# GeMRTOS: Multiprocessor RTOS

## User and Development Manual — Chapter 13: API Reference

*gemrtos.com*

August 2026

Revision: 0.9a-2026-06-01

© 2026 GeMRTOS. All rights reserved.

ALTERA, AGILEX, ARRIA, CYCLONE, MAX, STRATIX, MEGACORE, NIOS, QUARTUS and PLATFORM DESIGNER are trademarks of Altera Corporation.

All other words and logos identified as trademarks or service marks are the property of their respective holders.

CHAPTER 13

# API Reference

## 13.1 API Naming Conventions

GeMRTOS uses a three-prefix naming convention to distinguish functions and macros by their intended caller and access level:

- **gm_** — Hardware-level macros that read or write GeMRTOS controller registers directly. These are low-level primitives intended for kernel developers. They must only be called while the GeMRTOS Mutex is held, except where explicitly stated otherwise.
- **gu_** — User-callable API functions. These are the primary interface for application developers. They manage the GeMRTOS Mutex internally, so they must not be called from within a critical section (unless nested mutex support is active).
- **gk_** — Kernel-internal functions. These are not part of the public API and must not be called directly by application code. They are documented here only for kernel developers and contributors.

The following table summarises when each prefix may be called:

Table 13.1: GeMRTOS API prefix summary.

| Prefix | Layer | Callable from | Mutex on entry |
|---|---|---|---|
| gu_ | User API | Task, `main()` | Must NOT be held |
| gm_ | Hardware macro | Kernel, ISR handler (critical) | Must be held |
| gk_ | Kernel internal | Kernel only | Must be held |

## 13.2 Critical Section Macros - Detailed Semantics

This section provides detailed semantics for critical section management macros, explaining their hardware interactions and correct usage patterns.

### 13.2.1 Critical Section Enter Macro Semantics

The macro `gm_GeMRTOSCriticalSectionEnter()` implements the entry to a critical section with automatic nesting support. Understanding its behavior is essential for correct use of GeMRTOS synchronization.

**Macro Definition**

```
#define gm_GeMRTOSCriticalSectionEnter \
    if (!(gm_TimeCountersResetGet())) { \
        GRTOS_CMD_CRITICAL_SECTION_GET;  // Acquire hardware mutex \
        if ((gm_KernelModeGet() == G_FALSE)) { \
            g_kcb.G_PCBTbl[gm_ProcessorId()-1].PCB_EXECTCB->TCB_MTX_NESTED++; \
        } \
    }
```

Code 13.1: gm_GeMRTOSCriticalSectionEnter macro

**Step-by-Step Behavior**

1. **Check time counters:** If time counters are in reset state, skip the entire operation (no-op during reset)
2. **Acquire mutex:** Call `GRTOS_CMD_CRITICAL_SECTION_GET` macro, which writes to ADDR_REG_0 to request mutex ownership
   - Hardware acquires mutex (or blocks until available)
   - Hardware auto-syncs: `R1_MTX_NESTED` $\leftarrow$ (`TCB_MTX_NESTED` $== 0?0 : 1$)
3. **Update nesting counter (user mode only):** If task is not in kernel mode, increment the task's nesting counter in its TCB
   - Kernel mode tasks skip this (they don't track nesting)
   - User mode tasks track nesting for proper release logic
4. **(Implicit synchronization):** Hardware has already synced nested flag. If this is first entry (TCB was 0), flag is now 0. If already nested (TCB was $>$0), flag is now 1.

**Key Points**

- **Not re-entrant:** If same task calls this twice without exiting, nesting counter increments twice (correct behavior)
- **Kernel mode difference:** Kernel code does NOT increment nesting counter. This allows kernel to have unlimited nesting without affecting user-level mutex behavior
- **Hardware synchronization:** The mutex acquisition includes automatic synchronization of nested flag (no manual IOWR needed at this point)
- **Time counter check:** The time counter reset check allows system reset to proceed without mutex conflicts

### 13.2.2 Critical Section Exit Macro Semantics

The macro `gm_GeMRTOSCriticalSectionExit()` implements the exit from a critical section with proper release timing.

**Macro Definition**

```
#define gm_GeMRTOSCriticalSectionExit \
    if (!(gm_TimeCountersResetGet())) { \
        if ((gm_KernelModeGet() == G_FALSE)) { \
            g_kcb.G_PCBTbl[gm_ProcessorId()-1].PCB_EXECTCB->TCB_MTX_NESTED--; \
            gm_MutexNestedValue(g_kcb.G_PCBTbl[gm_ProcessorId()-1].PCB_EXECTCB->TCB_MTX_NESTED); \
        } \
        GRTOS_MTX_RLS; \
    }
```

Code 13.2: gm_GeMRTOSCriticalSectionExit macro

**Step-by-Step Behavior**

1. **Check time counters:** If time counters are in reset, skip (no-op)
2. **Update nesting counter (user mode only):**
   (a) Decrement task's nesting counter (TCB_MTX_NESTED–)
   (b) Call `gm_MutexNestedValue()` to synchronize nested flag
3. **Synchronize nested flag with hardware:** The `gm_MutexNestedValue()` function performs an IOWR to ADDR_CTRL_STATUS:
   - If nesting counter now $= 0$: writes value 0, so `R1_MTX_NESTED` $\leftarrow 0$
   - If nesting counter $> 0$: writes value 1, so `R1_MTX_NESTED` $\leftarrow 1$
4. **Request release:** Call `GRTOS_MTX_RLS` macro, which writes to ADDR_REG_1 bit 4 (release request)
   - Hardware checks: is `R1_MTX_NESTED` $== 0$?
   - If yes: mutex is released (next waiter can acquire)
   - If no: mutex is held (still nested)

**Critical Sequence**

The order of operations is critical:

1. Decrement counter first
2. Synchronize flag second (so hardware sees new value)
3. Request release third (so hardware uses updated flag)

If order were different (e.g., release first, then update flag), race conditions could occur.

### 13.2.3 Mutex Nested Value Synchronization

The function `gm_MutexNestedValue(data)` is the mechanism by which software informs the hardware controller of nesting level changes.

**Function Definition**

```
static inline void gm_MutexNestedValue(G_UINT32 data)
{
    const G_UINT32 value =
        ((G_UINT32)1u << CTRL_MTX_RLS_DSB_BIT) |
        ((data != 0u) ? (G_UINT32)1u : (G_UINT32)0u);

    gm_IOWR(GEMRTOS_DRIVER_BASE, ADDR_CTRL_STATUS, value);
}
```

Code 13.3: gm_MutexNestedValue function

**Behavior**

When the nesting counter changes, software must update the nested flag register:

1. If `data == 0` (no more nesting):
   - Write: Bit 24 set (CTRL_MTX_RLS_DSB_BIT), bit $0 = 0$
   - Hardware sets: `R1_MTX_NESTED` $\leftarrow 0$
   - Effect: Gate opens (release now possible)
2. If `data != 0` (still nested):
   - Write: Bit 24 set, bit $0 = 1$
   - Hardware sets: `R1_MTX_NESTED` $\leftarrow 1$
   - Effect: Gate closes (release blocked)

**Timing Guarantee**

The hardware ADDR_CTRL_STATUS handler processes the write in the same clock cycle. Software can immediately proceed to release request knowing the flag is updated.

## 13.3 GeMRTOS API Functions categories

### 13.3.1 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`

### 13.3.2 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_GatedClockDisable`

`gm_GatedClockEnable`

`gm_StatusFrozenModeEnableGet`

`gm_SystemReset`

`gm_TimeCountersReset`

`gm_TimeCountersHold`

`gm_TimeCountersUnhold`

`gm_TimeCoutersUnreset`

`gu_FrozenModeThresholdGet`

`gu_FrozenModeThresholdSet`

`gu_FrozenModeTimeGet`

### 13.3.3 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`

`gk_NextOccTimeProcessor`

`gm_TriggerPrcIRQ`

### 13.3.4 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

### 13.3.5 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_PRC_INT_DSB
gm_PRC_INT_ENB
gm_ProcessorHalt
gm_ProcessorId
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq

### 13.3.6 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.

gu_SchedulingListAssociateProcessor
gu_SchedulingListAssociateTask
gu_SchedulingListCreate
gu_SchedulingListExclusionSet

### 13.3.7 Semaphore and Mutex category

The **Semaphore** and **Mutex** category in GeMRTOS encompasses functions and macros designed to facilitate synchronization and resource management among concurrent tasks within the real-time operating system. **Semaphores** and **mutexes** 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** and **Mutex** category, developers can create and manage both binary and counting semaphores, and mutexes, allowing for fine-grained control over task execution and resource allocation. The functions within this category enable operations such as semaphore and mutex creation, waiting, and signaling, effectively coordinating task activities and synchronizing their behavior.
The use of semaphores and mutexes 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 and mutexes, 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_SemaphoreCreateBinary`
`gu_SemaphoreCreateMutex`
`gu_SemaphoreCreateRecursiveMutex`
`gu_SemaphoreGetCount`
`gu_SemaphoreGive`
`gu_SemaphoreTake`

### 13.3.8 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`

### 13.3.9 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_ERETAddressSet`
`gm_MutexNestedValue`
`gm_ReadInputs`
`gm_StatusFrozenModeActiveGet`
`gm_WriteOutputs`
`gu_fprintf`
`gu_printf`

### 13.3.10 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
```

### 13.3.11 Time Management category

In GeMRTOS, the core time management tasks revolve around manipulating system times and time intervals to support precise task synchronization and scheduling. These functions enable setting, reading, and adjusting system clocks, as well as defining time intervals for specific operations. Through effective time manipulation, developers can coordinate task execution and ensure activities occur at exact moments, maintaining the system's deterministic behavior.

```
gm_StatusCountingTimeGet
gm_StatusResetCountingTimeGet
gm_SystemTimePrescaleGet
gm_SystemTimePrescaleSet
gm_TimeIntervalGet
gu_SystemTotalTimeGet
```

### 13.3.12 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_TriggerSetTimeoutType
gu_TriggerWait
```

## 13.4 GeMRTOS Functions

### 13.4.1 gm_GeMRTOSCriticalSectionEnter

**Prototype**

```
void gm_GeMRTOSCriticalSectionEnter;  /* macro: GRTOS_USER_CRITICAL_SECTION_GET */
```

**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.

**Thread-safety**

Blocking — waits until the hardware mutex is granted. Not reentrant unless nested mutex is enabled.

**Error conditions**

None returned; execution blocks indefinitely if mutex is never released.

**Preconditions**

Must not already hold the GeMRTOS mutex (unless nested mutex is active via `gm_MutexNestedValue`).

**Postconditions**

The calling processor holds the GeMRTOS controller hardware mutex.

**See also**

`gm_GeMRTOSCriticalSectionExit`
`gm_IrqDisable`
`gm_IrqEnable`
`gm_ProcessorInterrupt`
`gu_ProcessorWaitForIrq`

### 13.4.2 gm_GeMRTOSCriticalSectionExit

**Prototype**

```
void gm_GeMRTOSCriticalSectionExit;  /* macro: GRTOS_USER_CRITICAL_SECTION_RELEASE */
```

**Description:** The `gm_GeMRTOSCriticalSectionExit` macro exits the critical section from the current processor, allowing other processors to 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.

**Thread-safety**
Must be called by the processor that called `gm_GeMRTOSCriticalSectionEnter`.

**Error conditions**
Undefined behavior if called without a prior matching Enter.

**Preconditions**
The calling processor must hold the GeMRTOS mutex.

**Postconditions**
The GeMRTOS mutex is released; other processors may acquire it.

**See also**
`gm_GeMRTOSCriticalSectionEnter`
`gm_IrqDisable`
`gm_IrqEnable`
`gm_ProcessorInterrupt`
`gu_ProcessorWaitForIrq`

---

### 13.4.3 gm_FrozenModeDisable

**Prototype**

```
void gm_FrozenModeDisable;  /* macro: GRTOS_CMD_FRZ_DSB_SET */
```

**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.

**Thread-safety**
Must be called within a critical section (GeMRTOS mutex held).

**Error conditions**
None.

**Preconditions**
GeMRTOS mutex must be held.

**Postconditions**
Frozen mode is disabled; system stays in non-frozen mode regardless of time threshold.

**See also**
`gm_FrozenModeEnable`
`gm_GatedClockDisable`
`gm_GatedClockEnable`
`gm_ProcessorInterrupt`
`gu_ProcessorWaitForIrq`

gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersHold
gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet
gu_FrozenModeTimeGet

### 13.4.4 gm_FrozenModeEnable

**Prototype**

```
void gm_FrozenModeEnable;  /* macro: GRTOS_CMD_FRZ_ENB_SET */
```

**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 gu_FrozenModeThresholdSet function. The gm_FrozenModeDisable macro disables the frozen mode.

**Parameters**

The gm_FrozenModeEnable macro does not accept any parameters.

**Returns**

The gm_FrozenModeEnable macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

GeMRTOS mutex must be held. Frozen threshold (gu_FrozenModeThresholdSet) should be configured first.

**Postconditions**

Frozen mode is enabled; controller will switch to frozen mode when system time exceeds the next event by the threshold.

**See also**

gm_FrozenModeDisable
gm_GatedClockDisable
gm_GatedClockEnable
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersHold
gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet

gu_FrozenModeTimeGet

## 13.4.5 gm_GatedClockDisable

**Prototype**

```
void gm_GatedClockDisable;  /* macro: GRTOS_CMD_GATED_CLOCK_DIS */
```

**Description:** The gm_GatedClockDisable macro enables the output clock of the the GeMRTOS controller. It may be used in special applications.

**Parameters**

The gm_GatedClockDisable macro does not accept any parameters.

**Returns**

The gm_GatedClockDisable macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

GeMRTOS mutex must be held.

**Postconditions**

The gated clock output port of the GeMRTOS controller is driven low.

**See also**

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_GatedClockEnable
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersHold
gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet
gu_FrozenModeTimeGet

## 13.4.6 gm_GatedClockEnable

**Prototype**

```
void gm_GatedClockEnable;  /* macro: GRTOS_CMD_GATED_CLOCK_ENB */
```

**Description:** The gm_GatedClockEnable macro enables the output clock of the the GeMRTOS controller. It may be used in special applications.

**Parameters**

The `gm_GatedClockEnable` macro does not accept any parameters.

**Returns**

The `gm_GatedClockEnable` macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

GeMRTOS mutex must be held.

**Postconditions**

The gated clock output port mirrors the GeMRTOS controller input clock.

**See also**

`gm_FrozenModeDisable`
`gm_FrozenModeEnable`
`gm_GatedClockDisable`
`gm_ProcessorInterrupt`
`gu_ProcessorWaitForIrq`
`gm_StatusFrozenModeEnableGet`
`gm_SystemReset`
`gm_TimeCountersReset`
`gm_TimeCountersHold`
`gm_TimeCountersUnhold`
`gm_TimeCoutersUnreset`
`gu_FrozenModeThresholdGet`
`gu_FrozenModeThresholdSet`
`gu_FrozenModeTimeGet`

---

## 13.4.7 gm_StatusFrozenModeEnableGet

**Prototype**

```
G_INT32 gm_StatusFrozenModeEnableGet;  /* macro: GRTOS_CMD_GET_FRZ_ENB */
```

**Description:** `gm_StatusFrozenModeEnableGet` returns the status of the enable of the frozen mode (G_TRUE if enabled, G_FALSE if disabled). The execution of `gm_FrozenModeEnable` enables the frozen mode. The execution of `gm_FrozenModeDisable` disables the frozen mode.

**Parameters**

`gm_StatusFrozenModeEnableGet` has no parameter.

**Returns**

The `gm_StatusFrozenModeEnableGet` macro returns the time prescale.

**Thread-safety**

Read-only; safe to call within a critical section.

**Error conditions**

None.

**Preconditions**
GeMRTOS mutex should be held for a consistent read.

**Postconditions**
None.

**See also**
`gm_FrozenModeDisable`
`gm_FrozenModeEnable`
`gm_GatedClockDisable`
`gm_GatedClockEnable`
`gm_SystemReset`
`gm_TimeCountersReset`
`gm_TimeCountersHold`
`gm_TimeCountersUnhold`
`gm_TimeCoutersUnreset`
`gu_FrozenModeThresholdGet`
`gu_FrozenModeThresholdSet`
`gu_FrozenModeTimeGet`

---

## 13.4.8 gm_SystemReset

**Prototype**

```
void gm_SystemReset;  /* writes CTRL_RESET_BIT to CTRL register */
```

**Description:** The `gm_SystemReset` macro resets the GeMRTOS controller. Is is executed only by the GeMRTOS kernel.

**Parameters**
The `gm_SystemReset` macro does not accept any parameters.

**Returns**
The `gm_SystemReset` macro does not return any value.

**Thread-safety**
Resets the entire GeMRTOS controller; all state is lost.

**Error conditions**
Does not return; system restarts.

**Preconditions**
GeMRTOS mutex must be held.

**Postconditions**
All GeMRTOS controller registers are reset to their initial state.

**See also**
`gm_FrozenModeDisable`
`gm_FrozenModeEnable`
`gm_GatedClockDisable`
`gm_GatedClockEnable`
`gm_ProcessorInterrupt`
`gu_ProcessorWaitForIrq`

gm_StatusFrozenModeEnableGet
gm_TimeCountersReset
gm_TimeCountersHold
gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet
gu_FrozenModeTimeGet

---

### 13.4.9 gm_TimeCountersReset

**Prototype**

```
void gm_TimeCountersReset;  /* macro: GRTOS_CMD_TM_CNT_CLR */
```

**Description:** The gm_TimeCountersReset macro resets and holds reset the time counters. Is is executed only by the GeMRTOS kernel.

**Parameters**

The gm_TimeCountersReset macro does not accept any parameters.

**Returns**

The gm_TimeCountersReset macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

Called at system startup before gm_TimeCoutersUnreset.

**Postconditions**

System time counters are reset and held at zero.

**See also**

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_GatedClockDisable
gm_GatedClockEnable
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersHold
gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet
gu_FrozenModeTimeGet

---

### 13.4.10 gm_TimeCountersHold

**Prototype**

```
void gm_TimeCountersHold;  /* writes CTRL_TM_CNTS_HLD_BIT */
```

**Description:** The gm_TimeCountersHold macro holds the time counters. Is is executed only by the GeMRTOS kernel.

**Parameters**

The gm_TimeCountersHold macro does not accept any parameters.

**Returns**

The gm_TimeCountersHold macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

GeMRTOS mutex must be held.

**Postconditions**

All time counters are frozen at their current values.

**See also**

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_GatedClockDisable
gm_GatedClockEnable
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet
gu_FrozenModeTimeGet

---

### 13.4.11 gm_TimeCountersUnhold

**Prototype**

```
void gm_TimeCountersUnhold;  /* clears CTRL_TM_CNTS_HLD_BIT */
```

**Description:** The gm_TimeCountersUnhold macro unholds the time counters running from its current state. Is is executed only by the GeMRTOS kernel.

**Parameters**

The gm_TimeCountersUnhold macro does not accept any parameters.

**Returns**

The gm_TimeCountersUnhold macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

GeMRTOS mutex must be held. gm_TimeCountersHold must have been called first.

**Postconditions**

Time counters resume incrementing from the held values.

**See also**

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_GatedClockDisable
gm_GatedClockEnable
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersHold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet
gu_FrozenModeTimeGet

---

## 13.4.12 gm_TimeCoutersUnreset

**Prototype**

```
void gm_TimeCoutersUnreset;  /* clears CTRL_TM_CNTS_CLR_BIT */
```

**Description:** The gm_TimeCoutersUnreset macro unresets theSystemTime Register for running. Is is executed only by the GeMRTOS kernel.

**Parameters**

The gm_TimeCoutersUnreset macro does not accept any parameters.

**Returns**

The gm_TimeCoutersUnreset macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

gm_TimeCountersReset must have been called first. GeMRTOS mutex must be held.

**Postconditions**

Time counters are released and begin incrementing.

**See also**

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_GatedClockDisable
gm_GatedClockEnable
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersHold
gm_TimeCountersUnhold
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet
gu_FrozenModeTimeGet

---

### 13.4.13 gu_FrozenModeThresholdGet

**Prototype**

```
G_UINT64 gu_FrozenModeThresholdGet(void);  /* reads FRZ_THR register pair */
```

**Description:** The gu_FrozenModeThresholdGet function 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 function is useful for determining the current threshold value, which is critical for managing the activation of frozen mode.

**Parameters**

The gu_FrozenModeThresholdGet function does not accept any parameters

**Returns**

The gu_FrozenModeThresholdGet function returns the current value of the Frozen Time Threshold register in the GeMRTOS controller.

**Thread-safety**

Safe to call from task context; acquires mutex internally.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

None.

**See also**

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_GatedClockDisable
gm_GatedClockEnable
gm_ProcessorInterrupt

gu_ProcessorWaitForIrq
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersHold
gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdSet
gu_FrozenModeTimeGet

---

## 13.4.14 gu_FrozenModeThresholdSet

**Prototype**

```
G_UINT32 gu_FrozenModeThresholdSet(G_UINT64 time_set);  /* GRTOS_CMD_FRZ_TM_THR_SET */
```

**Description:** The gu_FrozenModeThresholdSet function 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 function is essential for configuring the threshold that determines when the frozen mode becomes active when it is enabled.

**Parameters**

The gu_FrozenModeThresholdSet function 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 gu_FrozenModeThresholdSet function does not return any value.

**Thread-safety**

Safe to call from task context; acquires mutex internally.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

The frozen threshold is updated; frozen mode will activate when system time exceeds the next event time by this threshold.

**See also**

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_GatedClockDisable
gm_GatedClockEnable
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersHold

gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeTimeGet

### 13.4.15 gu_FrozenModeTimeGet

**Prototype**

```
G_UINT64 gu_FrozenModeTimeGet(void);  /* reads FRZ_TM register pair */
```

**Description:** gu_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 gu_FrozenModeTimeGet function has no parameter.

**Returns**

The gu_FrozenModeTimeGet function returns the accumulated time the system was in Frozen Mode, hold in the the R_FRZ_CNT register of the GeMRTOS controller.

**Thread-safety**

Safe from task context.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

None.

**See also**

gm_FrozenModeDisable
gm_FrozenModeEnable
gm_GatedClockDisable
gm_GatedClockEnable
gm_StatusFrozenModeEnableGet
gm_SystemReset
gm_TimeCountersReset
gm_TimeCountersHold
gm_TimeCountersUnhold
gm_TimeCoutersUnreset
gu_FrozenModeThresholdGet
gu_FrozenModeThresholdSet

### 13.4.16 gm_IrqDisable

**Prototype**

```
void gm_IrqDisable(G_INT32 irq);  /* macro: GRTOS_CMD_IRQ_ENB_CLR(irq) */
```

**Description:** The gm_IrqDisable macro disables the specified device interrupt request event (IRQ) in the GeMRTOS controller. The gm_IrqEnable(irq)macro enables the irq external interrupt 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.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

Undefined behavior if irq is out of range [1, ALT_NIRQ].

**Preconditions**

GeMRTOS mutex must be held. irq must be a valid IRQ index.

**Postconditions**

The specified device interrupt request is disabled in the GeMRTOS controller.

**See also**

gm_IrqEnable
gk_NextOccTimeProcessor
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_TriggerPrcIRQ

---

### 13.4.17 gm_IrqEnable

**Prototype**

```
void gm_IrqEnable(G_INT32 irq);  /* macro: GRTOS_CMD_IRQ_ENB_SET(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. The gm_IrqDisable(irq)macro disables the irq external interrupt in the GeMRTOS controller.

**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.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

Undefined behavior if irq is out of range.

**Preconditions**

GeMRTOS mutex must be held. A trigger must be associated with this IRQ via gu_TriggerCreate.

**Postconditions**

The specified device interrupt request will trigger a processor interrupt when the device asserts it.

**See also**

gm_IrqDisable
gk_NextOccTimeProcessor
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq
gm_TriggerPrcIRQ

---

## 13.4.18 gk_NextOccTimeProcessor

**Prototype**

```
void gk_NextOccTimeProcessor(void);  /* kernel-internal, static inline */
```

**Description:** The gk_NextOccTimeProcessor function reads the Kernel Control Block to determine which processor is associated with the next occurrence-time task, then sets the Next Occurrence Time Processor field in the GRTOS controller (or 0 if no such task is executing). Kernel-internal: not part of the public API, must be called with the mutex already held.

**Parameters**

None.

**Returns**

The gk_NextOccTimeProcessor macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

Undefined behavior if processor is not a valid CPUID.

**Preconditions**

GeMRTOS mutex must be held.

**Postconditions**

The GeMRTOS controller knows which processor executes the next timed event; it will interrupt that processor when the event occurs.

**See also**

gm_IrqDisable
gm_IrqEnable
gm_TriggerPrcIRQ

---

## 13.4.19 gm_TriggerPrcIRQ

**Prototype**

```
void gm_TriggerPrcIRQ(G_INT32 proc);  /* macro: GRTOS_CMD_PRC_INT(proc) */
```

**Description:** The gm_TriggerPrcIRQ macro interrupts the processor proc. If proc interrupt is enabled in the GRTOS controller, then the processor is interrupted.

**Parameters**

The `gm_TriggerPrcIRQ` macro accepts the following parameter:

- `proc:CPU_ID of processor to interrupt.`

**Returns**

The `gm_TriggerPrcIRQ` macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

Undefined behavior if `proc` is not a valid CPUID.

**Preconditions**

GeMRTOS mutex must be held. Target processor must have interrupts enabled.

**Postconditions**

The target processor is interrupted and enters its interrupt handler.

**See also**

`gm_IrqDisable`
`gm_IrqEnable`
`gk_NextOccTimeProcessor`
`gm_ProcessorInterrupt`
`gu_ProcessorWaitForIrq`

---

## 13.4.20 gu_MessageQueueCreate

**Prototype**

```
GS_RCB *gu_MessageQueueCreate(void);  /* calls gu_queue_create() */
```

**Description:** The `gu_MessageQueueCreate` function creates a new message queue resource. This resource is implemented using a GS_RCB structure, extended with fields from aT_QUEUE_RESOURCEstructure. The created queue includes event lists for producers (waiting to send) and MQreceivers (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 ( GS_RCB *) to the newly created message queue resource. This pointer is essential for all subsequent operations in the queue. A NULL pointer is returned if there is insufficient memory to create the queue or if no more queue resources are available.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `NULL` if no free RCB is available.

**Preconditions**

GeMRTOS must be initialized.

**Postconditions**

A new message queue RCB is allocated. Subscribers must be added with `gu_MessageQueueSubscribe` before

sending.

**See also**

gu_MessageQueuePrintf
gu_MessageQueueReceive
gu_MessageQueueSend
gu_MessageQueueSubscribe

---

## 13.4.21 gu_MessageQueuePrintf

**Prototype**

```
G_UINT32 gu_MessageQueuePrintf(GS_RCB *prcb, const 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 MQreceivers subscribed to the queue.

**Parameters**

The function accepts the following parameters:

- prcb: A pointer to the GS_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.

**Thread-safety**

Safe from task context.

**Error conditions**

Returns negative value on send failure.

**Preconditions**

prcb must point to a valid message queue RCB created by gu_MessageQueueCreate.

**Postconditions**

A formatted string message is enqueued for delivery to all subscribers.

**See also**

gu_MessageQueueCreate
gu_MessageQueueReceive
gu_MessageQueueSend
gu_MessageQueueSubscribe

---

## 13.4.22 gu_MessageQueueReceive

**Prototype**

```
int gu_MessageQueueReceive(GS_RCB *prcb);
```

**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 GS_RCB structure of the message queue from which to receive the message.

**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.

**Thread-safety**

Task context only; blocks until a message arrives.

**Error conditions**

Returns 0 if timeout or no message; returns message length on success.

**Preconditions**

The calling task must have subscribed via `gu_MessageQueueSubscribe`. `buffer_msg` must point to a buffer of at least `buffer_length` bytes.

**Postconditions**

The message is copied into `buffer_msg`; the task resumes.

**See also**

`gu_MessageQueueCreate`
`gu_MessageQueuePrintf`
`gu_MessageQueueSend`
`gu_MessageQueueSubscribe`

---

## 13.4.23 gu_MessageQueueSend

**Prototype**

```
int gu_MessageQueueSend(GS_RCB *prcb, const char *pmsg, int msg_length, G_UINT64 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 MQreceivers or until a timeout occurs.

**Parameters**

The function uses the following parameters:

- `prcb`: A pointer to the GS_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`: Agt_timevalue 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.

**Thread-safety**

Safe from task context.

**Error conditions**

Returns negative on timeout or queue error. Returns 0 on success.

**Preconditions**

`prcb` must be a valid queue. `pmsg` must not be NULL. `msg_length` must be positive.

**Postconditions**

Message is delivered to all active subscribers; sender blocks until all consumers receive it or timeout expires.

**See also**

`gu_MessageQueueCreate`
`gu_MessageQueuePrintf`
`gu_MessageQueueReceive`
`gu_MessageQueueSubscribe`

---

### 13.4.24 gu_MessageQueueSubscribe

**Prototype**

```
GS_ECB *gu_MessageQueueSubscribe(GS_TCB *ptcb, GS_RCB *presource, void *buffer_msg, G_UINT32 buffer_l
```

**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 MQreceivers 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 theTaskControl Block ( GS_TCB ) of the task being subscribed to the queue.
- **`presource`**: A pointer to the GS_RCB structure representing the message queue resource to which the task is subscribing.
- **`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.

**Returns**

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

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `NULL` if no free ECB is available.

**Preconditions**

`ptcb` and `presource` must be valid pointers. Typically called before the queue is used.

**Postconditions**

The task is registered as a consumer; it will receive all future messages sent to the queue.

**See also**

`gu_MessageQueueCreate`
`gu_MessageQueuePrintf`

gu_MessageQueueReceive
gu_MessageQueueSend

## 13.4.25 gm_PRC_INT_DSB

**Prototype**

```
void gm_PRC_INT_DSB;  /* macro: GRTOS_CMD_PRC_INT_DSB */
```

**Description:** The gm_PRC_INT_DSB macro disables the processor interrupt from the GeMRTOS controller.

**Parameters**

The gm_PRC_INT_DSB macro does not require any parameters.

**Returns**

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

**Thread-safety**

Affects only the calling processor's interrupt enable in the GeMRTOS controller.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

The GeMRTOS controller will not generate processor interrupt requests for the calling processor.

**See also**

gm_PRC_INT_ENB
gm_ProcessorHalt
gm_ProcessorId
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq

## 13.4.26 gm_PRC_INT_ENB

**Prototype**

```
void gm_PRC_INT_ENB;  /* macro: GRTOS_CMD_PRC_INT_ENB */
```

**Description:** The gm_PRC_INT_ENB macro enables the processor interrupt from the GeMRTOS controller.

**Parameters**

The gm_PRC_INT_ENB macro does not require any parameters.

**Returns**

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

**Thread-safety**

Affects only the calling processor.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

The GeMRTOS controller is enabled to interrupt the calling processor on events.

**See also**

`gm_PRC_INT_DSB`
`gm_ProcessorHalt`
`gm_ProcessorId`
`gm_ProcessorInterrupt`
`gu_ProcessorWaitForIrq`

---

## 13.4.27 gm_ProcessorHalt

**Prototype**

```
void gm_ProcessorHalt;  /* macro: GRTOS_CMD_HALT_PROCESSOR */
```

**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. When G_DEBUG_WAITING_LOOP is defined equal to 0, `gm_ProcessorHalt` is implemented by asserting the waitrequest signal when processor reads the wait bit of ADDR_REG_0 address in the processor memory of the GeMRTOS controller. When G_DEBUG_WAITING_LOOP is defined not equal to 0, `gm_ProcessorHalt` is implemented as a waiting loop reading the the wait bit of ADDR_REG_0 address in the processor memory of the GeMRTOS controller in order to be complilance with debugging tools that does not support processor halted with waitrequest signal.

**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.

**Thread-safety**

Halts the calling processor; execution resumes on next interrupt.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

Processor is in halt/idle state until the next interrupt fires.

**See also**

`gm_PRC_INT_DSB`
`gm_PRC_INT_ENB`
`gm_ProcessorId`
`gm_ProcessorInterrupt`

gu_ProcessorWaitForIrq

## 13.4.28 gm_ProcessorId

**Prototype**

```
G_INT32 gm_ProcessorId;  /* macro: GRTOS_CMD_PRC_ID */
```

**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.

**Thread-safety**

Safe to call from any context; reads a processor-specific register.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

None.

**See also**

gm_PRC_INT_DSB
gm_PRC_INT_ENB
gm_ProcessorHalt
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq

## 13.4.29 gm_ProcessorInterrupt

**Prototype**

```
void gm_ProcessorInterrupt(G_INT32 proc);  /* macro: GRTOS_CMD_PRC_INT(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.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

Undefined if proc is an invalid CPUID.

**Preconditions**

GeMRTOS mutex must be held. Target processor must be running and have interrupts enabled.

**Postconditions**

The target processor is interrupted; the caller waits until the target enters its IRQ handler.

**See also**

gm_PRC_INT_DSB
gm_PRC_INT_ENB
gm_ProcessorHalt
gm_ProcessorId
gu_ProcessorWaitForIrq

---

## 13.4.30 gu_ProcessorWaitForIrq

**Prototype**

```
void gu_ProcessorWaitForIrq(G_UINT32 IRQ_mask);  /* waits in low-power state for the given IRQ(s) */
```

**Description:** The gu_ProcessorWaitForIrq function halts the calling processor until an interrupt occurs on one of the specified masked IRQs. Useful for a user task or driver waiting on a specific interrupt subset, as opposed to the kernel's own gm_ProcessorHalt, which halts until any interrupt. Manages its own critical section internally.

**Parameters**

The gu_ProcessorWaitForIrq function 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 gu_ProcessorWaitForIrq function does not return any value.

**Thread-safety**

Affects only the calling processor.

**Error conditions**

None.

**Preconditions**

Must NOT be called from within a critical section – it enters and exits one internally.

**Postconditions**

Processor resumes execution upon receiving the next GeMRTOS controller interrupt.

**See also**

gm_PRC_INT_DSB
gm_PRC_INT_ENB
gm_ProcessorHalt
gm_ProcessorId
gm_ProcessorInterrupt

---

### 13.4.31 gu_SchedulingListAssociateProcessor

**Prototype**

```
G_UINT32 gu_SchedulingListAssociateProcessor(GS_LCB *plcb, G_UINT32 cpu_id, G_UINT32 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 theGS_LCBstructure representing the scheduling list to be associated with the processor.
- cpu_id: 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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns G_FALSE (0) on failure (invalid arguments or no free PCBAssocLCB slot).

**Preconditions**

plcb must have been created by gu_SchedulingListCreate. cpu_id must be a valid processor ID.

**Postconditions**

The processor with the given cpu_id will service tasks in plcb at the specified priority.

**See also**

gu_SchedulingListAssociateTask
gu_SchedulingListCreate
gu_SchedulingListExclusionSet

---

### 13.4.32 gu_SchedulingListAssociateTask

**Prototype**

```
G_UINT32 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 theGS_LCBstructure 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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns G_FALSE on failure.

**Preconditions**

Both ptcb and plcb must be valid. Task should be in Waiting state.

**Postconditions**

The task is assigned to the scheduling list; it will compete for processors associated with that list.

**See also**

gu_SchedulingListAssociateProcessor
gu_SchedulingListCreate
gu_SchedulingListExclusionSet

---

## 13.4.33 gu_SchedulingListCreate

**Prototype**

```
GS_LCB *gu_SchedulingListCreate(enum lcbtype lcb_type);
```

**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 createdGS_LCBstructure. 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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns NULL if no free LCB is available.

**Preconditions**

lcbtype must be GS_LCBTypeFP (fixed priority) or GS_LCBTypeEDF (earliest deadline first).

**Postconditions**

A new scheduling list is allocated and linked into the kernel LCB list.

**See also**

gu_SchedulingListAssociateProcessor
gu_SchedulingListAssociateTask
gu_SchedulingListExclusionSet

---

### 13.4.34 gu_SchedulingListExclusionSet

**Prototype**

```
G_UINT32 gu_SchedulingListExclusionSet(GS_LCB *plcb, G_UINT32 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 theGS_LCBstructure 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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns G_FALSE on failure.

**Preconditions**

plcb must be a valid scheduling list. exclusion must be $\geq 0$ (0 means no limit).

**Postconditions**

At most exclusion processors may simultaneously execute tasks from this list (prevents real-time anomalies).

**See also**

gu_SchedulingListAssociateProcessor
gu_SchedulingListAssociateTask
gu_SchedulingListCreate

---

### 13.4.35 gu_SemaphoreCreateBinary

**Prototype**

```
GS_RCB *gu_SemaphoreCreateBinary(int initial_count);  /* gu_sem_create(1) */
```

**Description:** Creates a binary semaphore, and returns a handle by which the semaphore can be referenced. The semaphore is created in the 'empty' state, meaning the semaphore must first be given using the gu_Semaphore Give()

**Parameters**

The following parameter is required.

- **initial_count**: The count value assigned to the semaphore when it is created.

**Returns**
If the semaphore is created successfully then a handle to the semaphore is returned, NULL otherwise

**Thread-safety**
Safe from task context or initialization.

**Error conditions**
Returns `NULL` if no free RCB is available.

**Preconditions**
None.

**Postconditions**
A binary semaphore (initial count 1) is allocated. Use `gu_SemaphoreTake`/`gu_SemaphoreGive` to operate it.

**See also**
`gu_SemaphoreCreateMutex`
`gu_SemaphoreCreateRecursiveMutex`
`gu_SemaphoreGetCount`
`gu_SemaphoreGive`
`gu_SemaphoreTake`

---

### 13.4.36 gu_SemaphoreCreateMutex

**Prototype**

```
GS_RCB *gu_SemaphoreCreateMutex(void);  /* gu_sem_create(1), non-recursive */
```

**Description:** Creates a mutex, and returns a handle by which the created mutex can be referenced. Mutexes are taken using `gu_SemaphoreTake()`, and given using `gu_SemaphoreGive()`.

**Parameters**
The function takes no parameters.

**Returns**
If the mutex type semaphore was created successfully then a handle to the created mutex is returned, NULL otherwise

**Thread-safety**
Safe from task context or initialization.

**Error conditions**
Returns `NULL` if no free RCB is available.

**Preconditions**
None.

**Postconditions**
A non-recursive mutex semaphore is allocated with initial count 1.

**See also**
`gu_SemaphoreCreateBinary`
`gu_SemaphoreCreateRecursiveMutex`
`gu_SemaphoreGetCount`
`gu_SemaphoreGive`
`gu_SemaphoreTake`

---

### 13.4.37 gu_SemaphoreCreateRecursiveMutex

**Prototype**

```
GS_RCB *gu_SemaphoreCreateRecursiveMutex(void);
```

**Description:** Creates a recursive mutex, and returns a handle by which the mutex can be referenced. Recursive mutexes are 'taken' (obtained) using the gu_SemaphoreTakeRecursive() and given (released) using the gu_SemaphoreGiveRecursive() API functions respectively.

**Parameters**

The function takes no parameters.

**Returns**

If the mutex type semaphore was created successfully then a handle to the created mutex is returned, NULL otherwise

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns NULL if no free RCB is available.

**Preconditions**

None.

**Postconditions**

A recursive mutex is allocated; the same task may take it multiple times without deadlock.

**See also**

gu_SemaphoreCreateBinary
gu_SemaphoreCreateMutex
gu_SemaphoreGetCount
gu_SemaphoreGive
gu_SemaphoreTake

---

### 13.4.38 gu_SemaphoreGetCount

**Prototype**

```
G_UINT32 gu_SemaphoreGetCount(const GS_RCB *presource);
```

**Description:** Returns the count of a semaphore. The semaphore must have previously been created with a call to gu_SemaphoreCreateBinary(), gu_SemaphoreCreateMutex()orgu_SemaphoreCreateCounting().

**Parameters**

The function takes the following parameters.

- `presource`: A handle to the semaphore being taken - obtained when the semaphore was created.

**Returns**

If the semaphore is a counting semaphore then the semaphores current count value is returned. If the semaphore is a binary semaphore then 1 is returned if the semaphore is available, and 0 is returned if the semaphore is not available.

**Thread-safety**

Read-only; consistent result requires holding the mutex.

**Error conditions**
Undefined if `presource` is `NULL`.

**Preconditions**
`presource` must be a valid semaphore RCB.

**Postconditions**
None.

**See also**
`gu_SemaphoreCreateBinary`
`gu_SemaphoreCreateMutex`
`gu_SemaphoreCreateRecursiveMutex`
`gu_SemaphoreGive`
`gu_SemaphoreTake`

---

### 13.4.39 gu_SemaphoreGive

**Prototype**

```
G_UINT32 gu_SemaphoreGive(GS_RCB *presource);  /* replaces gu_sem_post() */
```

**Description:** The `gu_SemaphoreGive` function releases a semaphore or mutex previously acquired. If tasks are waiting to acquire the resource, the highest-priority waiting task will be granted the resource. If no tasks are waiting, the semaphore's internal count is incremented.

**Parameters**
The function accepts one parameter:
- `presource`: A pointer to the GS_RCB structure representing the semaphore resource. This pointer was returned by the gu_SemaphoreCreate function.

**Returns**
The `gu_SemaphoreGive` function returns G_TRUE if the semaphore was successfully released, and G_FALSE otherwise.

**Thread-safety**
Safe from task context.

**Error conditions**
Returns `G_FALSE` (0) if the semaphore count would exceed its maximum.

**Preconditions**
`presource` must be a valid semaphore. The calling task should currently hold the semaphore (for mutex use).

**Postconditions**
Semaphore count is incremented by 1; the highest-priority waiting task (if any) is released.

**See also**
`gu_SemaphoreCreateBinary`
`gu_SemaphoreCreateMutex`
`gu_SemaphoreCreateRecursiveMutex`
`gu_SemaphoreGetCount`
`gu_SemaphoreTake`

---

## 13.4.40 gu_SemaphoreTake

**Prototype**

```
G_UINT32 gu_SemaphoreTake(GS_RCB *presource, G_UINT64 ticks_to_wait);  /* replaces gu_sem_wait(): se
```

**Description:** The gu_SemaphoreTake function attempts to acquire a semaphore or mutex. If the resource's current count is greater than 0, the resource 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 ready priority is determined by its ready priority.

**Parameters**

The function uses the following parameters:

- presource: A pointer to the GS_RCB structure representing the semaphore resource, as returned by gu_SemaphoreCreate.
- ticks_to_wait: The time to wait in system time units. If zero, no time to wait an returns immediatly. Set G_LATEST_TIME if want to wait for until the resource become available.

**Returns**

The gu_SemaphoreTake 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.

**Thread-safety**

Task context only; may block.

**Error conditions**

Returns G_FALSE (0) if blocking is 0 and the semaphore is not available. Returns G_TRUE on success.

**Preconditions**

Must be called from a task. presource must be a valid semaphore RCB.

**Postconditions**

On success, semaphore count is decremented by 1.

**See also**

gu_SemaphoreCreateBinary
gu_SemaphoreCreateMutex
gu_SemaphoreCreateRecursiveMutex
gu_SemaphoreGetCount
gu_SemaphoreGive

---

## 13.4.41 gu_SignalCreate

**Prototype**

```
GS_SCB *gu_SignalCreate(enum scbtype type, G_UINT32 priority, void *pxcb, void *signal_code, void *si
```

**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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `NULL` if no free SCB is available.

**Preconditions**

`Signal_Code` must be a valid function pointer. `pxcb` must point to the control block (TCB, RCB, ECB, or KCB) to associate the signal with.

**Postconditions**

A signal is created and linked to the specified control block. It executes `Signal_Code` when the signal condition occurs.

**See also**

`gu_SignalDestroy`

---

## 13.4.42 gu_SignalDestroy

**Prototype**

```
G_UINT32 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.

**Thread-safety**

Safe from task context.

**Error conditions**

Returns `G_FALSE` if `pscb` is invalid.

**Preconditions**

`pscb` must be a valid signal control block previously created.

**Postconditions**

The signal is unlinked from its associated control block and its SCB is returned to the free pool.

**See also**

gu_SignalCreate

---

## 13.4.43 gm_ERETAddressSet

**Prototype**

```
void gm_ERETAddressSet(G_INT32 address);  /* writes ADDR_ERET_REG */
```

**Description:** The gm_ERETAddressSet macro sets the register where the return from interrupt code is stored in the internal registers of the GeMRTOS controller. When such an address is read, it means the processor is returning from interrupt and consequently the GeMRTOS controller Mutex is released, if it is not nested.

**Parameters**

The gm_ERETAddressSet has no parameter.

- value:The offset in the GeMRTOS Global Region where the return from interrupt instruction is located.

**Returns**

The gm_ERETAddressSet macro does not return any value.

**Thread-safety**

Kernel use only; must be called within a critical section.

**Error conditions**

None.

**Preconditions**

GeMRTOS mutex must be held. For kernel initialization only.

**Postconditions**

When the processor reads this address, the GeMRTOS controller automatically releases the mutex (unless nested).

**See also**

gm_MutexNestedValue
gm_ReadInputs
gm_StatusFrozenModeActiveGet
gm_WriteOutputs
gu_fprintf
gu_printf

---

## 13.4.44 gm_MutexNestedValue

**Prototype**

```
void gm_MutexNestedValue(G_UINT32 data);  /* static inline, IOWR to ADDR_CTRL_STATUS */
```

**Description:** The gm_MutexNestedValue function sets the GeMRTOS controller Mutex nested parameter according to the value. If value is equal to 0, the Mutex is released next time a released is performed. Otherwise, if value is not 0, the the Mutex is not released.

**Parameters**

The gm_MutexNestedValue has one parameter:

- data:If 0 the Mutex released is enabled, otherwise the Mutex released is disabled.

**Returns**
The gm_MutexNestedValue function does not return any value.

**Thread-safety**
Must be called within a critical section.

**Error conditions**
None.

**Preconditions**
GeMRTOS mutex must be held.

**Postconditions**
Mutex nesting behavior updated; allows the same processor to re-enter the critical section without deadlock when value $> 0$.

**See also**
gm_ERETAddressSet
gm_ReadInputs
gm_StatusFrozenModeActiveGet
gm_WriteOutputs
gu_fprintf
gu_printf

---

## 13.4.45 gm_ReadInputs

**Prototype**

```
G_INT32 gm_ReadInputs(void);  /* reads INOUTS register */
```

**Description:** The gm_ReadInputs macro transfer the data from the gemrtos_phy input conduit of GeMRTOS controller.

**Parameters**
The gm_ReadInputs macro does not require any parameter.

**Returns**
The gm_ReadInputs does returns the data from the gemrtos_phy input conduit of GeMRTOS controller.

**Thread-safety**
Read-only; safe within a critical section.

**Error conditions**
None.

**Preconditions**
GeMRTOS mutex should be held for a consistent read.

**Postconditions**
None.

**See also**
gm_ERETAddressSet
gm_MutexNestedValue
gm_StatusFrozenModeActiveGet
gm_WriteOutputs
gu_fprintf

gu_printf

## 13.4.46 gm_StatusFrozenModeActiveGet

**Prototype**

```
G_INT32 gm_StatusFrozenModeActiveGet;  /* macro: GRTOS_CMD_GET_FRZ_ACT */
```

**Description:** gm_StatusFrozenModeActiveGet returns the status of the frozen mode event (G_TRUE if active, G_FALSE if inactive).

**Parameters**

gm_StatusFrozenModeActiveGet has no parameter.

**Returns**

The gm_StatusFrozenModeActiveGet macro returns the time prescale.

**Thread-safety**

Read-only.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

None.

**See also**

gm_ERETAddressSet
gm_MutexNestedValue
gm_ReadInputs
gm_WriteOutputs
gu_fprintf
gu_printf

## 13.4.47 gm_WriteOutputs

**Prototype**

```
void gm_WriteOutputs(G_INT32 data);  /* gu_write_outputs(data): writes ADDR_LEDS */
```

**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.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

GeMRTOS mutex must be held.

**Postconditions**

The INOUTS register output bits are updated with `data`.

**See also**

`gm_ERETAddressSet`
`gm_MutexNestedValue`
`gm_ReadInputs`
`gm_StatusFrozenModeActiveGet`
`gu_fprintf`
`gu_printf`

---

## 13.4.48 gu_fprintf

**Prototype**

```
void gu_fprintf(const 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 returns G_TRUE.

**Thread-safety**

Thread-safe; uses internal semaphore serialization.

**Error conditions**

None returned; output may be silently dropped if the JTAG UART buffer is full.

**Preconditions**

JTAG UART for STDERR must be included in the Platform Designer project.

**Postconditions**

Formatted string sent to STDERR JTAG UART.

**See also**

`gm_ERETAddressSet`
`gm_MutexNestedValue`
`gm_ReadInputs`
`gm_StatusFrozenModeActiveGet`
`gm_WriteOutputs`
`gu_printf`

---

## 13.4.49 gu_printf

**Prototype**

```
void gu_printf(const 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 returns G_TRUE.

**Thread-safety**

Thread-safe; uses internal semaphore serialization.

**Error conditions**

None returned; output may be silently dropped if the JTAG UART buffer is full.

**Preconditions**

JTAG UART for STDIO must be included in the Platform Designer project.

**Postconditions**

Formatted string sent to STDIO JTAG UART.

**See also**

gm_ERETAddressSet
gm_MutexNestedValue
gm_ReadInputs
gm_StatusFrozenModeActiveGet
gm_WriteOutputs
gu_fprintf

---

## 13.4.50 gu_TaskCreate

**Prototype**

```
void *gu_TaskCreate(void *task_code, void *p_arg, const char *format, ...);  /* gu_GetTask() + config
```

**Description:** The gu_TaskCreate function creates a task with default settings and returns a pointer to its GS_TCB structure.Taskparameters 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.

**Thread-safety**

Safe from main() or task context.

**Error conditions**

Returns NULL if no free TCB or stack memory is available.

**Preconditions**

GeMRTOS must be initialized. TaskCode must be a valid function pointer. Stack size is set by the G_TASK_STACKSIZE compile-time constant (default 4096 bytes).

**Postconditions**

A new TCB is allocated; task is placed in Waiting state. Call gu_TaskTypeSet, gu_SchedulingListAssociateTask, and gu_TaskReadyPrioritySet to configure it before its first release.

**See also**

gu_TaskDelay
gu_TaskDelayTime
gu_TaskGetCurrentTCB
gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet
gu_TaskResume
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskSuspend
gu_TaskTypeSet

---

### 13.4.51 gu_TaskDelay

**Prototype**

```
G_UINT32 gu_TaskDelay(G_UINT32 hours, G_UINT32 minutes, G_UINT32 seconds, G_UINT32 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.

**Thread-safety**

Task context only; suspends the calling task.

**Error conditions**

Returns G_FALSE on error. All time arguments must be non-negative.

**Preconditions**

Must be called from a task. The time arguments must form a valid duration (e.g., seconds $< 60$).

**Postconditions**

Calling task is moved to Waiting state and released after the specified duration.

**See also**

gu_TaskCreate
gu_TaskDelayTime
gu_TaskGetCurrentTCB
gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet
gu_TaskResume
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskSuspend
gu_TaskTypeSet

---

## 13.4.52 gu_TaskDelayTime

**Prototype**

```
G_UINT32 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.

**Thread-safety**

Task context only.

**Error conditions**

Returns G_FALSE on error. ticks must be greater than 0.

**Preconditions**

Must be called from a task.

**Postconditions**

Calling task is suspended for ticks system-time units.

**See also**

gu_TaskCreate
gu_TaskDelay
gu_TaskGetCurrentTCB
gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet

gu_TaskResume
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskSuspend
gu_TaskTypeSet

---

## 13.4.53 gu_TaskGetCurrentTCB

**Prototype**

```
GS_TCB *gu_TaskGetCurrentTCB(void);  /* gu_PCB_GetCurrentTCB() */
```

**Description:** The gu_TaskGetCurrentTCB function retrieves a pointer to theTaskControl 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.

**Thread-safety**

Safe from task context.

**Error conditions**

Returns NULL if called from initialization context before any task is running.

**Preconditions**

None.

**Postconditions**

None.

**See also**

gu_TaskCreate
gu_TaskDelay
gu_TaskDelayTime
gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet
gu_TaskResume
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskSuspend
gu_TaskTypeSet

---

## 13.4.54 gu_TaskKill

**Prototype**

```
G_UINT32 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.

**Thread-safety**

Safe from task context.

**Error conditions**

Returns `G_FALSE` if `ptcb` is invalid or already killed.

**Preconditions**

`ptcb` must be a valid TCB. Killing a task that holds a mutex may cause deadlock; ensure resources are released first.

**Postconditions**

The task is destroyed in any state; its TCB and stack are freed.

**See also**

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

---

## 13.4.55 gu_TaskPeriodSet

**Prototype**

```
G_UINT32 gu_TaskPeriodSet(struct gs_tcb *ptcb, unsigned int hours, unsigned int minutes, unsigned int
```

**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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `G_FALSE` on invalid arguments.

**Preconditions**

`ptcb` must be valid. Task type should be `G_TCBType_Periodic` (or `G_TCBType_Periodic_Skip`).

**Postconditions**

The task period is updated; the next release time is recalculated.

**See also**

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

---

## 13.4.56 gu_TaskReadyPrioritySet

**Prototype**

```
G_UINT32 gu_TaskReadyPrioritySet(struct gs_tcb *ptcb, G_UINT64 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_UINT64 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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `G_FALSE` on error.

**Preconditions**

`ptcb` must be valid. Lower `priority` value means higher scheduling priority.

**Postconditions**

The task's ready-state priority is updated.

**See also**

`gu_TaskCreate`

gu_TaskDelay
gu_TaskDelayTime
gu_TaskGetCurrentTCB
gu_TaskKill
gu_TaskPeriodSet
gu_TaskResume
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskSuspend
gu_TaskTypeSet

---

### 13.4.57 gu_TaskResume

**Prototype**

```
G_UINT32 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.

**Thread-safety**

Safe from task context.

**Error conditions**

Returns G_FALSE if the task is not in a suspended (Waiting) state.

**Preconditions**

ptcb must be a valid TCB in Waiting state.

**Postconditions**

Task is moved to Ready state and may be dispatched to a processor.

**See also**

gu_TaskCreate
gu_TaskDelay
gu_TaskDelayTime
gu_TaskGetCurrentTCB
gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskSuspend
gu_TaskTypeSet

---

## 13.4.58 gu_TaskRunPrioritySet

**Prototype**

```
G_UINT32 gu_TaskRunPrioritySet(struct gs_tcb *ptcb, G_UINT64 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_UINT64 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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns G_FALSE on error.

**Preconditions**

ptcb must be valid.

**Postconditions**

The task's running-state (execution) priority is updated.

**See also**

gu_TaskCreate
gu_TaskDelay
gu_TaskDelayTime
gu_TaskGetCurrentTCB
gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet
gu_TaskResume
gu_TaskStartWithOffset
gu_TaskSuspend
gu_TaskTypeSet

---

## 13.4.59 gu_TaskStartWithOffset

**Prototype**

```
G_UINT32 gu_TaskStartWithOffset(GS_TCB *ptcb,unsigned int hours, unsigned int minutes, unsigned int s
```

**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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `G_FALSE` on invalid arguments.

**Preconditions**

`ptcb` must be valid. Task must be in Waiting state.

**Postconditions**

Task is scheduled for its first release at current_time + offset.

**See also**

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

---

## 13.4.60 gu_TaskSuspend

**Prototype**

```
G_UINT32 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.

**Thread-safety**

Safe from task context (including self-suspension).

**Error conditions**

Returns `G_FALSE` if the task is not in Running or Ready state.

**Preconditions**

ptcb must be valid. Task must be in Running or Ready state.

**Postconditions**

Task is moved to Waiting state; a new task is dispatched to the freed processor.

**See also**

gu_TaskCreate
gu_TaskDelay
gu_TaskDelayTime
gu_TaskGetCurrentTCB
gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet
gu_TaskResume
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskTypeSet

---

### 13.4.61 gu_TaskTypeSet

**Prototype**

```
G_UINT32 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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns G_FALSE on invalid type.

**Preconditions**

type must be one of: G_TCBType_OneShot, G_TCBType_Periodic, G_TCBType_Periodic_Skip, G_TCBType_ISR, G_TCBType_IDLE.

**Postconditions**

Task type is set; this affects scheduling and completion behavior.

**See also**

gu_TaskCreate
gu_TaskDelay
gu_TaskDelayTime
gu_TaskGetCurrentTCB

gu_TaskKill
gu_TaskPeriodSet
gu_TaskReadyPrioritySet
gu_TaskResume
gu_TaskRunPrioritySet
gu_TaskStartWithOffset
gu_TaskSuspend

---

## 13.4.62 gm_StatusCountingTimeGet

**Prototype**

```
G_INT32 gm_StatusCountingTimeGet(void);  /* GRTOS_GET_INTERVAL (first read resets counter) */
```

**Description:** gm_StatusCountingTimeGet returns the status of the TIME_HOLD bit (G_TRUE or G_FALSE). The execution of gm_TimeCountersHold holds the time counters in its current state. The execution of gm_TimeCountersUnhold unholds the time counters.

**Parameters**

gm_StatusCountingTimeGet has no parameter.

**Returns**

The gm_StatusCountingTimeGet macro returns the time prescale.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

None.

**Preconditions**

GeMRTOS mutex must be held for a meaningful reading.

**Postconditions**

The internal elapsed-time counter is reset to zero after this read.

**See also**

gm_StatusResetCountingTimeGet
gm_SystemTimePrescaleGet
gm_SystemTimePrescaleSet
gm_TimeIntervalGet
gu_SystemTotalTimeGet

---

## 13.4.63 gm_StatusResetCountingTimeGet

**Prototype**

```
G_INT32 gm_StatusResetCountingTimeGet(void);  /* reads and resets ELAPSE_TM register */
```

**Description:** gm_StatusResetCountingTimeGet return the status of the time reset counter bit (G_TRUE or G_FALSE). When G_TRUE, the time counters are reset and hold. The time counters are reset and hold reset, when executing gm_TimeCountersReset. The time counters are release to run again when execution gm_TimeCoutersUnreset.

**Parameters**
The `gm_StatusResetCountingTimeGet` macro has no parameter.

**Returns**
The `gm_StatusResetCountingTimeGet` macro returns the time prescale.

**Thread-safety**
Must be called within a critical section.

**Error conditions**
None.

**Preconditions**
GeMRTOS mutex must be held.

**Postconditions**
The elapsed-time counter is reset; the returned value is the interval since the last reset.

**See also**
`gm_StatusCountingTimeGet`
`gm_SystemTimePrescaleGet`
`gm_SystemTimePrescaleSet`
`gm_TimeIntervalGet`
`gu_SystemTotalTimeGet`

---

### 13.4.64 gm_SystemTimePrescaleGet

**Prototype**

```
G_INT32 gm_SystemTimePrescaleGet(void);  /* macro: GRTOS_CMD_GET_TIME_PRESCALE */
```

**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.

**Thread-safety**
Read-only; safe within a critical section.

**Error conditions**
None.

**Preconditions**
None.

**Postconditions**
None.

**See also**
`gm_StatusCountingTimeGet`
`gm_StatusResetCountingTimeGet`
`gm_SystemTimePrescaleSet`
`gm_TimeIntervalGet`
`gu_SystemTotalTimeGet`

## 13.4.65 gm_SystemTimePrescaleSet

**Prototype**

```
void gm_SystemTimePrescaleSet(G_INT32 scale);  /* macro: GRTOS_CMD_SET_TIME_PRESCALE(scale) */
```

**Description:** 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. The gm_SystemTimePrescaleGet macro returns the time prescale.

**Parameters**

The gm_SystemTimePrescaleSet has one parameter:

- scale:The prescale configuration.

**Returns**

The gm_SystemTimePrescaleSet macro does not return any value.

**Thread-safety**

Must be called within a critical section.

**Error conditions**

Undefined if scale is 0.

**Preconditions**

GeMRTOS mutex must be held. Typically set during initialization to achieve 10 MHz ticks from the system clock.

**Postconditions**

System time counter increments every scale clock cycles.

**See also**

gm_StatusCountingTimeGet
gm_StatusResetCountingTimeGet
gm_SystemTimePrescaleGet
gm_TimeIntervalGet
gu_SystemTotalTimeGet

## 13.4.66 gm_TimeIntervalGet

**Prototype**

```
G_INT32 gm_TimeIntervalGet(void);  /* macro: GRTOS_GET_INTERVAL */
```

**Description:** The gm_TimeIntervalGet macro resets the Interval Time Counter, returning the last value.

**Parameters**

The gm_TimeIntervalGet macro has no parameter.

**Returns**

The gm_TimeIntervalGet macro returns the time prescale.

**Thread-safety**

Read-only within a critical section.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

The interval counter is reset.

**See also**

gm_StatusCountingTimeGet
gm_StatusResetCountingTimeGet
gm_SystemTimePrescaleGet
gm_SystemTimePrescaleSet
gu_SystemTotalTimeGet

---

## 13.4.67 gu_SystemTotalTimeGet

**Prototype**

```
G_UINT64 gu_SystemTotalTimeGet(void);  /* GRTOS_CMD_SYS_TM_GET */
```

**Description:** gu_SystemTotalTimeGet returns the total system time. It is the time in non frozen mode plus the time in frozen mode.

**Parameters**

The gu_SystemTotalTimeGet macro has no parameter. It has to be noted that temporal constraints (deadline, period) are related to system time, obtained using the gu_SystemTimeGet function, which is the time in non frozen mode.

**Returns**

The gu_SystemTotalTimeGet macro returns the total system time.

**Thread-safety**

Safe from task context.

**Error conditions**

None.

**Preconditions**

None.

**Postconditions**

None.

**See also**

gm_StatusCountingTimeGet
gm_StatusResetCountingTimeGet
gm_SystemTimePrescaleGet
gm_SystemTimePrescaleSet
gm_TimeIntervalGet

---

## 13.4.68 gu_TriggerCreate

**Prototype**

```
GS_RCB *gu_TriggerCreate(int irq_id, G_UINT64 ticks_to_wait);
```

**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.
- `ticks_to_wait`: Timeout for trigger in number of ticks.

**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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `NULL` if no free RCB is available or `irq_id` is out of range.

**Preconditions**

`irq_id` must be in `[0, ALT_NIRQ-1]`. The IRQ must be connected in Platform Designer.

**Postconditions**

A trigger RCB is allocated and associated with `irq_id`. Call `gu_TriggerRegisterTask` to associate tasks.

**See also**

`gu_TriggerDisable`
`gu_TriggerDisableHook`
`gu_TriggerEnable`
`gu_TriggerEnableHook`
`gu_TriggerRegisterTask`
`gu_TriggerRelease`
`gu_TriggerSetTimeoutType`
`gu_TriggerWait`

---

## 13.4.69 gu_TriggerDisable

**Prototype**

```
G_UINT32 gu_TriggerDisable(GS_RCB *ptrigger);
```

**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:

- `ptrigger`: Handle of the trigger.

**Returns**

The function returns G_TRUE if the operation was successful.

**Thread-safety**

Safe from task context.

**Error conditions**

Returns `G_FALSE` if `ptrigger` is NULL or trigger not created.

**Preconditions**

Trigger must have been created with `gu_TriggerCreate`.

**Postconditions**

The IRQ is disabled; the trigger will not fire until re-enabled.

**See also**

`gu_TriggerCreate`
`gu_TriggerDisableHook`
`gu_TriggerEnable`
`gu_TriggerEnableHook`
`gu_TriggerRegisterTask`
`gu_TriggerRelease`
`gu_TriggerSetTimeoutType`
`gu_TriggerWait`

---

## 13.4.70 gu_TriggerDisableHook

**Prototype**

```
G_UINT32 gu_TriggerDisableHook(GS_RCB *ptrigger, void (*code_callback)(void *), 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:

- **`ptrigger`**: Handle of the trigger.
- **`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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `G_FALSE` on invalid arguments.

**Preconditions**

Trigger must exist. `code_callback` must be a valid function pointer or `NULL`.

**Postconditions**

`code_callback` is called (with `p_arg`) each time this trigger is disabled.

**See also**

gu_TriggerCreate
gu_TriggerDisable
gu_TriggerEnable
gu_TriggerEnableHook
gu_TriggerRegisterTask
gu_TriggerRelease
gu_TriggerSetTimeoutType
gu_TriggerWait

---

## 13.4.71 gu_TriggerEnable

**Prototype**

```
G_UINT32 gu_TriggerEnable(GS_RCB *ptrigger);
```

**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:

- ptrigger: Handle of the trigger.

**Returns**

The function returns G_TRUE if the operation was successful.

**Thread-safety**

Safe from task context.

**Error conditions**

Returns G_FALSE if ptrigger is NULL or trigger not created.

**Preconditions**

Trigger must have been created. Typically called after gu_TriggerWait returns to re-arm the trigger.

**Postconditions**

The IRQ is enabled; the next device interrupt will release registered tasks.

**See also**

gu_TriggerCreate
gu_TriggerDisable
gu_TriggerDisableHook
gu_TriggerEnableHook
gu_TriggerRegisterTask
gu_TriggerRelease
gu_TriggerSetTimeoutType
gu_TriggerWait

---

## 13.4.72 gu_TriggerEnableHook

**Prototype**

```
G_UINT32 gu_TriggerEnableHook(GS_RCB *ptrigger, void (*code_callback)(void *), 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:

- **ptrigger**: Handle of the trigger.
- **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.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns G_FALSE on invalid arguments.

**Preconditions**

Trigger must exist. code_callback must be a valid function pointer or NULL.

**Postconditions**

code_callback is called (with p_arg) each time this trigger is enabled.

**See also**

gu_TriggerCreate
gu_TriggerDisable
gu_TriggerDisableHook
gu_TriggerEnable
gu_TriggerRegisterTask
gu_TriggerRelease
gu_TriggerSetTimeoutType
gu_TriggerWait

---

## 13.4.73 gu_TriggerRegisterTask

**Prototype**

```
G_UINT32 gu_TriggerRegisterTask(struct gs_tcb *ptcb, GS_RCB *ptrigger);
```

**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.
- **ptrigger**: Handle of the trigger.

**Returns**

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

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns `G_FALSE` if `ptcb` is invalid or `ptrigger` has not been created.

**Preconditions**

`ptrigger` must exist (created by `gu_TriggerCreate`). `ptcb` must be valid.

**Postconditions**

The task is registered; when the trigger fires it will be released.

**See also**

`gu_TriggerCreate`
`gu_TriggerDisable`
`gu_TriggerDisableHook`
`gu_TriggerEnable`
`gu_TriggerEnableHook`
`gu_TriggerRelease`
`gu_TriggerSetTimeoutType`
`gu_TriggerWait`

---

## 13.4.74 gu_TriggerRelease

**Prototype**

```
GS_TRGStatus gu_TriggerRelease(GS_RCB *ptrigger);
```

**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:

- `ptrigger`: Handle of the trigger.

**Returns**

Returns the trigger status after activation (`GS_TRGStatus`).

**Thread-safety**

Safe from task context.

**Error conditions**

Returns `G_FALSE` if `ptrigger` is `NULL` or invalid.

**Preconditions**

Trigger must exist.

**Postconditions**

All tasks registered to this trigger are moved from Waiting to Ready state.

**See also**

`gu_TriggerCreate`
`gu_TriggerDisable`
`gu_TriggerDisableHook`
`gu_TriggerEnable`
`gu_TriggerEnableHook`
`gu_TriggerRegisterTask`
`gu_TriggerSetTimeoutType`

gu_TriggerWait

## 13.4.75 gu_TriggerSetTimeoutType

**Prototype**

```
G_UINT32 gu_TriggerSetTimeoutType(GS_RCB *ptrigger, GS_TRGTimeOutType type);
```

**Description:** Sets the trigger timeout type for a givenTrigger. This function assigns the specified timeout type to the provided task control block's TRGType member.

**Parameters**

The gu_TriggerSetTimeoutType requires the following parameters:

- **ptrigger**: Pointer to the trigger ( GS_RCB *) whose timeout type is to be set.
- **type**: The timeout type to be assigned (valid values are: GS_TRGTimeOutType_disabled, GS_TRGTimeOutType_restart_when_enable, GS_TRGTimeOutType_restart_when_timeout, GS_TRGTimeOutType_no_restart.

**Returns**

The function returns G_TRUE to indicate success.

**Thread-safety**

Safe from task context or initialization.

**Error conditions**

Returns G_FALSE on invalid arguments.

**Preconditions**

ptrigger must be a valid trigger RCB. Use before the first gu_TriggerWait.

**Postconditions**

The timeout behavior for waiting tasks is configured.

**See also**

gu_TriggerCreate
gu_TriggerDisable
gu_TriggerDisableHook
gu_TriggerEnable
gu_TriggerEnableHook
gu_TriggerRegisterTask
gu_TriggerRelease
gu_TriggerWait

## 13.4.76 gu_TriggerWait

**Prototype**

```
GS_TRGStatus 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**

Returns a GS_TRGStatus value: G_TRGStatus_triggered if the trigger fired, or G_TRGStatus_timeout if a timeout occurred.

**Thread-safety**

Task context only; suspends the calling task.

**Error conditions**

Returns status code indicating whether trigger fired (G_TRGStatus_triggered) or timed out.

**Preconditions**

Calling task must be registered to a trigger via gu_TriggerRegisterTask. Must be called from task context.

**Postconditions**

Task is moved to Waiting state; resumes when its registered trigger fires or a timeout occurs.

**See also**

gu_TriggerCreate
gu_TriggerDisable
gu_TriggerDisableHook
gu_TriggerEnable
gu_TriggerEnableHook
gu_TriggerRegisterTask
gu_TriggerRelease
gu_TriggerSetTimeoutType

## 13.4.77 gu_ConvertTime

**Prototype**

```
G_UINT64 gu_ConvertTime(unsigned int hours, unsigned int minutes, unsigned int seconds, unsigned int
```

**Description:** Converts a wall-clock duration into the system tick value used by every GeMRTOS timeout and period argument. Use it wherever a function expects a G_UINT64 tick count instead of computing ticks by hand.

**Parameters**

- hours — hours component of the duration.
- minutes — minutes component.
- seconds — seconds component.
- ms — milliseconds component.

**Returns** The equivalent duration expressed in system ticks.

## 13.4.78 gu_SemaphoreCreateCounting

**Prototype**

```
GS_RCB *gu_SemaphoreCreateCounting(int max_count, int initial_count);
```

**Description:** Creates a counting semaphore and returns a handle by which it can be referenced. A counting

semaphore grants up to max_count simultaneous holders, which suits pools of interchangeable resources.

**Parameters**
- max_count — maximum number of grants the semaphore may hold.
- initial_count — grants available immediately after creation.

**Returns** Handle to the new semaphore, or 0 if it could not be created.

### 13.4.79 gu_SemaphoreDestroy

**Prototype**

```
G_UINT32 gu_SemaphoreDestroy(GS_RCB *psemaphore);
```

**Description:** Destroys a semaphore and returns its resource control block to the free pool.

**Parameters**
- psemaphore — handle returned by a gu_SemaphoreCreate* function.

**Returns** G_TRUE on success, G_FALSE otherwise.

### 13.4.80 gu_MessageQueueDestroy

**Prototype**

```
G_UINT32 gu_MessageQueueDestroy(GS_RCB *prcb);
```

**Description:** Destroys a message queue and returns its resource control block to the free pool.

**Parameters**
- prcb — handle returned by gu_MessageQueueCreate.

**Returns** G_TRUE on success, G_FALSE otherwise.

### 13.4.81 gu_TriggerDestroy

**Prototype**

```
G_UINT32 gu_TriggerDestroy(GS_RCB *ptrigger);
```

**Description:** Destroys a trigger resource, returning its resource control block and its timeout event to the free pool.

**Parameters**
- ptrigger — handle returned by gu_TriggerCreate.

**Returns** G_TRUE on success, G_FALSE otherwise.