monitor data from its transmit channels) and may be lifted *in toto* from this application and used in another without modification.

As shown, analog sampling at the input and output are controlled using interrupt service routines operating on a time-tick to define a tightly regulated sample rate. In theory, a high-priority thread could have been used in place of either ISR but, because it is controlled directly in hardware, an ISR provides the quickest response and most consistent timing available in a software implementation.

The *ADC ISR* reads the A/D converter and places *Feedback* Samples into a message queue as they are received; the Input Filter thread waits on the sample queue for its data, processing one sample through the filter algorithm for each pass through its loop and using the parameters stored in a shared memory location to determine the function of the algorithm. We assume that the filtering operation takes significantly less time than the interval between samples, so the thread suspends between samples while it waits for more data to be pushed into the queue.

Each time the *Input Filter* thread produces a filtered/subsampled Feedback Signal, that signal is placed into a second message queue for use by the *Control Algorithm* thread. The *Control Algorithm* thread waits on the signal queue for its data, and processes one signal value through the control algorithm for each pass through its loop. As with the Input Filter thread, the Control Algorithm thread uses parameters stored in a shared memory location to determine the current algorithm configuration. And, again, we assume that the operation of the algorithm takes much less time than the interval between *Feedback Signals*, so the thread is suspended between samples.

For each Feedback Signal value the *Control Algorithm* thread produces a *Control Sample*, which is placed into the output register for use by the *DAC ISR*. Each time the *DAC ISR* is activated by its time-tick it reads the value from the output register and writes it to the DAC; if, for some reason, the output register has not been updated since the last sample (properly designed, this should not happen) the ISR will hold the output at the old value.

Note that the control loop imposes deadlines on the processing of the *Feedback Samples* into *Control Samples*; in particular, the filtering and control processing must always be finished before the next DAC time tick. To ensure that the threads meet their deadlines we would assign high priorities to the *Input Filter* and *Control Algorithm* threads, such that when data was available for processing they would not be blocked by the other application threads. Priority assignments based on a Rate Monotonic Scheduling algorithm would ensure that the threads could meet their deadlines.

In addition to posting the filtered/subsampled *Feedback Signals* to the *Control Algorithm* the input filter thread also places both the raw *Feedback Samples* and the filtered/subsampled Feedback Signals into a separate message queue for archiving. The Archive Manager thread waits for data on that message queue and writes it to the disk as it is received. Since the Input Filter thread posts the data to the message queue in the sequence in which it was processed the *Archive Manager* thread will receive it and write it to the disk in the same sequence. However, the disk write operation does not have to be completed on any particular deadline as long as, in aggregate, it can keep up with the sampled data rate. Therefore, the *Archive Manager* thread is assigned a relatively low priority to ensure it operates on an “as available” basis and does not interfere with the control threads. This, of course, requires the designer to assign a message queue large enough to ensure it does not overflow as the *Input Filter* gets ahead of the *Archive* Manager over the short term.

Input filter and control algorithm parameters in the shared memory locations can be updated by the Serial Parser thread as it handles incoming messages. The shared memory locations are protected by interlocks to ensure that the processing threads do not operate using partially-updated parameters if an update is in progress when a new sample arrives. Since we presume the *Serial Parser* thread will operate at a lower priority than the control threads, the scheduler will automatically and temporarily boost its priority to match the control thread priority whenever a control thread tries to take the interlock while the Serial Parser thread holds it, thus preventing a priority inversion that could disrupt control loop timing.

The communication between the serial port and the *Serial Parser* thread is implemented differently than the communication between the A/D channel and the *Input Filter* thread. In the case of the *Input Filter* we assume it will be assigned a high priority (to ensure it meets its deadlines) and that its processing time will be much less than the sample interval. Hence we expect that under normal circumstances it will be finished with its processing before the next sample arrives, and we use a message queue to notify it as each sample is received.

As with the *Archive Manager*, however, the *Serial Parser* will be assigned a relatively low priority to ensure that it does not interfere with the control functions, and so we might expect – especially for a high-speed communication channel – that there may be several (or many) characters queued up awaiting processing by the time the thread gets around to reading its input channel. Further, although the function shown in the figure (update the processing parameters) would probably not take a significant amount of time, in general a message parser may take quite a long time to process some messages and may very well not finish before more characters, or even several complete messages, have been received and queued. For that reason, we provide a simple FIFO in memory to hold the accumulated characters and an event signal to notify the thread that data is available.

An event signal exists in one of two states, *“signaled”* or *“unsignaled”*. Normally, the event will be “unsignaled” and the *Serial Parser* thread will suspend, awaiting a “signaled” state. When the Serial Port ISR receives a character, it writes it into the FIFO and sets the state of the event to “signaled” to indicate that the character has been queued; the setting of the “signaled” state alerts the *Serial Parser* thread and activates it. Once the *Serial Parser* thread wakes up (which may take a while if higher priority threads are active) it clears the state of the event to “unsignaled” and reads all the characters available out of the FIFO.

Note that this arrangement allows the *Serial Parser* thread to handle multiple characters – as many as are queued – before waiting again for a notification. That minimizes the number of calls into the scheduler in the case where we expect the data to arrive faster than the thread can respond.

If the ISR receives a new character after the thread clears the state of the event it will, again, set the state to “signaled”. In that case, when the thread again waits for the event it will not suspend but will return immediately with the “signaled” event and process characters from the FIFO until it is empty. This process will continue until there is no more data to process and the state of the event remains “unsignaled”.