SAFE**RTOS**® Datasheet

SAFERTOS® is a highly popular, safety certified, real time kernel for microcontrollers, delivering superior performance and pre-certified dependability while using minimal resources.

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.
- Tasks can block for a fixed period.
- Tasks can block to wait for a specified time.
- Tasks can block with a specified timeout period to wait for events.
- Software timers.
- FPU support.
- Definition and manipulation of MPU regions on a per task basis.
- Run time statistics.

Compact Footprint

Typical ROM Requirements	6-15kB
Typical RAM Requirements	500 bytes
Typical Stack Requirements	400 bytes/task

Key Features

IEC 61508 SIL3 certified

IEC 62304 Class C compliant

Full Design Assurance Pack

Supports popular 32 bit microcontrollers

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'.
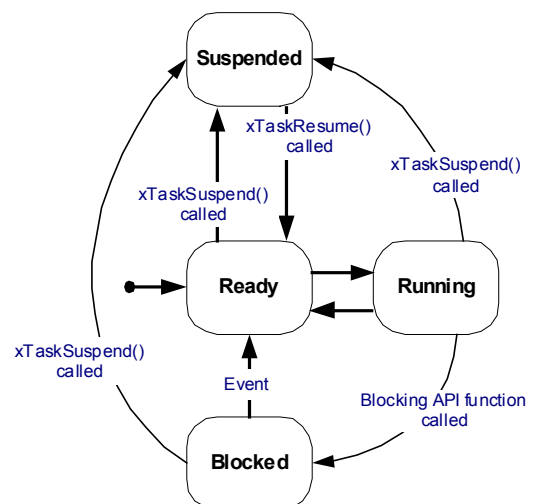

Figure *Valid Task State Transitions*

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.
Suspended	A task will enter the Suspended state when it is the subject of a call to the xTaskSuspend() 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'.
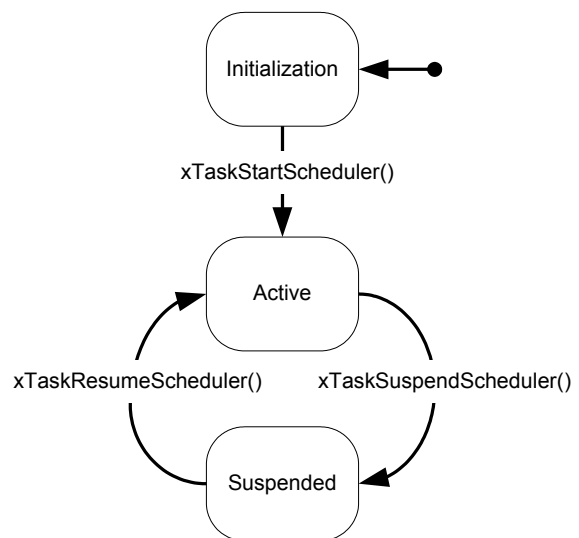
Figure *Valid Scheduler State Transitions*

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.

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 xTaskSuspendScheduler(), 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 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 xTaskSuspendScheduler(), 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 behavior.

Queue Characteristics

The following bullet points summarize the queue implementation:

- 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 a set of interrupt safe API calls for Queue and semaphore handling from within interrupt service routines

MPU Protection

The Memory Protection Unit (MPU) allows software of different Safety Integrity Levels (SIL) to co-exist in a single build of code without unwanted mutual interference.

Figure *MPU Task Isolation*

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, or no access for both privileged and user modes. This can be used to set up an environment where only kernel or system code can access certain hardware registers or sections of code.

SAFE**RTOS** supports the definition and manipulation of MPU regions on a per task basis, where each task is assigned to specific memory regions.

Once the regions are defined, the MPU is enabled and the processors Memory Manage Fault handler is enabled. Any access violation of a region will cause a Memory Manage Fault, and the processor fault handler will be activated. The fault handler is invoked prior to the actual memory access.

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.

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 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;

- When the context of a task is restored, the setting of the Processor Status Register (PSR) is checked against its expected value;

- 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.

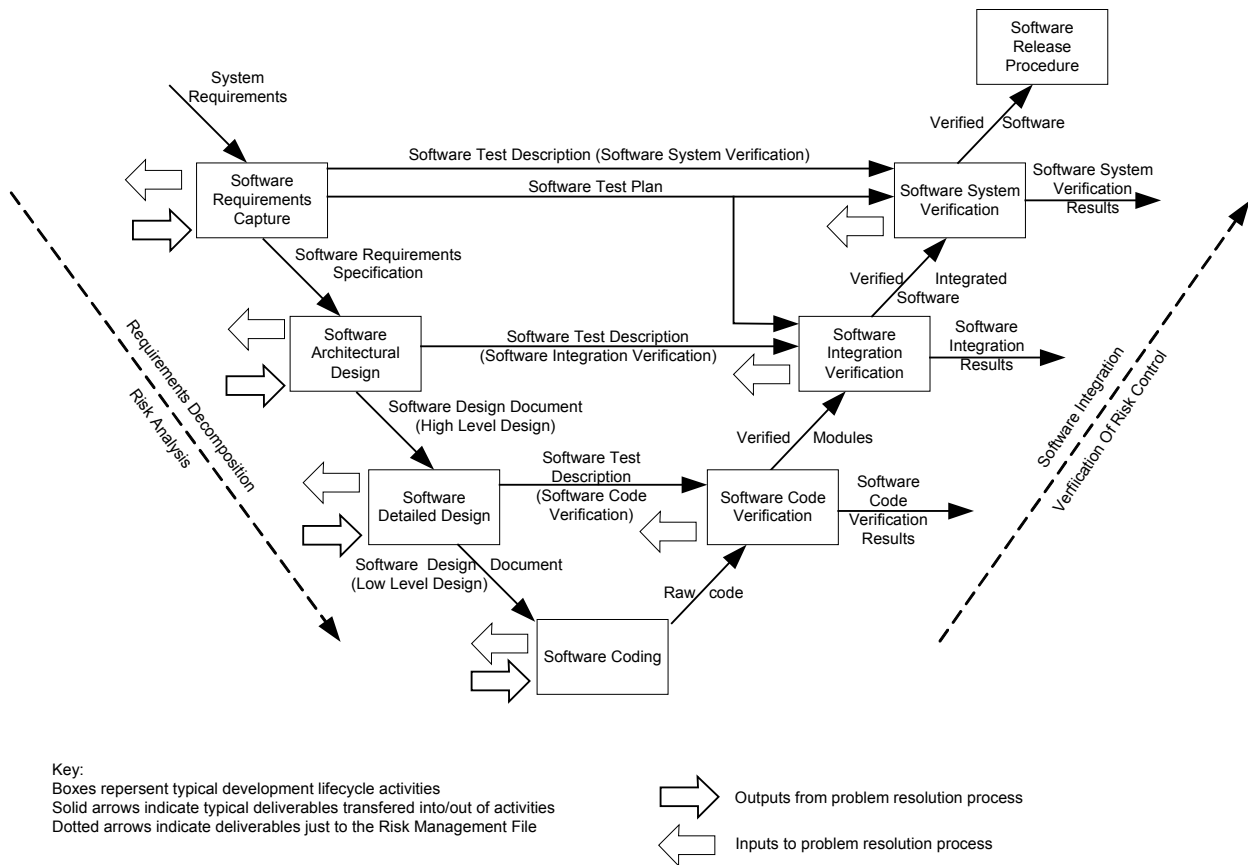A failure in any of these integrity checks will result in a call to the Application Error Hook function call.

Figure *The SAFERTOS Development Life Cycle*

Development Life Cycle

SAFE**RTOS** was built as a complementary offering to FreeRTOS™, with common functionality but with a uniquely designed safety critical implementation.

The FreeRTOS functional model was subjected to a full HAZOP and all weaknesses within the functional model and API were identified and resolved. The resulting requirements set was put through a full IEC 61508 SIL3 development life cycle, the highest possible for a software only component.

The code base has been developed to conform to the majority of the MISRA guide lines, with a few additional rules of our own for extra safety.

Underpinning the SAFE**RTOS** development life cycle is our requirements management process. We have achieved 100% traceability across the complete SAFE**RTOS** project, from the very first customer requirement all the way through to the very last validation test result. The 100% traceability is an important element when taking a product through certification, as it demonstrates the completeness of the design.

Design Assurance Pack

SAFE**RTOS** is supplied with a Design Assurance Pack (DAP) which contains every design artefact produced during the full development life cycle, from development and safety life cycle plans, requirements specifications and design documents, to HAZOPS, the source code, all verification and validation documents and relating evidence.

The full test harness, with User and Safety manuals, is also supplied. Customers who purchase SAFE**RTOS** have all the information required to independently complete their own product certification.

The SAFE**RTOS** Safety Manual clearly identifies each and every component included within the SAFE**RTOS** delivery, and their related checksums. The safety manual contains a concise list of instructions clearly identifying the installation and integration process engineers should follow when incorporating SAFE**RTOS** into their development environment. Should additional assistance be required our experienced engineers can provide support.

The DAP is accessed via a graphical interface containing hyperlinks into each individual document.

Figure *SAFERTOS DAP Graphical Interface*

The DAP contains the following documents:

- SAFE**RTOS** User Manual: Provides an overview of SAFE**RTOS** and gives a description of the SAFE**RTOS** task, queue and scheduling mechanisms, describes the installation and setup procedure, provides an API reference.

- SAFE**RTOS** Safety Manual: Contains a concise list of instructions clearly identifying the installation and integration process engineers should follow when incorporating SAFE**RTOS** into their development environment.

- Upgrading from FreeRTOS to SAFE**RTOS** Application Note: Highlights the areas requiring modification when moving an application from FreeRTOS to SAFE**RTOS**.

- Demonstration Program: "Out of the box" examples configured for the selected toolchain demonstrating the features of SAFE**RTOS**.

- Software Version Description: Contains an inventory of the materials released, their related checksums and a record of the changes made to this deliverable over its lifetime.

- Software Development Plan: Defines the IEC 61508 SIL 3 compliant development life cycle used in the development of this SAFE**RTOS** delivery.

- Configuration Management Plan: Gives an overview of our configuration management tool, identifies the items under configuration control, and the configuration management rules and working procedures used in the development of this SAFE**RTOS** delivery.

- Software Test Plan: Defines the objectives for each Verification and Validation (V&V) phase, and the related test environment. Defines the complete V&V schedule.

- Software Safety Management Plan: The objective of the Software Safety Management Plan is to adequately justify that SAFE**RTOS** meets its high level safety requirements. This SSMP forms the plan for ensuring that safety is considered throughout the SAFE**RTOS** development programme and is appropriately designed into SAFE**RTOS**.

- Customer Requirements Specification: Defines the customer requirements for the SAFE**RTOS** product.

- Software Requirements Specification: Defines the software requirements for SAFE**RTOS** with respect to its functional requirements and qualification methods.

- Architecture Software Design Description: Describes the architectural software design for the SAFE**RTOS** product with respect to its structure and design constraints.

- Detailed Software Design Description: Describes the detailed software design for the SAFE**RTOS** product with respect to the individual modules.

- HAZOP Reports: This document contains the SAFE**RTOS** Hazard and Operability Study, the hazard assessment, risk reduction methods, the safety related requirements and details of any residual risks.

- API Usage Safety Review: Analyses the functions and macros that constitute the API to determine any actual or potential behavior of the SAFE**RTOS** API that could lead to unsafe or inappropriate use by a user or other interested person.

- Software Test Description: Details every test step, within each test case, by its inputs, expected outputs and operating instructions.

- Kernel Source Code: Full source code and build files provided in C and Assembler format.

- Test harness: Full source code and build files provided for the test harness used to verify the SAFE**RTOS** product.

- Test Harness Build Procedure: Provides instructions on building and using the software test harness when performing formal testing of the product.

- Individual Test Logs: The actual test result log files.

- Software Test Report: The Software test report gives an overview of the results from the V&V process, before going into the details of each test case.

- Evidence Supporting IEC 61508-3 SIL3 Claim: The claim collates and references all evidence that supports the claim that SAFE**RTOS** requirements have been specified and met and that it has been developed and certified to meet the software development requirements of IEC 61508-3 Safety Integrity Level (SIL) 3". It uses a Claims, Argument and Evidence approach.

- IEC 61508 Compliance Matrix: Provides Cross-references from IEC 61508: 2010 into the SAFE**RTOS** development life cycle.

SAFE**RTOS** Configuration

SAFE**RTOS** is licensed as a SAFE**RTOS** variant, where a variant is defined according to the developer's choice of micro-processor and tool chain. A robust RTOS that inherently has less risk - the API and the core SAFE**RTOS** design and code is common between all SAFE**RTOS** variants; the remaining port layer is adapted to support the selected micro-processor. Each SAFE**RTOS** variant is subjected to the full IEC 61508 compliant development life cycle.

Supported Processors

SAFE**RTOS** has been specifically designed for the 32 bit micro-controller market. No allowances or compromises have been incorporated to support other platforms. This has resulted in a highly optimized design and minimal resource footprint, providing a concise feature set.

Certification

SAFE**RTOS** can be licensed as a pre-certified IEC 61508 SIL3 software component.

SAFE**RTOS** was initially certified by TÜV SÜD in 2007, resulting in the world's first ever pre-certified RTOS.

SAFE**RTOS** is suitable for inclusion in developments requiring certification against IEC 62304 Class C, IEC 61508 SIL3 and domain adaptations of IEC 61508.

License Model

SAFE**RTOS** is royalty free, with flexible licensing models. Please give one of our representatives a call today to discuss which license model best suits your needs.

WITTENSTEIN high **integrity** systems

Worldwide Sales and Support
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@highintegritysystems.com
Web: www.high**integrity**systems.com