HighIntegritySystems

WITTENSTEIN

SAFERTOS® for Aerospace Datasheet

SAFERTOS® for DO-178C provides aerospace developers with a responsive, robust and deterministic embedded Real Time Operating System (RTOS), supported by clear and concise certification planning documentation, comprehensive design and verification evidence supporting certification up to DAL A.

Functional Overview

The SAFE**RTOS** pre-emptive real time scheduler has the following features:

- Any number of tasks can be created - system RAM constraints are the limiting factor.
- Each task is assigned a priority - any number of priorities can be used.
- Any number of tasks can share the same priority - allowing for maximum application design flexibility.
- The highest priority task that is able to execute (i.e. that is not blocked or suspended) will be the task selected by the scheduler to execute.
- Supports time sliced round robin scheduling for tasks of equal priority.
- Queues can be used to send data between tasks, and to send data between interrupt service routines and tasks.
- Binary semaphores and counting semaphores make use of the queue primitive – ensuring code size is kept to a minimum.
- Mutexes and Recursive Mutexes supporting a priority inheritance mechanism.
- Tasks can block for a fixed period, or until a specific time is reached.
- Task Notifications, a lightweight alternative to using Queues, Semaphores and Event Groups.
- Event Groups/Flags, Tasks can be woken either by a Single Event or a combination of Events from the same Event Group.
- Software timers.
- Definition and manipulation of MPU regions on a per task basis.

Compact Footprint

Typical ROM Requirements	32kB
Typical RAM Requirements	1 kB
Typical Stack Requirements	400 bytes/task

Key Features

Certifiable to RTCA DO-178C up to DAL A

Design Assurance Pack with full source code

Royalty free production license

Unlimited number of developers

System Tasks

Including SAFE**RTOS** in your application allows the application to be structured as a set of autonomous tasks - the resultant system functionality being the sum of the functionality of the multiple tasks that make up the application.

Each task executes within its own context with no coincidental dependency on other tasks within the system or the scheduler itself.

Task States

Only one task can actually be executing at any one time. The scheduler is responsible for selecting the task to execute in accordance with each task's relative priority and state.

A task can exist in one of the states described in the Table 'Task States', with valid transitions between states depicted by the Figure 'Valid task state transitions'.

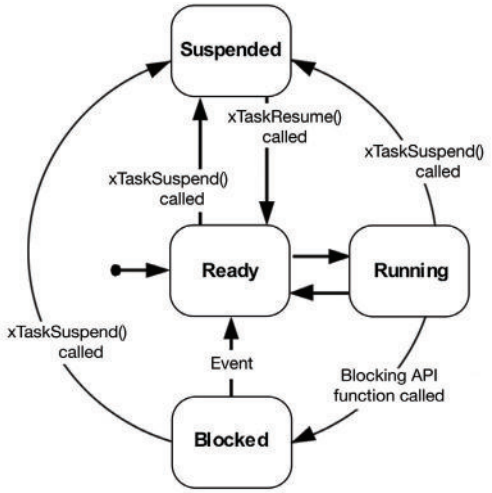

Figure *Valid Task State Transitions*

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600

email: sales@highintegritysystems.com
web: www.highintegritysystems.com

Table *Task States*

Task State	Description
Running	The task selected by the scheduler to execute and which is currently utilizing the processor.
Blocked	A task waiting for an event. It cannot continue until the event occurs. Tasks in the Blocked state always have a timeout period, after which the task will become unblocked.
Suspend-ed	A task will enter the Suspended state when it is the subject of a call to the xTaskSus-pend() API function, and remain in the Suspended state until unsuspended by a call to the xTaskResume() API function.
Ready	A task is in the Ready state if it is able to enter the Running state but is not currently the task that is selected to execute.

Task Priorities

A priority is assigned to each task when the task is created, the task priority can be altered during runtime.

Low numeric values denote low priority tasks. The lowest priority value that can be assigned to a task is 0. High numeric values denote high priority tasks. The maximum priority that can be assigned to a task is user configurable.

The Scheduler

The Scheduler has responsibility for:

- Deciding which task will be the task selected to enter the Running state, and performing the context switching accordingly.
- Measuring the passage of time.
- Transitioning tasks from the Blocked state into the Ready state upon the expiration of a timeout period.

Measuring Time

A periodic (tick) timer interrupt is used to measure time. The time between two consecutive timer interrupts is defined to be one “tick” period. Times are therefore measured and specified in “tick” units.

Scheduling Policy

The scheduler selects as the task to be in the Running state the highest priority task that would otherwise be in the Ready state. In other words, the task chosen to execute is the highest priority task that is able to execute. Tasks in the Blocked or Suspended state are not able to execute. Different tasks can be assigned the same priority.

When this is the case the tasks of equal priority are selected to enter the Running state in turn. Each task will execute for a maximum of one tick period before the scheduler selects another task of equal priority to enter the Running state.

While the scheduler will ensure that tasks of equal priority will be selected to enter the Running state in turn, it is not guaranteed that each such task will get an equal share of processing time.

Yielding

Yielding is where a task volunteers to leave the Running state by re-entering the Ready state. When a task yields the scheduler re-evaluates which task should be in the Running state. If no tasks of higher or equal priority to the yielding task are in the Ready state then the yielding task shall again be selected as the task to enter the Running state.

A task can yield by explicitly calling the taskYIELD() macro, or by calling an API function that changes the state or priority of another task within the application.

Scheduler States

The scheduler can exist in one of the states described by the Table 'Scheduler States', with valid transitions between states depicted by the Figure 'Valid Scheduler State Transitions'.

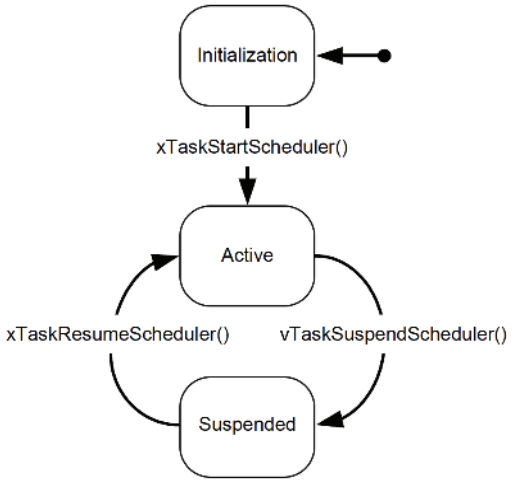

Figure *Valid Scheduler State Transitions*

The scheduler is started using the xTaskStartScheduler() API function. Calling xTaskStartScheduler() causes the creation of the Idle task. The Idle task never enters the Blocked or Suspended state. It is created to ensure there is always at least one task that is able to enter the Running state.

The scheduler enters the Suspended state following a call to vTaskSuspendScheduler(), and returns to the Active state following a call to xTaskResumeScheduler().

A code section that must be executed atomically (without interruption from other tasks or interrupts) to guarantee data integrity is called a critical region. The traditional method

Table *Scheduler States*

Scheduler State	Description
Initialization	The initial state, prior to the scheduler being started. While in the Initialization state the scheduler has no control over the application execution. Tasks and queues can be created while the scheduler is in the Initialization state.
Active	While in the Active state the scheduler controls the application execution by selecting the task that is in the Running state.
Suspended	The task that was in the Running state when the scheduler entered the Suspended state shall remain in the Running state until the scheduler returns to the Active state.

of implementing a critical region of code is to disable then re-enable interrupts as the critical region is entered then exited respectively. The macros taskENTER_CRITICAL() and taskEXIT_CRITICAL() are provided for this purpose.

Implementing a critical section through the use of taskENTER_CRITICAL() and taskEXIT_CRITICAL() has the disadvantage of the application being unresponsive to interrupts for the duration of the critical region. The scheduler suspension mechanism provides an alternative approach that permits interrupts to remain enabled during critical regions.

When the scheduler is in the Suspended state, by calling vTaskSuspendScheduler(), a switch to another task will never occur. The task executing the critical region is guaranteed to remain as the task in the Running state until xTaskResumeScheduler() is called.

Interrupts remain enabled while the scheduler is in the Suspended state. Critical regions implemented using the scheduler suspension mechanism therefore protect the critical data from access by other tasks, but not by interrupts. It is safe for an interrupt to access a queue while the scheduler is in the Suspended state.

A switch to a higher priority task that enters the Ready state while the scheduler is in the Suspended state will be held pending until xTaskResumeScheduler() is called. It is therefore still desirable for the scheduler not to be held in the Suspended state for an extended period. Doing so will reduce the responsiveness of high priority tasks.

Intertask Communication

SAFE**RTOS**® provides a queue implementation that permits data to be transferred safely between tasks. The queue implementation is flexible and can be used to achieve a number of objectives, including simple data transfer, synchronization and semaphore type behaviour.

Queue Characteristics

- The size of each item and the maximum number of items that the queue can hold are configured when the queue is created.
- Items are sent to a queue using the xQueueSend() API function.
- Items are read from a queue using the xQueueReceive() API function.
- Queues are FIFO buffers - that is, the first item sent to a queue is the first item retrieved from the queue.
- Data transferred through a queue is done so by copy - the data is copied byte for byte into the queue when the data is sent, and then copied byte for byte out of the queue when the data is subsequently received.
- Queues can have multiple senders and receivers.

Queue Events

Data being sent to or received from a queue is called a queue “event”.

When calling xQueueSend() a task can specify a period during which it should be held in the Blocked state to wait for space to become available in the queue if it finds the queue is already full. The task is blocking on a queue event and will leave the Blocked state automatically when another task or interrupt removes an item from the queue.

When calling xQueueReceive() a task can specify a period during which it should be held in the Blocked state to wait for data to become available from the queue if it finds the queue is already empty.

Again the task is blocking on a queue event and will leave the Blocked state automatically when another task or interrupt writes data to the queue.

If more than one task is blocked waiting for the same event then the task unblocked upon the occurrence of the event is the task that has the highest priority. Where more than one task of the same priority is blocked waiting for the same event then the task unblocked upon the occurrence of the event will be the task that has been in the Blocked state for the longest time.

Binary Semaphores

Semaphores are a means for a task to signal that it wishes to have exclusive access to data or other resources. While

the task 'has' the semaphore other tasks know they are excluded from accessing the protected resource.

To be permitted access to the resource the task must first 'take' the semaphore, and, when it has finished with the resource, 'give' the semaphore back. If it cannot 'take' the semaphore it knows the resource is already in use by another task and it must wait for the semaphore to become available. If a task chooses to enter the Blocked state to wait for a semaphore it will automatically be moved back to the Ready state as soon as the semaphore is available.

SAFE**RTOS**® includes API functions which fully support binary semaphores. To keep the code size small the semaphore implementation makes use of the queue primitive.

A binary semaphore can be considered to be a queue that can contain, as a maximum, one item. For efficiency the item size can be zero, thus preventing any data actually being copied into and out of the queue. The important information is whether or not the queue is empty or full (the only two states as it can only contain one item), not the value of the data it contains.

When the resource is available, the queue (representing the semaphore) is full. To 'take' the semaphore the task simply receives from the queue – resulting in the queue being empty. To 'give' the semaphore the task simply sends to the queue, resulting in the queue again being full. If, when attempting to receive from the queue, it finds the queue is already empty, a task knows it cannot access the resource and can choose whether or not it wishes to enter the Blocked state to wait for the resource to become available again.

Counting Semaphores

Counting semaphores are implemented in a similar fashion to Binary semaphores. Underlying the counting semaphore APIs SAFE**RTOS** makes use of the queue primitives to achieve an efficient design.

Counting semaphores have a maximum limit on the depth of their queue. As long as there are items in the queue, the resource is available. As with binary semaphores, once the count reaches zero, the resource is unavailable, and the task can choose whether or not it wishes to enter the Blocked state to wait for the resource to become available again.

Communication between Tasks and Interrupts

SAFE**RTOS** provides an alternative API for communication and synchronisation between Interrupt Service Routines and Tasks.

Mutexes

Mutexes are Binary Semaphores that include a priority inheritance mechanism. Binary Semaphores are the better choice for implementing synchronization (between Tasks or between Tasks and an Interrupt), Mutexes are the better choice for implementing simple mutual exclusion (hence 'MUT'ual 'EX'clusion).

When used for mutual exclusion the Mutex acts like a token that is used to guard a resource. When a Task wishes to access the resource it must first obtain ('take') the token. When it has finished with the resource it must 'give' the token back - allowing other Tasks the opportunity to access the same resource.

Mutexes also permit a block time to be specified. The block time indicates the maximum number of 'ticks' that a Task should enter the Blocked state for when attempting to 'take' a Mutex if the Mutex is not immediately available. Unlike Binary Semaphores however, Mutexes employ priority inheritance. This means that if a high priority Task blocks while attempting to obtain a Mutex (token) that is currently held by a lower priority Task, then the priority of the Task holding the token is temporarily raised to that of the blocking Task. This mechanism is designed to ensure the higher priority Task is kept in the blocked state for the shortest time possible, and in so doing minimises the 'priority inversion' that has already occurred.

Priority inheritance does not cure priority inversion! It just minimises its effect in some situations. Hard real time applications should be designed so that priority inversion does not happen in the first place.

Recursive Mutexes

A Mutex used recursively can be 'taken' repeatedly by the owner. The Mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful xSemaphoreTakeRecursive() request. For example, if a Task successfully 'takes' the same Mutex five times then the Mutex will not be available to any other Task until it has also 'given' the Mutex back exactly five times.

This type of Semaphore uses a priority inheritance mechanism so a Task 'taking' a Semaphore must always 'give' the Semaphore back once the Semaphore is no longer required.

Mutex type Semaphores cannot be used from within interrupt service routines.

Event Flags

Event Flags are used to inform tasks of the occurrence of Events. A Task can be woken either by a single Event or a combination of Events from the same Event Group. An Event Group is constructed from a set of Event Flags.

Event Group API functions are provided that allow a Task to set or clear one or more Event Flags within an Event Group, and pend (enter the Blocked state so the Task does not consume any processing time) to wait for a Group of one or more Event Flags to become set within an Event Group.

Event Groups can also be used to synchronise Tasks, creating what is often referred to as a Task 'rendezvous'. A

Task synchronisation point is a place in application code at which a Task will wait in the Blocked state (not consuming any CPU time) to wait for all the other Tasks taking place in the synchronisation to also reach their synchronisation point.

Task Notifications

The Task Notifications feature provides a lightweight alternative to using Queues, Semaphores and Event Groups. Task Notifications are suitable where only one Task consumes the information being provided and offer significant performance and RAM benefits.

Each RTOS Task has a 32-bit notification value. An RTOS Task Notification is an event sent directly to a Task that can unblock the receiving Task, and optionally update the receiving Task's notification value. Task Notifications can update the receiving Task's notification value in the following ways:

- Set the receiving Task's notification value without overwriting a previous value
- Overwrite the receiving Task's notification value
- Set one or more bits in the receiving Task's notification value
- Increment the receiving Task's notification value

This flexibility allows Task Notifications to be used where previously it would have been necessary to create a separate Queue, Binary Semaphore, Counting Semaphore or Event Group. Unblocking an RTOS Task with a direct Notification is up to 45% faster and uses less RAM than unblocking a Task with a Binary Semaphore.

Notifications are sent using the xTaskNotifySend() and xTaskNotifySend() API functions (and their Interrupt safe equivalents), and remain pending until the receiving RTOS Task calls either of the xTaskNotifyWait() or ulTaskNotifyWait() API functions. If the receiving RTOS Task was already Blocked waiting for a Notification then when one arrives the receiving RTOS Task will be removed from the Blocked state and the Notification cleared.

Spatial Separation of Tasks

SAFE**RTOS**® supports the definition and manipulation of MPU regions on a per task basis. This feature provides the tools allowing developers to add a degree of spatial separation between tasks, which used effectively, can help prevent tasks directly making unintentional or accidental access to incorrect memory regions.

The MPU implementation is tightly coupled to the selected processor core and provides a means to establish access permissions for regions of memory. The actual number of memory regions allowed, the size of the regions and addressing is processor dependent. Each region has access permissions which are also heavily processor dependent. Code execution can be allowed or disallowed for a region. A region can be set for read-only access, read/write access,

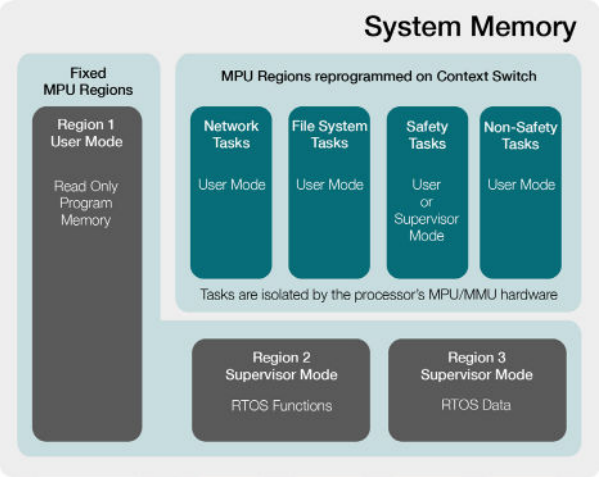

Figure *MPU Task Isolation*

or no access for both privileged and user modes.

The kernel code and data regions are allocated to memory with privileged rights. The remaining MPU regions are allocated to Tasks, and these can be set at the appropriate access level (user or privileged). MPU memory regions relating to Tasks are reprogrammed at each context switch, enabling each Task to have an individual memory access profile.

Once the memory regions are assigned, the MPU and the processors Memory Manage Fault handler is enabled. An access violation of a memory region will cause a Memory Manage Fault, and the processor fault handler will be activated.

If the processor only supports a MMU, where applicable, MMU will be configured to behave as a MPU.

Task Local Storage Pointers

Thread local storage (or TLS) allows the application writer to store values inside a Task's control block, making the value specific to (local to) the Task itself, and allowing each Task to have its own unique value.

Thread local storage is most often used to store values that a single threaded program would otherwise store in a global variable. For example, many libraries include a global variable called errno. If a library function returns an error condition to the calling function, then the calling function can inspect the errno value to determine what the error was. In a single threaded application it is sufficient to declare errno as a global variable, but in a multi-threaded application each thread (Task) must have its own unique errno value - otherwise one Task might read an errno value that was intended for another Task.

Error Checking

SAFE**RTOS** performs thorough API input validity checking (as far as practically possible) in order to mitigate the risk of misuse by the host application. If the value of an API parameter is found to be invalid, the API function will not perform any action other than returning an error code

email: sales@highintegritysystems.com
web: www.highintegritysystems.com

indicative of the error encountered.
SAFE**RTOS** performs the following run time integrity checking with the intention of facilitating the detection of data corruption:

- The execution context of a task that is not in the Running state is stored on the stack allocated to the task. The context of a task will only be saved onto the stack of the task if there is sufficient stack space remaining to hold the entire context;

- A check is performed to ensure that the Task Control Block associated with the task selected to enter the Running state is valid. This is achieved by checking key data parameters against their mirror copies;
- Prior to incrementing the tick count value, a check is performed to test that the current tick count value remains at the last written and therefore expected value.

- SAFE**RTOS** variants that support a MPU implementation will verify the correct processor modes and privilege levels are restored.

A failure in any of these integrity checks will result in a call to the Application Error Hook function call.

Hook Functions

The host application is required to provide one hook (or callback) function with three further optional hook functions.

Table *Hook Functions*

Hook Function	Description
Application Error Hook	Called upon the detection of a fatal error – either a corruption within the scheduler data structures or a potential stack overflow while performing a context switch. The Application Error Hook enables the host application to perform application specific error handling to ensure the system is placed into a safe state. The Error Hook function is required.
Application Task Delete Hook	Called when a task is deleted. Its purpose is to inform the host application that the memory allocated by the application for use by the task is once again free for use for other purposes.
Application Idle Hook	Called repeatedly by the scheduler idle task to allow application specific functionality to be executed within the idle task context. It is common to use the idle task hook to perform low priority application specific background tasks, or simply put the processor into a low power sleep mode.
Tick Hook	The Tick Hook function is called on each execution of the Tick handler to allow application specific functionality to be executed on a periodic basis. It is possible to use the Tick Hook to implement an application timer.
SetupTimerInterrupt Hook	The SetupTimerInterrupt Hook allows an alternative timer to be used to generate the SAFE**RTOS** tick interrupt. The hook function is invoked to initialise and start the desired timer peripheral instead of the default implementation.

Certification & Development Assurance

SAFE**RTOS** for DO-178C is developed in accordance with a structured RTCA DO-178C compliant lifecycle, following a controlled V model (figure) and is fully traceable from requirements through to verification. All design and verification artefacts are independently reviewed and quality assured.

The Development Lifecycle has been divided into 4 distinct project phases (as seen in figure *Delivery Milestones*):

Certification Planning Phase
Defines the certification strategy, scope, and deliverables. Includes WHIS standard certification plans and a project-specific Plan for Software Aspects of Certification (PSAC). The definition of the Software Components to be developed and prototype software.

Design Phase
The development of the requirements, design and code.

Verification Phase
The verification of the software against the requirements and design.

Accomplishment Phase
Finalises the certification evidence set.

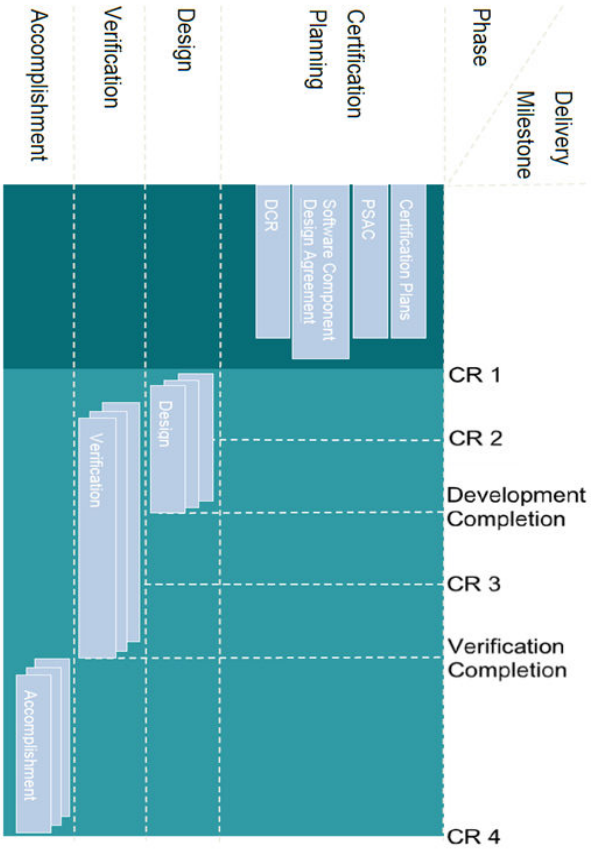

Figure *Delivery Milestones*

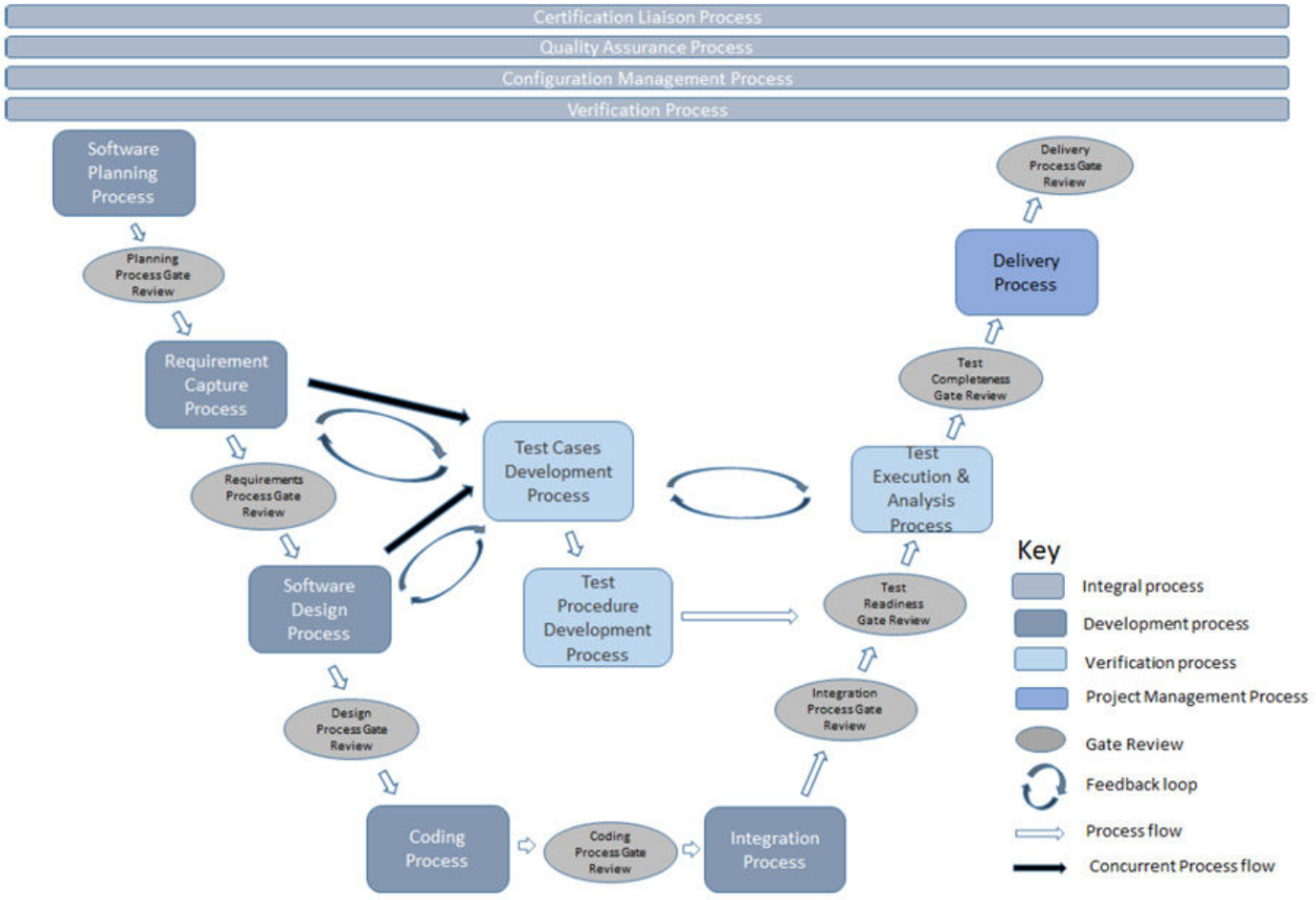

Figure *V Model for Software Development Lifecycle*

email: sales@high**integrity**systems.com
web: www.high**integrity**systems.com

SOI Audits and Compliance Reviews

The development lifecycle is monitored through four formal Stage of Involvement (SOI) audits or Compliance Reviews (CR) aligned to DO-178C expectations. The SOI/CRs will be led by the licensees Designated Engineering Representative and attended by the relevant quality representatives and software engineers from both companies.

SOI/CR 1
Planning Review, held at the end of the Certification Planning Phase. Verifies that the Certification Plans and Certification Strategies are adequate before implementation begins.

SOI/CR 2
Development/Design Review, typically held around 50% completion of the design phase. Confirms if the requirements, design and code are being developed according to the certification plans.

SOI/CR 3
Verification Review, typically held around 50% completion of the verification phase. Evaluates if the verification process is following the certification plans.

SOI/CR 4
Final Compliance Review, held at the end of the Accomplishment Phase. Final review of all deliverables and SOI/CR findings.

All major findings need to be resolved before the SOI/CR can be successfully passed. Minor findings must be resolved before the next SOI/CR.

DO-178C Design Assurance Pack (DAP)

SAFE**RTOS** is supplied with a comprehensive Design Assurance Pack, containing all artefacts required to satisfy DO-178C objectives. The DAP provides complete lifecycle data, traceability, and certification evidence:

Planning Artefacts
- Plan for software aspects of certification
- Software development plan
- Software configuration management plan
- Software quality assurance plan
- Tool qualification plan
- Software requirements standards
- Software design standards
- Software code standards
- Software component design agreement detailing scope, functionality and API of all software components
- Prototype software

Design Artefacts
- Software requirements
- Software architecture description
- Software detailed design descriptions
- Source Code
- Users manual
- Traceability report
- Software configuration index

Verification Artefacts
- Software verification cases and procedures
- Software verification results report
- Timing and memory report
- Errata analysis report
- Structure data and control coupling report
- Structural statement coverage report
- Structural decision coverage report
- MC/DC report
- Source to object code report
- Full source code and verified binary libraries
- Test harness

Final Artefacts
- Safety manual
- Software lifecycle environment configuration index
- Software configuration index
- Software accomplishment Summary
- SQA and CM records

SAFE**RTOS Configuration**

SAFE**RTOS** is licensed for a Processor/Complier combination for use in either a product, or multi product or corporation wide use.

Supplied with a full, royalty free production license:
- No limit on production units.
- Unlimited number of developers from the designated Development Unit.
- SAFE**RTOS** is tool and compiler agnostic, choose your preferred tool chain.

For those starting development with FreeRTOS, it is easy to migrate from FreeRTOS to SAFE**RTOS.**

SAFE**RTOS** was originally built as a complementary offering to the FreeRTOS kernel, with common functionality, but has been fully redesigned for safety critical implementation satisfying the requirements of DO-178C.

WHIS also supplies aerospace qualified UDP stack and Board Support Packages, please contact a WHIS representative for more information.

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600