GeMRTOS

by R. Cayssials

GeMRTOS: Multiprocessor RTOS

API reference

Date: October 30, 2024

Revision: 1.0

Contents

GeMRTOS API Functions categories

Critical Section category

The Critical Section category in GeMRTOS provides essential macros for protecting shared data structures during concurrent task execution. These macros are specifically designed to manage access to GeMRTOS data structures, ensuring that operations on shared resources are executed atomically to prevent race conditions and maintain data integrity. Proper implementation of these critical section macros is crucial for protecting GeMRTOS-specific data structures from unexpected behaviors that may arise from simultaneous access by multiple tasks or processors. This category empowers developers to create secure and efficient real-time applications by ensuring that critical operations on data structures are performed without interruption or conflict.

gm_GeMRTOSCriticalSectionEnter
gm_GeMRTOSCriticalSectionExit
gm_GeMRTOSCriticalSectionGrantedTime

Frozen Mode category

The Frozen Mode category in GeMRTOS includes functions and macros that manage system behavior during critical operational states. This mode allows the system to temporarily suspend certain tasks and processes to preserve resources and maintain stability under specific conditions. By entering Frozen Mode, developers can prevent unwanted interruptions and manage timing more effectively, particularly in scenarios that require precise timing or resource allocation. Functions related to Frozen Mode enable the configuration of thresholds and control mechanisms, allowing for efficient activation and deactivation of this mode as needed. This capability is essential for optimizing performance and ensuring system reliability in real-time applications.

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_FrozenModeThresholdGet
gm_FrozenModeThresholdSet
gm_FrozenModeTimeGet

IRQ Management category

The IRQ Management category in GeMRTOS encompasses functions and macros designed to handle interrupt requests (IRQs) efficiently within the real-time operating system. This category provides essential tools for enabling, disabling, and managing interrupts, allowing tasks to respond promptly to external events and system signals. Effective IRQ management is crucial for optimizing system performance, as it facilitates the prioritization of tasks and ensures that critical events are

addressed in a timely manner.
gm_IrqDisable
gm_IrqEnable
gm_IrqStatusGet

Message Queue category

The Message Queue category in GeMRTOS provides essential functions and macros for implementing inter-task communication through message passing mechanisms. This category enables tasks to exchange data and synchronize their operations efficiently, facilitating seamless collaboration within a real-time system.
By utilizing the Message Queue category, developers can create, send, and receive messages between tasks, allowing for asynchronous communication that enhances system responsiveness. The functions within this category support various operations, including message queue creation, message enqueuing and dequeuing to ensure robust data exchange.
With the capability to configure message priorities and handling, the Message Queue category not only streamlines communication but also aids in managing task dependencies and resource sharing. This is particularly important in complex applications where timely and reliable message transfer is critical. By ensuring effective inter-task communication, the Message Queue category plays a vital role in optimizing performance and contributing to the overall reliability of the GeMRTOS environment.
gu_MessageQueueCreate
gu_MessageQueuePrintf
gu_MessageQueueReceive
gu_MessageQueueSend
gu_MessageQueueSubscribe

Processor category

The Processor category in GeMRTOS includes functions and macros that provide critical tools for managing processor-level operations and configurations within the real-time operating system. This category facilitates the control of individual processors, allowing developers to optimize task scheduling, interrupt handling, and overall system performance. Functions in this category enable manipulation of processor states, including halting, resuming, and managing processor interrupts, as well as retrieving processor-specific information. The Processor category is essential for developing robust real-time applications that require precise control over processing resources, promoting responsiveness, and achieving effective synchronization between tasks and hardware components.
gm_ProcessorHalt
gm_ProcessorId
gm_ProcessorInterrupt
gm_ProcessorInterruptDisable
gm_ProcessorInterruptEnable

gm_ProcessorWaitForIrq

Scheduling List category

The Scheduling List category in GeMRTOS encompasses functions and macros designed to facilitate the management and manipulation of hybrid scheduling lists within the real-time operating system. These tools provide essential data structure capabilities that allow developers to create, modify, and traverse collections of tasks efficiently, enabling dynamic and flexible scheduling approaches.
By using the Scheduling List category functions, developers can implement effective algorithms for task prioritization, resource allocation, and event handling, all while maintaining high performance and minimal overhead. The functions within this category support various operations such as adding and removing tasks, adjusting priorities, and specifying scheduling criteria. Notably, the configuration of scheduling list exclusions can help prevent real-time anomalies, ensuring that critical tasks receive the attention they need while balancing processor loads effectively.
Integration of the Scheduling List category into real-time applications enhances task organization and scheduling efficiency, enabling the system to respond rapidly to changes in workload and processor availability. This capability is critical for applications where timing, responsiveness, and resource management are paramount.
gm_SchedulingListExclusionSectionEnter
gm_SchedulingListExclusionSectionExit
gu_SchedulingListAssociateProcessor
gu_SchedulingListAssociateTask
gu_SchedulingListCreate
gu_SchedulingListExclusionSet

Semaphore category

The Semaphore category in GeMRTOS encompasses functions and macros designed to facilitate synchronization and resource management among concurrent tasks within the real-time operating system. Semaphores are essential for controlling access to shared resources, preventing race conditions, and ensuring data integrity by regulating how tasks interact with one another.
By utilizing the Semaphore category, developers can create and manage both binary and counting semaphores, allowing for fine-grained control over task execution and resource allocation. The functions within this category enable operations such as semaphore creation, and waiting, effectively coordinating task activities and synchronizing their behavior.
The use of semaphores is crucial in environments where multiple tasks need to access shared resources without conflict, as it helps maintain system stability and performance. Additionally, by leveraging semaphores, developers can enhance the efficiency of their applications, ensuring that critical tasks are executed in a timely manner while preventing task starvation and optimizing resource utilization.
gu_SemaphoreCreate
gu_SemaphorePost

gu_SemaphoreWait

Signal category

The Signal category in GeMRTOS provides essential functions and macros for implementing event-driven synchronization mechanisms between tasks within the real-time operating system. Signals serve as lightweight notification tools that allow tasks to communicate important state changes, alerts, or operational events efficiently.
By utilizing the Signal category, developers can create and manage signals that facilitate asynchronous task coordination, enabling tasks to respond promptly to specific events without polling or constant checking. Functions within this category support operations such as signal creation, allowing tasks to seamlessly be notified when critical actions need to take place.
gu_SignalCreate
gu_SignalDestroy

System category

The System category in GeMRTOS encompasses critical functions and macros that provide core capabilities for managing and configuring the operating environment. This category is vital for overseeing system-level operations, resource management, and overall application behavior within the real-time operating system.
By utilizing the System category, developers can access functions that facilitate system initialization, configuration of kernel parameters, and management of system states.
gm_SystemTimePrescaleGet
gm_SystemTimePrescaleSet
gm_SystemTotalTimeGet
gm_WriteOutputs
gu_fprintf
gu_printf

Task category

The Task category in GeMRTOS includes essential functions and macros for creating, managing, and scheduling tasks within the real-time operating system. This category is fundamental for implementing multitasking, allowing applications to perform multiple operations concurrently and efficiently utilize system resources.
By leveraging the Task category, developers can create tasks with specified priority levels, resource requirements, and execution parameters, enabling fine control over how tasks are executed and scheduled. Functions within this category support a wide range of operations, including task creation, and suspension, as well as priority management.
The flexibility offered by the Task category supports responsive applications that can adapt to

dynamic conditions in real-time environments. The Task category is crucial for building robust, efficient, and responsive applications in the GeMRTOS ecosystem, facilitating the seamless management of concurrent operations in complex real-time systems.
gu_TaskCreate
gu_TaskDelay
gu_TaskDelayTime
gu_TaskGetCurrentTCB
gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet
gu_TaskResume
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskSuspend
gu_TaskTypeSet

Trigger category

The Trigger category in GeMRTOS encompasses functions and macros that facilitate event-driven mechanisms within the real-time operating system. These functions enable tasks to respond to specific events, interrupts, or conditions, enhancing the system's interactivity and responsiveness. Triggers play a crucial role in synchronization, allowing tasks to be activated based on the occurrence of defined events, thereby optimizing resource utilization and improving overall system efficiency.
gu_TriggerCreate
gu_TriggerDisable
gu_TriggerDisableHook
gu_TriggerEnable
gu_TriggerEnableHook
gu_TriggerRegisterTask
gu_TriggerRelease
gu_TriggerWait

GeMRTOS Control Blocks definitions

---

struct g_rcb
struct T_QUEUE_RESOURCE
struct gs_ecb
struct gs_tcb
struct gs_lcb
struct T_SEMAPHORE_RESOURCE

struct gs_scb

GeMRTOS Enumeration definitions

enum lcbtype
enum scbtype
enum tcbtype

GeMRTOS Functions

gm_GeMRTOSCriticalSectionEnter

Prototype

gm_GeMRTOSCriticalSectionEnter;

Description

The gm_GeMRTOSCriticalSectionEnter macro defines the entry point into a critical section for the management of kernel data. It is designed to ensure that modifications to shared kernel resources occur safely, although it may be interrupted while waiting for the GeMRTOS controller mutex. This macro should be utilized whenever there is a need to modify kernel data to prevent data corruption and maintain system stability.

Parameters

The gm_GeMRTOSCriticalSectionEnter macro does not accept any parameters.

Returns

The gm_GeMRTOSCriticalSectionEnter macro does not return any value but blocks the code execution until the GeMRTOS controller mutex is granted.

See also

gm_GeMRTOSCriticalSectionExit, gm_GeMRTOSCriticalSectionGrantedTime, gm_IrqDisable, gm_IrqEnable, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

gm_GeMRTOSCriticalSectionExit

Prototype

gm_GeMRTOSCriticalSectionExit;

Description

The gm_GeMRTOSCriticalSectionExit macro exits the critical section from the current processor, allowing other processes enter. It is essential to use this macro in all user functions that execute kernel functions or modify kernel data, ensuring that the critical section is properly released and preventing potential deadlocks or resource contention.

Parameters

The gm_GeMRTOSCriticalSectionExit macro does not accept any parameters.

Returns

The gm_GeMRTOSCriticalSectionExit macro does not return any value.

See also

gm_GeMRTOSCriticalSectionEnter, gm_GeMRTOSCriticalSectionGrantedTime, gm_IrqDisable, gm_IrqEnable, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

---

gm_GeMRTOSCriticalSectionGrantedTime

Prototype

gm_GeMRTOSCriticalSectionGrantedTime;

Description

The gm_GeMRTOSCriticalSectionGrantedTime macro returns the time the mutex was granted in system time units.

Parameters

The gm_GeMRTOSCriticalSectionGrantedTime macro does not accept any parameters.

Returns

The gm_GeMRTOSCriticalSectionGrantedTime macro returns the time the mutex was granted in system time units.

See also

gm_GeMRTOSCriticalSectionEnter, gm_GeMRTOSCriticalSectionExit

---

gm_FrozenModeDisable

Prototype

gm_FrozenModeDisable;

Description

The gm_FrozenModeDisable macro disables the frozen mode event. By default, the frozen mode starts in a disabled state.

Parameters

The gm_FrozenModeDisable macro does not accept any parameters.

Returns

The gm_FrozenModeDisable macro does not return any value.

See also

gm_FrozenModeEnable, gm_FrozenModeThresholdGet, gm_FrozenModeThresholdSet, gm_FrozenModeTimeGet, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

---

gm_FrozenModeEnable

Prototype

gm_FrozenModeEnable;

Description

The gm_FrozenModeEnable macro enables the frozen mode event. By default, the frozen mode starts in a disabled state. Before invoking this macro, ensure that the frozen threshold is properly configured using the gm_FrozenModeThresholdSet function.

Parameters
The gm_FrozenModeEnable macro does not accept any parameters.
Returns
The gm_FrozenModeEnable macro does not return any value.
See also
gm_FrozenModeDisable, gm_FrozenModeThresholdGet, gm_FrozenModeThresholdSet, gm_FrozenModeTimeGet, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

---

gm_FrozenModeThresholdGet

Prototype
TIMEPRIORITY interval = gm_FrozenModeThresholdGet;
Description
The gm_FrozenModeThresholdGet macro retrieves the value of the Frozen Time Threshold register from the GeMRTOS controller (R_FRZ_TM_THR). By default, the frozen mode is disabled, and the frozen threshold is set to zero. This macro is useful for determining the current threshold value, which is critical for managing the activation of frozen mode.
Parameters
The gm_FrozenModeThresholdGet macro does not accept any parameters
Returns
The gm_FrozenModeThresholdGet macro returns the current value of the Frozen Time Threshold register in the GeMRTOS controller.
See also
gm_FrozenModeDisable, gm_FrozenModeEnable, gm_FrozenModeThresholdSet, gm_FrozenModeTimeGet, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

---

gm_FrozenModeThresholdSet

Prototype
gm_FrozenModeThresholdSet(timeset);
Description
The gm_FrozenModeThresholdSet macro sets the value of the Frozen Time Threshold register in the GeMRTOS controller. By default, the frozen mode is disabled, and the frozen threshold is initialized to zero. This macro is essential for configuring the threshold that determines when the frozen mode becomes active when it is enabled.
Parameters
The gm_FrozenModeThresholdSet macro accepts the following parameter:

- timeset:The frozen threshold value specified in system ticks units. This value establishes the interval of delay in processing timed events after which the frozen mode will be triggered when frozen mode is enabled.

Returns
The gm_FrozenModeThresholdSet macro does not return any value.

See also
gm_FrozenModeDisable, gm_FrozenModeEnable, gm_FrozenModeThresholdGet, gm_FrozenModeTimeGet, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

---

gm_FrozenModeTimeGet

Prototype
G_INT64 time = gm_FrozenModeTimeGet;

Description
gm_FrozenModeTimeGet returns the accumulated time the system was in Frozen Mode. This time is hold in the R_FRZ_CNT register of the GeMRTOS controller.

Parameters
The gm_FrozenModeTimeGet macro has no parameter.

Returns
The gm_FrozenModeTimeGet macro returns the accumulated time the system was in Frozen Mode, hold in the the R_FRZ_CNT register of the GeMRTOS controller.

See also
gm_FrozenModeDisable, gm_FrozenModeEnable, gm_FrozenModeThresholdGet, gm_FrozenModeThresholdSet

---

gm_IrqDisable

Prototype
gm_IrqDisable(irq);

Description
The gm_IrqDisable macro disables the specified device interrupt request event (IRQ) in the GeMRTOS controller. This macro is essential for managing interrupt handling and preventing specified DIRQs from triggering.

Parameters
The gm_IrqDisable macro accepts the following parameter:

- irq:The number of the IRQ to be disabled.

Returns
The gm_IrqDisable macro does not return any value.

See also
gm_IrqEnable, gm_IrqStatusGet, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

---

gm_IrqEnable

Prototype
gm_IrqEnable(irq);

Description
The gm_IrqEnable macro enables the specified device interrupt request event (DIRQ) in the GeMRTOS controller. This macro is crucial for allowing specified DIRQ to trigger.

Parameters

The gm_IrqEnable macro accepts the following parameter:

- irq:The number of the DIRQ to be enabled.

Returns

The gm_IrqEnable macro does not return any value.

See also

gm_IrqDisable, gm_IrqStatusGet, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

---

gm_IrqStatusGet

Prototype

G_INT32 status = gm_IrqStatusGet;

Description

The gm_IrqStatusGet macro retrieves the current DIRQ status from the GeMRTOS controller. It reflects the value of the input signals from the device IRQ inputs, prior to the application of any enabling logic. This macro is useful for determining which interrupt requests are currently active.

Parameters

The gm_IrqStatusGet macro does not accept any parameters.

Returns

The macro returns the status of the DIRQ register of the GeMRTOS controller.

See also

gm_IrqDisable, gm_IrqEnable, gm_ProcessorInterrupt, gm_ProcessorWaitForIrq

---

gu_MessageQueueCreate

Prototype

G_RCB *gu_MessageQueueCreate(void);

Description

The gu_MessageQueueCreate function creates a new message queue resource. This resource is implemented using a G_RCB structure, extended with fields from a T_QUEUE_RESOURCE structure. The created queue includes event lists for producers (waiting to send) and consumers (waiting to receive) messages. Producer tasks add themselves to the producer event list when they are waiting to send a message. This function can be called from either the main application code or from within a task. If called within a task, it must be called before any message send or receive operations; otherwise, an error will occur.

Parameters

The gu_MessageQueueCreate function takes no parameters.

Returns

The gu_MessageQueueCreate function returns a pointer (G_RCB *) to the newly created message queue resource. This pointer is essential for all subsequent operations on the queue. A NULL pointer is returned if there is insufficient memory to create the queue or if no more queue resources are available.

See also

gu_MessageQueuePrintf, gu_MessageQueueReceive, gu_MessageQueueSend, gu_MessageQueueSubscribe

gu_MessageQueuePrintf

Prototype

int gu_MessageQueuePrintf(G_RCB *prcb, char *format, ...);

Description

The gu_MessageQueuePrintf function sends a formatted message to a message queue. The calling task will block until the message is successfully delivered to all consumers subscribed to the queue.

Parameters

The function accepts the following parameters:

- **prcb**: A pointer to the G_RCB structure representing the message queue. This pointer is the value returned by gu_MessageQueueCreate when the queue was created.

- **format**: A null-terminated string containing the format string, similar to the standard printf function. This string can include format specifiers (e.g., d, s, x) that are replaced by subsequent arguments.

Returns

The function returns G_TRUE if the message was successfully sent to the queue, and G_FALSE otherwise.

See also

gu_MessageQueueCreate, gu_MessageQueueReceive, gu_MessageQueueSend, gu_MessageQueueSubscribe

gu_MessageQueueReceive

Prototype

int gu_MessageQueueReceive(G_RCB *prcb, void *buffer_msg, G_INT32 buffer_length);

Description

The gu_MessageQueueReceive function retrieves the next message from a message queue. The calling task must have previously subscribed to the queue using gu_MessageQueueSubscribe. The received message is copied into the buffer specified by buffer_msg. If the message is larger than buffer_length, it will be truncated to fit the buffer.

Parameters

The function takes three parameters:

- **prcb**: A pointer to the G_RCB structure of the message queue from which to receive the message.

- **buffer_msg**: A pointer to the memory buffer where the received message will be stored.

- **buffer_length**: An integer specifying the maximum number of bytes to receive. This should be equal to or greater than the size of the buffer_msg buffer.

Returns

The gu_MessageQueueReceive function returns an integer representing the number of bytes actually received. This value may be less than buffer_length if the received message was shorter than the buffer or if the message was truncated due to buffer size limitations.

See also

gu_MessageQueueCreate, gu_MessageQueuePrintf, gu_MessageQueueSend, gu_MessageQueueSubscribe

---

gu_MessageQueueSend

Prototype

int gu_MessageQueueSend(G_RCB *prcb, char *pmsg, int msg_length, gt_time timeout);

Description

The gu_MessageQueueSend function transmits a message to a message queue. The sending task blocks until the message has been successfully delivered to all subscribed consumers or until a timeout occurs.

Parameters

The function uses the following parameters:

- **prcb**: A pointer to the G_RCB structure representing the message queue resource. This pointer was returned by gu_MessageQueueCreate when the queue was created.
- **pmsg**: A pointer to the message data to be sent
- **msg_length**: An integer representing the length of the message to be sent, in bytes.
- **timeout**: A gt_time value specifying the timeout period for sending the message.

Returns

The gu_MessageQueueSend function returns G_TRUE if the message was successfully sent within the timeout period, and G_FALSE otherwise. G_FALSE indicates either a timeout or another error condition.

See also

gu_MessageQueueCreate, gu_MessageQueuePrintf, gu_MessageQueueReceive, gu_MessageQueueSubscribe

---

gu_MessageQueueSubscribe

Prototype

GS_ECB *gu_MessageQueueSubscribe(GS_TCB *ptcb, G_RCB *presource);

Description

The gu_MessageQueueSubscribe function subscribes the task to a message queue resource. This subscription is crucial for message delivery; a consumer task must be subscribed to a queue before it can receive messages using gu_MessageQueueReceive. Furthermore, the execution of this function for each receiving task allows the message queue to track the number of consumers subscribed. This count is essential for producers; a producer message is only considered fully delivered when it has been received by every subscribed consumer. The message queue must have been previously created using gu_MessageQueueCreate.

Parameters

The function uses the following parameters:

- **ptcb**: A pointer to the Task Control Block (GS_TCB) of the task being subscribed to the queue.
- **presource**: A pointer to the G_RCB structure representing the message queue resource to which the task is subscribing.

Returns

The gu_MessageQueueSubscribe function returns a pointer to the GS_ECB structure associated with the message queue resource. A NULL return value likely indicates an error.

See also

gu_MessageQueueCreate, gu_MessageQueuePrintf, gu_MessageQueueReceive, gu_MessageQueueSend

---

gm_ProcessorHalt

Prototype

gm_ProcessorHalt;

Description

The gm_ProcessorHalt macro places the processor into halt mode, effectively stopping its execution until an interrupt is issued for this processor by the GeMRTOS controller. This mode is often used to conserve power or to wait for external events before resuming normal operation.

Parameters

The gm_ProcessorHalt macro does not require any parameters.

Returns

The gm_ProcessorHalt macro returns control to the calling function when the processor is interrupted and the ISR routine executed, allowing it to resume execution.

See also

gm_ProcessorId, gm_ProcessorInterrupt, gm_ProcessorInterruptDisable, gm_ProcessorInterruptEnable, gm_ProcessorWaitForIrq

---

gm_ProcessorId

Prototype

G_INT32 prcID = gm_ProcessorId;

Description

The gm_ProcessorId macro retrieves the ID of the current processor. This macro is useful for identifying the processor.

Parameters

The gm_ProcessorId macro does not accept any parameters.

Returns

The gm_ProcessorId macro returns the ID of the current processor.

See also
gm_ProcessorHalt, gm_ProcessorInterrupt, gm_ProcessorInterruptDisable, gm_ProcessorInterruptEnable, gm_ProcessorWaitForIrq

---

gm_ProcessorInterrupt

Prototype
gm_ProcessorInterrupt(proc);
Description
gm_ProcessorInterrupt issues an interrupt for the processor with the specified ID and waits until it reaches the ISR and disables its interrupt in the GeMRTOS controller.
Parameters
The gm_ProcessorInterrupt macro accepts the following parameter:

- proc:The ID of the processor to be interrupted. This specifies which processor will receive the interrupt signal.

Returns
The gm_ProcessorInterrupt macro returns when the target processor disables its interrupt in the GeMRTOS controller.
See also
gm_ProcessorHalt, gm_ProcessorId, gm_ProcessorInterruptDisable, gm_ProcessorInterruptEnable, gm_ProcessorWaitForIrq

---

gm_ProcessorInterruptDisable

Prototype
gm_ProcessorInterruptDisable;
Description
The gm_ProcessorInterruptDisable macro disables GeMRTOS processor interrupt in the GeMRTOS controller.
Parameters
The gm_ProcessorInterruptDisable macro does not accept any parameters.
Returns
The gm_ProcessorInterruptDisable macro does not return any value.
See also
gm_ProcessorHalt, gm_ProcessorId, gm_ProcessorInterrupt, gm_ProcessorInterruptEnable, gm_ProcessorWaitForIrq

---

gm_ProcessorInterruptEnable

Prototype
gm_ProcessorInterruptEnable;
Description
The gm_ProcessorInterruptEnable macro enables processor interrupts in the GeMRTOS controller.

Parameters
The gm_ProcessorInterruptEnable macro does not accept any parameters.
Returns
The gm_ProcessorInterruptEnable macro does not return any value.
See also
gm_ProcessorHalt, gm_ProcessorId, gm_ProcessorInterrupt, gm_ProcessorInterruptDisable, gm_ProcessorWaitForIrq

---

gm_ProcessorWaitForIrq

Prototype
gm_ProcessorWaitForIrq(IRQ_mask);
Description
The gm_ProcessorWaitForIrq macro halts the processor until an interrupt occurs on one of the specified masked IRQs. This function is useful for enabling the processor to wait for specific interrupt events.
Parameters
The gm_ProcessorWaitForIrq macro accepts the following parameter:

- IRQ_mask:A mask of the DIRQs that the processor will wait for. This mask specifies which interrupts should wake the processor from its halted state. The interrupt should be disabled in order to be used to wake up the processor.

Returns
The gm_ProcessorWaitForIrq macro does not return any value.
See also
gm_ProcessorHalt, gm_ProcessorId, gm_ProcessorInterrupt, gm_ProcessorInterruptDisable, gm_ProcessorInterruptEnable

---

gm_SchedulingListExclusionSectionEnter

Prototype
gm_SchedulingListExclusionSectionEnter;
Description
The gm_SchedulingListExclusionSectionEnter macro set temporalely the scheduling list exclusion parameter equal to 1 in order to avoid any other task to execute the following critial code. If another processor placed the exclusion before, then the task is suspended until the task executing the critical code restore the scheduling list exclusion parameter to its original value.
Parameters
The gm_SchedulingListExclusionSectionEnter macro does not require any parameters.
Returns
The gm_SchedulingListExclusionSectionEnter macro returns control to the calling function when the exclusion parameter is set to 1.
See also
gm_ProcessorId, gm_ProcessorInterrupt, gm_ProcessorInterruptDisable, gm_ProcessorInterruptEnable, gm_ProcessorWaitForIrq, gm_SchedulingListExclusionSectionExit,

gu_SchedulingListAssociateProcessor, gu_SchedulingListAssociateTask, gu_SchedulingListCreate, gu_SchedulingListExclusionSet

---

gm_SchedulingListExclusionSectionExit

Prototype

gm_SchedulingListExclusionSectionExit;

Description

The gm_SchedulingListExclusionSectionExit macro restores the exclusion parameter of the scheduling list previous to the execution of gm_SchedulingListExclusionSectionEnter. It should executed when the critical code ends. It also enables the processor interrupt in the GeMRTOS controller.

Parameters

The gm_SchedulingListExclusionSectionExit macro does not require any parameters.

Returns

The gm_SchedulingListExclusionSectionExit macro returns control to the calling function when the exclusion parameter is set to 1.

See also

gm_ProcessorId, gm_ProcessorInterrupt, gm_ProcessorInterruptDisable, gm_ProcessorInterruptEnable, gm_ProcessorWaitForIrq, gm_SchedulingListExclusionSectionEnter, gu_SchedulingListAssociateProcessor, gu_SchedulingListAssociateTask, gu_SchedulingListCreate, gu_SchedulingListExclusionSet

---

gu_SchedulingListAssociateProcessor

Prototype

G_INT32 gu_SchedulingListAssociateProcessor(GS_LCB *plcb, G_INT32 CPUID, G_INT32 priority);

Description

The gu_SchedulingListAssociateProcessor function associates a system processor with a specified scheduling list. The priority is assigned to the association between the processor and the scheduling list. When tasks are ready to execute, the processor will select and execute the task from the highest priority scheduling list that it is associated with. The association with the lowest numerical value indicates the highest priority, ensuring that tasks in the most critical scheduling lists are prioritized for execution.

Parameters

The following parameters are required for the gu_SchedulingListAssociateProcessor function:

- **plcb**: A pointer to the GS_LCB structure representing the scheduling list to be associated with the processor.
- **CPUID**: The ID of the processor to be associated with the scheduling list.
- **priority**: The priority level for the association. A lower value indicates a higher priority, and the processor will first search the scheduling lists associated with the highest priority tasks that are ready to execute.

Returns

The gu_SchedulingListAssociateProcessor function returns G_TRUE if the association is successful. It returns G_FALSE if the association fails.

See also

gm_SchedulingListExclusionSectionEnter, gm_SchedulingListExclusionSectionExit, gu_SchedulingListAssociateTask, gu_SchedulingListCreate, gu_SchedulingListExclusionSet

---

gu_SchedulingListAssociateTask

Prototype

G_INT32 gu_SchedulingListAssociateTask(struct gs_tcb *ptcb, struct gs_lcb *plcb);

Description

The gu_SchedulingListAssociateTask function assigns a task to a specific scheduling list. Once assigned, the task will be scheduled according to the priority discipline defined for that scheduling list.

Parameters

The function accepts two parameters:

- **ptcb**: A pointer to the GS_TCB structure representing the task to be assigned.
- **plcb**: A pointer to the GS_LCB structure representing the scheduling list to which the task should be added.

Returns

The gu_SchedulingListAssociateTask function returns G_TRUE if the task was successfully assigned to the scheduling list, and G_FALSE otherwise.

See also

gm_SchedulingListExclusionSectionEnter, gm_SchedulingListExclusionSectionExit, gu_SchedulingListAssociateProcessor, gu_SchedulingListCreate, gu_SchedulingListExclusionSet

---

gu_SchedulingListCreate

Prototype

GS_LCB *gu_SchedulingListCreate(enum lcbtype lcbtype);

Description

The gu_SchedulingListCreate function creates a new scheduling list. The type of scheduling discipline used by the list is determined by the lcbtype parameter.

Parameters

The function accepts one parameter:

- **lcbtype**: An enumeration value specifying the type of scheduling list to create. This defines the scheduling discipline that will govern task scheduling within the new scheduling list.

Returns

The gu_SchedulingListCreate function returns a pointer (GS_LCB *) to the newly created GS_LCB structure. This pointer is essential for all subsequent operations involving this specific scheduling list. A '(GS_LCB *) 0' return value indicates failure to create the scheduling list.

See also
gm_SchedulingListExclusionSectionEnter, gm_SchedulingListExclusionSectionExit, gu_SchedulingListAssociateProcessor, gu_SchedulingListAssociateTask, gu_SchedulingListExclusionSet

---

gu_SchedulingListExclusionSet

Prototype
G_INT32 gu_SchedulingListExclusionSet(GS_LCB *plcb, G_INT32 exclusion);

Description
The gu_SchedulingListExclusionSet function sets the exclusion level for a scheduling list. The exclusion level limits the number of tasks from that list that can be simultaneously in the execution state. This mechanism can be used for load balancing or to ensure real-time properties by protecting against multiprocessor anomalies. Setting the exclusion to 1 can help safeguard real-time task scheduling from anomalies within the scheduling list.

Parameters
The function accepts two parameters:

- **plcb**: A pointer to the GS_LCB structure of the scheduling list whose exclusion level is to be modified.

- **exclusion**: An integer value that specifies the new exclusion level. A value of 1 ensures that multiple tasks from the scheduling list do not run concurrently on different processors. Values between 2 and the number of processors assigned to the scheduling list determine the number of tasks that can execute simultaneously on different processors. Additionally, values exceeding the number of processors assigned to the scheduling list will have no effect.

Returns
The function returns a G_TRUE.

See also
gm_SchedulingListExclusionSectionEnter, gm_SchedulingListExclusionSectionExit, gu_SchedulingListAssociateProcessor, gu_SchedulingListAssociateTask, gu_SchedulingListCreate

---

gu_SemaphoreCreate

Prototype
G_RCB *gu_SemaphoreCreate(int initial_count);

Description
The gu_SemaphoreCreate function creates a new semaphore resource. The semaphore is implemented using a G_RCB structure extended with fields from a T_SEMAPHORE_RESOURCE structure. The created semaphore includes event lists for tasks waiting to acquire the semaphore and tasks that currently hold the semaphore. Tasks requesting the semaphore and encountering a blocking condition (semaphore already acquired) will add themselves to the waiting list. This function can be called from either the main application code or from within a task; however, if

called within a task, it must be called before any semaphore request or release operations are performed, or an error will result.

Parameters

The function takes one parameter:

- **initial_count**: An integer specifying the initial count of the semaphore. This value determines the number of tasks that can simultaneously acquire the semaphore. A value of 1 creates a binary semaphore.

Returns

The gu_SemaphoreCreate function returns a pointer (G_RCB *) to the G_RCB structure that implements the semaphore resource. This pointer is used in all subsequent semaphore operations. A NULL pointer is returned if there is insufficient memory to create the semaphore. The return value should always be checked for errors.

See also

gu_SemaphorePost, gu_SemaphoreWait

---

gu_SemaphorePost

Prototype

G_INT32 gu_SemaphorePost(G_RCB *presource);

Description

The gu_SemaphorePost function releases a semaphore previously acquired by the currently executing task. If tasks are waiting to acquire the semaphore, the highest-priority waiting task will be granted the semaphore. If no tasks are waiting, the semaphore's internal count is incremented.

Parameters

The function accepts one parameter:

- **presource**: A pointer to the G_RCB structure representing the semaphore resource. This pointer was returned by the gu_SemaphoreCreate function.

Returns

The gu_SemaphorePost function returns G_TRUE if the semaphore was successfully released, and G_FALSE otherwise.

See also

gu_SemaphoreCreate, gu_SemaphoreWait

---

gu_SemaphoreWait

Prototype

G_INT32 gu_SemaphoreWait(G_RCB *presource, int blocking);

Description

The gu_SemaphoreWait function attempts to acquire a semaphore resource. If the semaphore's current count is greater than 0, the semaphore is granted to the calling task, and the count is decremented. If the count is 0, the behavior depends on the blocking parameter: if blocking is G_TRUE, the task is suspended until the semaphore becomes available; if blocking is G_FALSE, the function returns immediately without blocking. When blocking, the task's waiting priority is determined by its ready priority.

Parameters

The function uses the following parameters:

- **presource**: A pointer to the G_RCB structure representing the semaphore resource, as returned by gu_SemaphoreCreate.
- **blocking**: An integer flag. If G_TRUE, the task blocks until the semaphore is available; if G_FALSE, the function returns immediately if the semaphore is unavailable.

Returns

The gu_SemaphoreWait function returns G_TRUE if the semaphore was granted to the task, and G_FALSE when the semaphore was unavailable and blocking was G_FALSE.

See also

gu_SemaphoreCreate, gu_SemaphorePost

---

gu_SignalCreate

Prototype

GS_SCB *gu_SignalCreate(enum scbtype Type, G_INT32 Priority, void *pxcb, void *Signal_Code, void *Signal_Arg);

Description

The gu_SignalCreate function creates a signal of a specified type and associates it with a task or other system entity. The signal's priority determines its execution order when multiple signals are pending.

Parameters

The function takes the following parameters:

- **Type**: An enumeration value specifying the type of signal to create (e.g., G_SCBType_TCB_ABORTED).
- **Priority**: An integer representing the priority of the signal. Higher priority signals are executed before lower priority signals when multiple signals are pending.
- **pxcb**: A pointer to a control structure. This structure could represent various system entities like tasks, resources, processors, or events, to which the signal is linked.
- **Signal_Code**: A pointer to the function that implements the signal's behavior (the signal handler).
- **Signal_Arg**: A pointer to an argument that will be passed to the Signal_Code function when the signal is executed.

Returns

The gu_SignalCreate function returns a pointer to the newly created GS_SCB structure. A NULL return indicates failure.

See also

gu_SignalDestroy

---

gu_SignalDestroy

Prototype

G_INT32 gu_SignalDestroy(GS_SCB *pscb);

Description

The gu_SignalDestroy function removes a signal from a control block. This disassociates the signal from its associated task or system entity, preventing further execution of the signal's handler.

Parameters

The function accepts one parameter:

- **pscb**: A pointer to the GS_SCB structure representing the signal to be removed.

Returns

The gu_SignalDestroy function returns G_TRUE if the signal was successfully removed, and G_FALSE otherwise.

See also

gu_SignalCreate

---

gm_SystemTimePrescaleGet

Prototype

G_INT64 time = gm_SystemTimePrescaleGet;

Description

The gm_SystemTimePrescaleGet macro returns the time prescale. This prescale is used to obtaine the system time unit from the system clock.

Parameters

The gm_SystemTimePrescaleGet macro has no parameter.

Returns

The gm_SystemTimePrescaleGet macro returns the time prescale.

See also

gm_SystemTimePrescaleSet, gm_SystemTotalTimeGet, gm_WriteOutputs, gu_fprintf, gu_printf

---

gm_SystemTimePrescaleSet

Prototype

gm_SystemTimePrescaleSet(scale);

Description

gm_SystemTimePrescaleSet sets the system clock prescale to get the system time unitbuf
The gm_SystemTimePrescaleSet macro sets the system clock prescale to get the system time unit. By default, the prescale is set to configure a 10MHz frecuency for system time units.

Parameters

The gm_SystemTimePrescaleSet has one parameter:

- scale:The prescale configuration.

Returns

The gm_SystemTimePrescaleSet macro does not return any value.

See also
gm_SystemTimePrescaleGet, gm_SystemTotalTimeGet, gm_WriteOutputs, gu_fprintf, gu_printf

---

gm_SystemTotalTimeGet

Prototype
G_INT64 time = gm_SystemTotalTimeGet;
Description
gm_SystemTotalTimeGet returns the total system time. It is the time in non frozen mode plus the time in frozen mode.
Parameters
The gm_SystemTotalTimeGet macro has no parameter. It has to be noted that temporal constraints (deadline, period) are related to system time, obtained using the gm_SystemTimeGet macro, which is the time in non frozen mode.
Returns
The gm_SystemTotalTimeGet macro returns the total system time.
See also
gm_SystemTimePrescaleGet, gm_SystemTimePrescaleSet, gm_WriteOutputs, gu_fprintf, gu_printf

---

gm_WriteOutputs

Prototype
gm_WriteOutputs;
Description
The gm_WriteOutputs macro transfer the data input to the gemrtos_phy output conduit of GeMRTOS controller.
Parameters
The gm_WriteOutputs macro requires the following parameter:

- data:Data to be transfered the the gemrtos_phy output conduit of GeMRTOS controller.

Returns
The gm_WriteOutputs macro does not return any value..
See also
gm_SystemTimePrescaleGet, gm_SystemTimePrescaleSet, gm_SystemTotalTimeGet, gu_fprintf, gu_printf

---

gu_fprintf

Prototype
void gu_fprintf(char *format, ...);
Description
The gu_fprintf function formats text and writes it to standard error output (stderr).
Parameters
The following parameter is required for the gu_fprintf function:

- **format**: A string that may contain format specifiers like d, s, etc., which control the formatting of subsequent arguments.

Returns
The gu_fprintf function does not return any value.
See also
gm_SystemTimePrescaleGet, gm_SystemTimePrescaleSet, gm_SystemTotalTimeGet, gm_WriteOutputs, gu_printf

---

gu_printf

Prototype
void gu_printf(char *format, ...);
Description
The gu_printf function formats text and writes it to standard output.
Parameters
The following parameter is required for the gu_printf function:

- **format**: A string that may contain format specifiers like d, s, etc., which control the formatting of subsequent arguments.

Returns
The gu_printf function does not return any value.
See also
gm_SystemTimePrescaleGet, gm_SystemTimePrescaleSet, gm_SystemTotalTimeGet, gm_WriteOutputs, gu_fprintf

---

gu_TaskCreate

Prototype
void *gu_TaskCreate(void *TaskCode, void *p_arg, char *format, ...);
Description
The gu_TaskCreate function creates a task with default settings and returns a pointer to its GS_TCB structure. Task parameters can be modified before creation by adjusting default settings or after creation using task-related functions. While the function requires only TaskCode and p_arg, it allows for optional task description formatting using a printf-style format string and arguments.
Parameters
The function uses the following parameters:

- **TaskCode**: A pointer to the function that implements the task's code (the task's entry point). It is the name of the function that implements the task code.

- **p_arg**: A pointer to an argument that will be passed to the TaskCode function each time the task is invoked. This is a void * and can be cast to other types within the task code.

- **format**: A format string, similar to printf, used to create a description string for the task (up to G_TCB_DESCRIPTION_LENGTH characters). This string can contain format specifiers that are replaced by subsequent arguments.

Returns
The gu_TaskCreate function returns a pointer to the GS_TCB structure of the newly created task. This pointer should be used in all subsequent calls related to that task. A NULL return indicates task creation failure.
See also
gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

---

gu_TaskDelay

Prototype
G_INT32 gu_TaskDelay(G_INT32 hours, G_INT32 minutes, G_INT32 seconds, G_INT32 ms);
Description
The gu_TaskDelay function suspends the execution of the currently running task for a specified time interval. This function is useful within the infinite loop of a task to create periodic behavior.
Parameters
The function uses the following parameters to define the sleep interval:

- **hours**: The number of hours to sleep
- **minutes**: The number of minutes to sleep.
- **seconds**: The number of seconds to sleep.
- **ms**: The number of milliseconds to sleep.

Returns
The gu_TaskDelay function always returns G_TRUE.
See also
gu_TaskCreate, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

---

gu_TaskDelayTime

Prototype
G_INT32 gu_TaskDelayTime(gt_time ticks);
Description
The gu_TaskDelayTime function suspends the execution of the current task for a specified number of system clock ticks. This function provides a more direct way to specify sleep duration compared to gu_TaskDelay, using the system's time units directly.
Parameters
The function takes one parameter:

- **ticks**: The number of system clock ticks for which the task should sleep.

Returns

The gu_TaskDelayTime function always returns G_TRUE.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

---

gu_TaskGetCurrentTCB

Prototype

GS_TCB *gu_TaskGetCurrentTCB(void);

Description

The gu_TaskGetCurrentTCB function retrieves a pointer to the Task Control Block (GS_TCB) of the currently executing task.

Parameters

This function takes no parameters.

Returns

The gu_TaskGetCurrentTCB function returns a pointer to the GS_TCB structure of the currently running task.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

---

gu_TaskKill

Prototype

G_INT32 gu_TaskKill(GS_TCB *ptcb);

Description

The gu_TaskKill function terminates a task and releases all associated resources, returning them to the free lists.

Parameters

The function takes one parameter:

- **ptcb**: A pointer to the GS_TCB structure of the task to be terminated.

Returns

The gu_TaskKill function always returns G_TRUE.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

---

gu_TaskPeriodSet

Prototype

G_INT32 gu_TaskPeriodSet(struct gs_tcb *ptcb, unsigned int hours, unsigned int minutes, unsigned int seconds, unsigned int ms);

Description

The gu_TaskPeriodSet function sets the period for the next invocation of a task. The current task invocation period remains unaffected; the new period will apply only to subsequent invocations.

Parameters

The function uses the following parameters:

- **ptcb**: A pointer to the GS_TCB structure of the task whose period is to be set.
- **hours**: The number of hours in the new period.
- **minutes**: The number of minutes in the new period.
- **seconds**: The number of seconds in the new period.
- **ms**: The number of milliseconds in the new period.

Returns

The gu_TaskPeriodSet function always returns G_TRUE.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

---

gu_TaskReadyPrioritySet

Prototype

G_INT32 gu_TaskReadyPrioritySet(struct gs_tcb *ptcb, G_INT64 priority);

Description

The gu_TaskReadyPrioritySet function sets the ready priority of a task. This priority determines the task's position in the ready queue and influences its scheduling order. Note that larger values of priority represent lower priority; smaller values indicate higher priority.

Parameters

The function takes two parameters:

- **ptcb**: A pointer to the GS_TCB structure of the task whose ready priority is to be set.
- **priority**: A G_INT64 value representing the new ready priority for the task. Larger values indicate lower priority; smaller values indicate higher priority.

Returns

The gu_TaskReadyPrioritySet function always returns G_TRUE.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

gu_TaskResume

Prototype

G_INT32 gu_TaskResume(GS_TCB *ptcb);

Description

The gu_TaskResume function resumes a task that is currently in a waiting state.

Parameters

The function takes one parameter:

- **ptcb**: A pointer to the GS_TCB structure of the task to be resumed.

Returns

The gu_TaskResume function returns G_TRUE if the task was successfully resumed and G_FALSE otherwise.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

gu_TaskRunPrioritySet

Prototype

G_INT32 gu_TaskRunPrioritySet(struct gs_tcb *ptcb, G_INT64 priority);

Description

The gu_TaskRunPrioritySet function sets the run-time priority of a task. This priority determines the task's execution order when it is running. Larger values of priority represent lower priority; smaller values represent higher priority.

Parameters

The function takes these parameters:

- **ptcb**: A pointer to the GS_TCB structure of the task whose run-time priority is to be set.
- **priority**: A G_INT64 value specifying the new run-time priority. Larger values mean lower priority, and smaller values mean higher priority.

Returns

The gu_TaskRunPrioritySet function always returns G_TRUE.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskStartWithOffset, gu_TaskSuspend, gu_TaskTypeSet

gu_TaskStartWithOffset

Prototype

G_INT32 gu_TaskStartWithOffset(GS_TCB *ptcb, unsigned int hours, unsigned int minutes, unsigned int seconds, unsigned int ms);

Description

The gu_TaskStartWithOffset function starts a previously created task for execution, allowing the specification of a time offset for the task's first execution. This offset determines when the task will begin running relative to the time the function is called.

Parameters

The function uses the following parameters:

- **ptcb**: A pointer to the GS_TCB structure of the task to be started (obtained from gu_TaskCreate during task creation).
- **hours**: The number of hours in the starting offset.
- **minutes**: The number of minutes in the starting offset.
- **seconds**: The number of seconds in the starting offset.
- **ms**: The number of milliseconds in the starting offset.

Returns

The gu_TaskStartWithOffset function returns G_TRUE upon successful task startup.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskSuspend, gu_TaskTypeSet

---

gu_TaskSuspend

Prototype

G_INT32 gu_TaskSuspend(GS_TCB *ptcb);

Description

The gu_TaskSuspend function suspends a task, changing its state to waiting.

Parameters

The function takes one parameter:

- **ptcb**: A pointer to the GS_TCB structure of the task to be suspended.

Returns

The gu_TaskSuspend function always returns G_TRUE.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskTypeSet

---

gu_TaskTypeSet

Prototype

G_INT32 gu_TaskTypeSet(struct gs_tcb *ptcb, enum tcbtype type);

Description

The gu_TaskTypeSet function sets the type of a task. The valid task types are G_TCBType_PERIODIC and G_TCBType_OneShot.

Parameters

The function takes these parameters:

- **ptcb**: A pointer to the GS_TCB structure of the task whose type is to be modified.
- **type**: The type must be either G_TCBType_PERIODIC or G_TCBType_OneShot. Any other value will result in failure.

Returns

The gu_TaskTypeSet function returns G_TRUE if the task type was successfully set, and G_FALSE otherwise.

See also

gu_TaskCreate, gu_TaskDelay, gu_TaskDelayTime, gu_TaskGetCurrentTCB, gu_TaskKill, gu_TaskPeriodSet, gu_TaskReadyPrioritySet, gu_TaskResume, gu_TaskRunPrioritySet, gu_TaskStartWithOffset, gu_TaskSuspend

---

gu_TriggerCreate

Prototype

G_RCB *gu_TriggerCreate(int IRQ_ID);

Description

The function creates a trigger resource. It accepts an IRQ_ID argument to allow associating the trigger resource with a hardware interrupt.

Parameters

The trigger is created and initilized with the following paramenters:

- **IRQ_ID**: The IRQ_ID argument specifies the number of the hardware interrupt to associate with the trigger resource. Setting this argument to -1 indicates that no association with a hardware interrupt is desired.

Returns

The gu_TriggerCreate function returns a pointer to the newly created trigger resource. This pointer must be used to reference the trigger resource in all subsequent trigger-related functions.

See also

gu_TriggerDisable, gu_TriggerDisableHook, gu_TriggerEnable, gu_TriggerEnableHook, gu_TriggerRegisterTask, gu_TriggerRelease, gu_TriggerWait

---

gu_TriggerDisable

Prototype

G_INT32 gu_TriggerDisable(unsigned int IRQ_ID);

Description
The gu_TriggerDisable function disables the trigger resource, preventing it from being activated using either the gu_TriggerRelease function or the associated hardware interrupt.
Parameters
The disabling of the trigger resource is performed with the following parameter:

- **IRQ_ID**: This parameter represents either the IRQ number of the hardware interrupt associated with the trigger resource or the pointer to the trigger resource returned by the gu_TriggerCreate function. If the IRQ_ID is a pointer, it should be cast to an integer (int) for use within the function.

Returns
The function returns G_TRUE if the operation was successful.
See also
gu_TriggerCreate, gu_TriggerDisableHook, gu_TriggerEnable, gu_TriggerEnableHook, gu_TriggerRegisterTask, gu_TriggerRelease, gu_TriggerWait

---

gu_TriggerDisableHook

Prototype
G_INT32 gu_TriggerDisableHook(int IRQ_ID, void *code_callback, void *p_arg);
Description
The gu_TriggerDisableHook function sets the hook function to be called after the trigger resource is disabled.
Parameters
The disable hook function is specified using the following parameters:

- **IRQ_ID**: This parameter represents either the IRQ number of the hardware interrupt associated with the trigger resource or the pointer to the trigger resource returned by the gu_TriggerCreate function. If the IRQ_ID is a pointer, it should be cast to an integer (int) for use within the function.

- **code_callback**: This parameter defines the name of the function to be executed as a hook function when the trigger resource is disabled.

- **p_arg**: This parameter represents the value to be passed to the hook function when it is called. This allows the same hook function to be used for multiple trigger resources with different parameter values.

Returns
The function returns G_TRUE if the disable hook function was successfully configured; otherwise, it returns G_FALSE.
See also
gu_TriggerCreate, gu_TriggerDisable, gu_TriggerEnable, gu_TriggerEnableHook, gu_TriggerRegisterTask, gu_TriggerRelease, gu_TriggerWait

---

gu_TriggerEnable

Prototype

G_INT32 gu_TriggerEnable(int IRQ_ID);

Description

The gu_TriggerEnable function enables the trigger resource, allowing it to be activated using either the gu_TriggerRelease function or the associated hardware interrupt.

Parameters

The enabling of the trigger resource is performed with the following parameters:

- **IRQ_ID**: This parameter represents either the IRQ number of the hardware interrupt associated with the trigger resource or the pointer to the trigger resource returned by the gu_TriggerCreate function. If the IRQ_ID is a pointer, it should be cast to an integer (int) for use within the function.

Returns

The function returns G_TRUE if the operation was successful.

See also

gu_TriggerCreate, gu_TriggerDisable, gu_TriggerDisableHook, gu_TriggerEnableHook, gu_TriggerRegisterTask, gu_TriggerRelease, gu_TriggerWait

---

gu_TriggerEnableHook

Prototype

G_INT32 gu_TriggerEnableHook(int IRQ_ID, void *code_callback, void *p_arg);

Description

The gu_TriggerEnableHook sets the hook function to be called before the trigger resource is enabled.

Parameters

The gu_TriggerEnableHook function requires the following paramenters:

- **IRQ_ID**: This parameter represents either the IRQ number of the hardware interrupt associated with the trigger resource or the pointer to the trigger resource returned by the gu_TriggerCreate function. If the IRQ_ID is a pointer, it should be cast to an integer (int) for use within the function.

- **code_callback**: This parameter defines the name of the function to be executed as a hook function when the trigger resource is enabled.

- **p_arg**: This parameter represents the value to be passed to the hook function when it is called. This allows the same hook function to be used for multiple trigger resources with different parameter values.

Returns

The function returns G_TRUE if the enable hook function was successfully configured; otherwise, it returns G_FALSE.

See also

gu_TriggerCreate, gu_TriggerDisable, gu_TriggerDisableHook, gu_TriggerEnable, gu_TriggerRegisterTask, gu_TriggerRelease, gu_TriggerWait

gu_TriggerRegisterTask

Prototype

G_INT32 gu_TriggerRegisterTask(struct gs_tcb *ptcb, G_INT32 irq_nbr);

Description

The gu_TriggerRegisterTask function associates a task with a trigger resource.

Parameters

The task registration with the trigger resource is performed using the following parameters:

- **ptcb**: This is a pointer to the GS_TCB structure of the task to be associated with the trigger resource.
- **irq_nbr**: This is either the IRQ number of the hardware interrupt associated with the trigger or the pointer to the trigger resource returned by the gu_TriggerCreate function, cast to an integer (int).

Returns

The gu_TriggerRegisterTask function returns G_TRUE if the task registration with the trigger resource is successful; otherwise, it returns G_FALSE.

See also

gu_TriggerCreate, gu_TriggerDisable, gu_TriggerDisableHook, gu_TriggerEnable, gu_TriggerEnableHook, gu_TriggerRelease, gu_TriggerWait

gu_TriggerRelease

Prototype

G_INT32 gu_TriggerRelease(int irq_nbr);

Description

The function activates a trigger resource. If the trigger resource is enabled and all associated tasks are in a waiting state for the trigger, then the tasks are resumed or restarted.

Parameters

The trigger resource is activated using the following parameter:

- **irq_nbr**: This parameter represents either the IRQ number of the hardware interrupt associated with the trigger resource or the pointer to the trigger resource returned by the gu_TriggerCreate function. If the irq_nbr is a pointer, it should be cast to an integer (int) for use within the function.

Returns

The function returns G_TRUE if the trigger resource was successfully activated; otherwise, it returns G_FALSE.

See also

gu_TriggerCreate, gu_TriggerDisable, gu_TriggerDisableHook, gu_TriggerEnable, gu_TriggerEnableHook, gu_TriggerRegisterTask, gu_TriggerWait

gu_TriggerWait

Prototype

G_INT32 gu_TriggerWait(void);

Description

The gu_TriggerWait function places the task into a waiting state for the trigger resource it is registered to. It can be executed anywhere in the task's code, and the same effect occurs when the task completes its execution (assuming it's not an infinite loop).

Parameters

The gu_TriggerWait function does not require any parameters, as the task automatically waits for the trigger resource it is associated with.

Returns

The function returns G_TRUE if it is executed from within a task's code; otherwise, it returns G_FALSE if it is executed from the main code.

See also

gu_TriggerCreate, gu_TriggerDisable, gu_TriggerDisableHook, gu_TriggerEnable, gu_TriggerEnableHook, gu_TriggerRegisterTask, gu_TriggerRelease

---

GeMRTOS Control Blocks

struct g_rcb

Type	Field	Description
	BLOCK_HASH of the RCB: (GS_RCB *) + G_RCB_HASH. More...	
	Type of resource control block. More...	
	Pointer to linked list of waiting events of this event. More...	
	Pointer to the linked highest priority event. More...	
	Pointer to link resources in free list. More...	
	Pointer to the Linked list of signals. More...	
};		
	is the semaphore resource structure More...	
	is the queue resource, defined in mq.h More...	
	is the trigger resource, defined in trigger.h More...	
};		

---

struct T_QUEUE_RESOURCE

Type	Field	Description
G_INT32	MQ_priority_send	Priority for the next ECB to send (to put last)
G_INT32	MQ_msg_seq	Number of sequence of the current message

---

struct gs_ecb

Type	Field	Description
unsigned int	BLOCK_HASH	BLOCK_HASH of the ECB: (GS_ECB *) + G_ECB_HASH.
enum ecbstate	ECBState	Granted, Waiting, Free.
enum ecbtype	ECBType	Type of event control block.
TIMEPRIORITY	ECBValue	Occurrence Time of the event or Priority
struct gs_ecb *	ECB_NextECB	Pointer to linked list of waiting events of this event.
struct gs_ecb *	ECB_PrevECB	Pointer to linked list of waiting events of this event.
struct gs_tcb *	ECB_AssocTCB	Pointer to the task associated with the event
struct g_rcb *	ECB_AssocRCB	Pointer to the resource associated with the event
struct gs_ecb *	ECB_NextTCBAEL	Pointer to the next event of the same task.
struct gs_ecb *	ECB_PrevTCBAEL	Pointer to the previous event of the same task
struct gs_ecb *	ECB_NextECBAEL	Pointer to the event associated with this (ie timeout)
struct gs_scb *	ECB_NextECBASL	Pointer to the Linked list of signals.
struct gs_rrds *	ECB_RRDS	Pointer to the resource request structure or MCB.

---

struct gs_tcb

Type	Field	Description
unsigned int	BLOCK_HASH	BLOCK_HASH of the TCB: (GS_TCB *) + G_TCB_HASH.
enum tcbstate	TCBState	STATE of the task.
enum tcbtype	TCBType	TYPE of the task.

Type	Field	Description
G_INT64	TCBReadyPriority	Priority of the Task when Ready.
G_INT64	TCBRunPriority	Priority when it is executed.
G_INT64	TCBPeriod	Period of the task.
G_INT32	TCB_PrevExclusion	Previous Exclusion if task set the current Exclusion section (0 otherwise)
volatile GS_STK *	TCB_StackPointer	Pointer to current top of stack.
volatile GS_STK *	TCB_StackBottom	Botton Stack of the Task.
volatile GS_STK *	TCB_StackTop	Botton Stack of the Task.
volatile void *	TCB_TaskCode	Pointer to the Task Code.
volatile void *	TCB_TaskArg	Pointer to the argument of the first call.
struct gs_tcb *	TCB_NextTCB	Pointer to next TCB in the TCB list.
struct gs_tcb *	TCB_PrevTCB	Pointer to previous TCB in the TCB list.
struct gs_ecb *	TCB_NextTCBAEL	Pointer to linked list of waiting events of this task.
struct gs_scb *	TCB_NextTCBASL	Pointer to the Linked list of signals.
struct gs_scb *	TCB_NextTCBPSL	Pointer to signals waiting to execute.
G_INT32	TCB_AssocPCB	Processor assigned this task (0 is no assigned)
int	TCB_INTNumber	IRQ number if it is a ISR TCB.
struct gs_lcb *	TCB_RDY_LCB_Index	pointer to the ready list that should be inserted
volatile G_INT32	TCB_MTX_NESTED	Count for Mutex nesting of the task.

Type	Field	Description
char	TCB_description [G_TCB_DESCRIPTION_LENGTH]	
ucontext_t	uctx	
void *	uctx_stack	

struct gs_lcb

Type	Field	Description
unsigned int	BLOCK_HASH	BLOCK_HASH of the LCB: (GS_LCB *) + G_LCB_HASH.
enum lcbstate	LCBState	State of the List Control Block.
enum lcbtype	LCBType	Type of the List Control Block <>
G_INT32	LCBCurrentRunning	Current number of running tasks.
G_INT32	LCBExclusion	Maximum number of running task (0 for no limit)
struct gs_tcb *	LCB_NextTCBRUNL	Pointer to the TCB list of running tasks.
struct gs_tcb *	LCB_NextTCBRDYL	Pointer to the TCB of the Highest Priority Task.
struct gs_lcb *	LCB_NextLCBL	Pointer to the next list ordered by priority.
struct gs_lcb *	LCB_PrevLCBL	Pointer to the next list ordered by priority.
struct gs_pcb *	LCB_NextLCBFPL	Next free processor for this list.

struct T_SEMAPHORE_RESOURCE

Type	Field	Description
G_INT32	SEM_Current_Count	It is the current count of the semaphore. If it is equal to 0, no more grants are allowed. It is initialized with the SEM_Maximum_Count when the semaphore is created with the gu_SemaphoreCreate function.
G_INT32	SEM_Maximum_Count	It is the initial value of the SEM_Current_Count field. It is defined for debugging purposes only, and it could be removed.

struct gs_scb

Type	Field	Description
unsigned int	BLOCK_HASH	BLOCK_HASH of the SCB: (GS_SCB *) + G_SCB_HASH.
enum scbstate	SCBState	STATE of the signal.
enum scbtype	SCBType	TYPE of the signal.
G_INT32	SCBPriority	Priority of the SCB when it is linked.
void *	SCB_TaskCode	Pointer to the code of the signal.
void *	SCB_TaskArg	Pointer to the argument of the signal.
struct gs_scb *	SCB_NextSCB	Pointer to the next SCB linked.
void *	SCB_AssocXCB	Pointer to the data structure root of the SCBASL.

GeMRTOS Enumerations types

enum lcbtype

- **GS_LCBTypeEDF**: The GS_LCBTypeEDF scheduling list type implements the Earliest Deadline First (EDF) discipline among the tasks assigned to the scheduling list. In the EDF discipline, the earliest deadline, the highest priority. In order to maintain consistency, tasks that are assigned to an EDF scheduled list should be of the periodic type, with the deadline taken into account starting from the release time.
- **GS_LCBTypeFP**: The GS_LCBTypeFP scheduling list type implements the Fixed Priority (FP) discipline among the tasks assigned to the scheduling list. In the FP discipline, a priority is assigned to each task. Task priority may be modified during runtime. In order to maintain consistency, tasks that are assigned to an FP scheduled list should not be of the infinite-loop type without waiting for event suspension in order to avoid starving lower-priority tasks. Only the lowest-priority task could be implemented as an infinite-loop code.

---

enum scbtype

- **G_SCBType_TCB_ABORTED**: The G_SCBType_TCB_ABORTED signal type is defined to signaling when a task is aborted. A task aborting happens when a new release of a periodic task takes place before the previos invokation completes. The associated abortion function will be executed prior the execution of the next instance of the task.
- **G_SCBType_FROZEN_MODE**: The G_SCBType_FROZEN_MODE signal type is defined to signaling when GeMRTOS controller enters in frozen mode.

enum tcbtype

- **G_TCBType_OneShot**: The G_TCBType_OneShot task type makes the task code to be executed just once. The task must be released once again for another execution if the task code does not contain an infinite loop. Initialization tasks may be implemented as a G_TCBType_OneShot task type without infinite loop in the task code. G_TCBType_OneShot tasks are often implemented as an infinite loop to keep them running. When a task with an infinite loop is executed, it will take as much processor time as possible. It is possible to use different strategies to prevent one or many system tasks from being overly greedy about processor time and starving the others: Assigning lowest priorities to tasks: in this way, infinite-loop tasks will be executed only when the highest priority tasks are not requiring for execution. Suspending the task until an event: the task is suspended inside the infinite loop, waiting for an event. The events may be timed events (to execute the task regularly) or trigger events (such as waiting for an interrupt). Reducing the task priority: the task priority may be reduced inside the infinite loop to let the new higher-priority task be executed. This tactic should be implemented in all the infinite-loop tasks of the scheduling list to dynamically preserve a valid relationship among the system task priorities. Defining a round-robin scheduling mechanism in the scheduling list: a round-robin mechanism will execute each task during a certain interval, granting the processor access to each task in the scheduling list.
- **G_TCBType_PERIODIC**: The G_TCBType_PERIODIC task type makes the task code to be executed periodically. The period of the task is configured when the type is specified. The period of the task and the initial offset determines the future releases times of the task. If previous invocation of the task does not completes, then the previous invocation may be defined to be aborted or the next release skipped. Periodic tasks are useful to meet Nyquist and Shannon theorems in cyber-physical applications. However, if no scheduling analysis is performed, the system may became oversaturated and the deadlines missed.
- **G_TCBType_ISR**: The G_TCBType_ISR task type determines that the task is associated with a trigger resource. The G_TCBType_ISR type of the task should be set after the task is created using the gu_TriggerRegisterTask function.
- **G_TCBType_IDLE**: The G_TCBType_IDLE task type determines the task that a processor executes when no task requires for execution. The IDLE task is a GeMRTOS system task and there is one for each system processor. By default, the G_TCBType_IDLE task turns the processor into sleep mode in order to save energy and reduce the system bus utilization.