GeMRTOS: Multiprocessor RTOS

User and Development Manual — Chapter 9: Programming Guide

gemrtos.com

August 2026

Revision: 0.9a-2026-06-01

CHAPTER 9

GeMRTOS Programming Guide

9.1 Introduction

GeMRTOS provides an efficient and intuitive mechanism for implementing embedded multiprocessor systems on Altera Field-Programmable Gate Array (FPGA)s. Building upon familiar concepts from traditional Real-Time Operating System (RTOS), GeMRTOS allows developers to structure tasks as functions enhanced with real-time features, such as priority-based scheduling and synchronization. This approach facilitates a smooth transition for developers experienced with single-processor embedded systems, while unlocking the performance potential of multiprocessor architectures.

This programming guide delves into the core elements of GeMRTOS programming, offering practical insights and code examples to accelerate development. We will explore task management techniques, synchronization primitives (including semaphores and mutexes), and inter-process communication (IPC) mechanisms such as message queues, emphasizing the unique opportunities and challenges that arise in a multiprocessor context. Understanding these elements is crucial for creating responsive and reliable applications.

Beyond basic functionality, this guide also introduces advanced features such as trigger-based event handling and real-time scheduling policies that can be implemented with GeMRTOS, allowing developers to tailor their systems to meet specific performance requirements. Particular attention will be paid to utilizing these features in a way that reduces overhead and improves resource utilization across multiple processors.

By mastering the techniques and tools presented in this chapter, developers can effectively design, implement, and optimize robust, high-performance embedded applications with GeMRTOS, fully leveraging the parallel processing capabilities offered by Altera FPGA devices. This will allow them to confidently tackle complex, time-critical operations within the constraints of real-time systems.

9.2 Task Management

Task management is a cornerstone of the GeMRTOS architecture, enabling the efficient execution of multiple applications and tasks within a real-time environment. The operating system orchestrates these tasks by managing their state, priority, and resource allocation, ensuring critical tasks complete within defined time constraints. This complex process relies on data structures, algorithms, and interrupt mechanisms that define GeMRTOS's task management capabilities.

The Task Control Block (TCB) is the fundamental unit of task management. Each task has a TCB storing its state (ready, running, waiting), priority, stack pointer, and code location. Figure 9.1 illustrates the key components of a task.

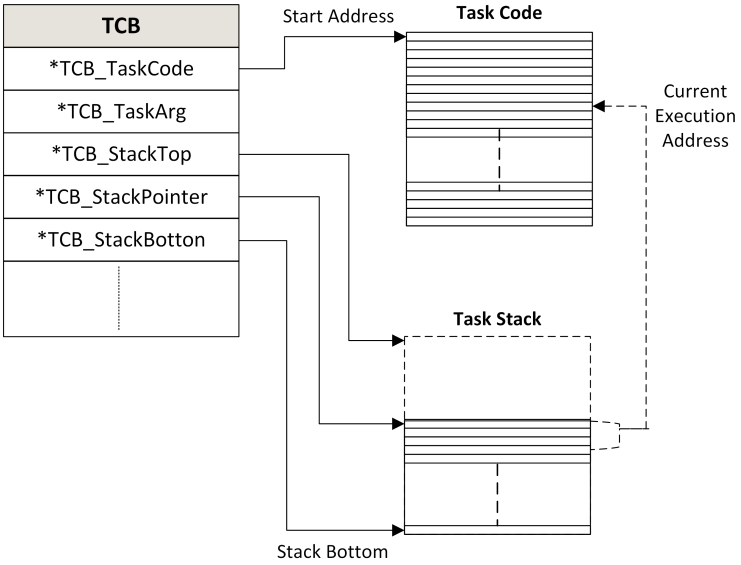

Figure 9.1: GeMRTOS Task Main Components

The TCB allows the operating system to:

- Track task progress.
- Manage task resources.
- Perform context switching.

Multiple linked lists (TCBRDYL - ready list, TCBRUNL - running list, TCBWL - waiting list) organize tasks based on their current state, enabling the scheduler to quickly identify and select the next task for execution.

GeMRTOS utilizes scheduling disciplines for each Hybrid scheduling list to determine task execution order based on priority. When a processor is available, the scheduler selects the highest-priority task from the TCBRDYL lists assigned to that processor. This involves context switching: saving the current state of the running task and restoring the state of the new task.

To ensure efficient resource distribution, GeMRTOS's timer-tickless architecture improves processor usage. Furthermore, processors are allocated according to task priorities, ensuring high-priority tasks are favored. By effectively managing task creation, scheduling, and processor allocation, GeMRTOS allows applications to meet stringent real-time requirements and delivers dependable performance.

9.2.1 Periodic and Infinite-Loop (One-Shot) Tasks

GeMRTOS provides two distinct approaches to task execution: periodic and infinite-loop tasks, and crucially supports both simultaneously. This empowers developers to build flexible and adaptable real-time embedded systems by leveraging the strengths of each task type:

- **Periodic Tasks:** These tasks execute repeatedly at a fixed time interval, crucial for applications demanding consistent and timely performance, such as sensor data collection, control algorithms, and periodic system monitoring. Developers configure these tasks with specific periods and deadlines, allowing GeMRTOS's scheduler to enforce temporal constraints and guarantee critical operations occur on time. Importantly,

even if an instance of a periodic task misses its deadline due to temporary system overload, the next scheduled instance will still be released periodically, providing a built-in mechanism for recovering from transient delays and ensuring long-term system stability.

- **Infinite-Loop Tasks:** These tasks execute continuously within a loop. Unlike basic implementations, well-designed infinite loop tasks within GeMRTOS voluntarily suspend themselves for a specified interval. By relinquishing processor control through deliberate suspension, these tasks allow the GeMRTOS scheduler to allocate processor to other tasks, ensuring system responsiveness. This controlled behavior enables infinite-loop tasks to efficiently operate in the background, performing non-time-critical operations or continuously polling for external events, without monopolizing the processor or compromising real-time performance.

Within GeMRTOS, true infinite-loop tasks, as traditionally understood, are not directly implemented. Instead, they are modeled as one-shot tasks that are executed only once. To achieve persistent, continuous execution, these tasks require explicitly including an infinite loop within their code. Without this infinite loop, the task executes only once and will not automatically restart. This loop structure, combined with the deliberate suspension mechanism, is key to ensuring their compatibility with real-time scheduling paradigms.

Figure 9.2 illustrates the time execution behavior of a periodic task versus an infinite-loop task:

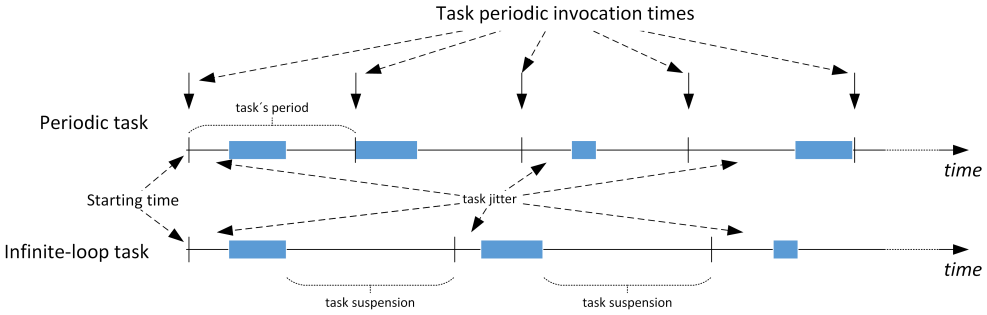

Figure 9.2: Periodic vs. Infinite-Loop Task Runtime Behavior

The harmonious co-existence of both task types is a key differentiator for GeMRTOS. Developers can strategically use periodic tasks for critical, time-bound functions, benefiting from their inherent resilience to occasional deadline misses, while simultaneously deploying infinite-loop tasks for background processing. This comprehensive approach requires thoughtful design to balance processor utilization and ensure the real-time requirements of periodic tasks are always met. By effectively managing the interplay between periodic and infinite-loop tasks, GeMRTOS facilitates the creation of highly efficient and responsive embedded systems tailored to a wide range of applications.

9.2.2 Task Creation and Configuration

Tasks in GeMRTOS are defined as C functions, offering a familiar programming model. The `hellogemrtos.c` template provides examples, showcasing the `\gls{task}\periodic` and `task_infinite_loop` functions.

Periodic Task Template

Code 9.1 shows a template for a periodic task, implemented as a C function called `task_periodic`. This function executes and terminates after each invocation, taking a `void*` argument for task-specific data.

```
1 void task_periodic (void* pdata)
2 {
3   /* code to be executed periodically with pdata as argument */
```

```
  /* Set a 600ms waiting timeout for LEDs Mutex */
  /* Define Mutex timeout (600ms) */
  G_UINT64 timeout_leds = gu_ConvertTime(0, 0, 0, 600);

  /* Request LEDs Mutex with timeout */
  if (gu_SemaphoreTake(mutex_leds, timeout_leds) == G_TRUE) {
    leds = leds ^ 1;         /* Invert LED 1          */
    gm_WriteOutputs(leds);    /* Write LEDs to GeMRTOS output */
    gu_SemaphoreGive(mutex_leds); /* Release Mutex        */
  } else {
    /* mutex_leds timeout */
    gu_printf("Semaphore not taken in \gls{task} periodic\n");
  }
}
```

Code 9.1: Periodic Task Template

The task configuration is performed within the application's `main()` function (Code 9.2). First, the task is created using `gu_TaskCreate`, specifying the task name (`task_periodic`), an argument (arbitrarily 1), and a descriptive name ("periodic task").

If the task creation is successful (returning a non-null pointer to the Task Control Block (TCB) of the created task), the code specifies that the task is periodic by executing the primitive `gu_TaskTypeSet(ptcb, G_TCBType_Periodic)`. Following this, the ready and run priorities are specified using `gu_TaskReadyPrioritySet` and `gu_TaskRunPrioritySet`, respectively. Finally, the task's period is set to 1 second using `gu_TaskPeriodSet`, and the first invocation is set to 5 seconds after the system's initial time using `gu_TaskStartWithOffset`.

If task creation is unsuccessful, `gu_TaskCreate` returns `G_FALSE`, and the code halts, printing an error message.

```
t_semaphore_resource *mutex_leds;

int main(void)
{
  // System configuration
  mutex_leds = gu_SemaphoreCreateRecursiveMutex();
  ...
  GS_TCB *ptcb = gu_TaskCreate((void *) task_periodic, /* Task code     */
  (void *) 1,        /* Call argument  */
  "periodic \gls{task}");      /* Task description */

  if (ptcb != (GS_TCB *) 0) { /* Task was created successfully */
    gu_TaskTypeSet(ptcb, G_TCBType_Periodic); /* Set the \gls{task} as periodic */

    gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 1); /* Set READY priority */
    gu_TaskRunPrioritySet(ptcb,(G_UINT64) 1);  /* set RUN priority   */

    gu_TaskPeriodSet(ptcb, 0, 0, 1, 0); /* Set \gls{task} period (1 second) */

    gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0); /* Set starting time */

  } else {
    gu_printf("Error when creating \gls{task}\n");
```

```
    while(1);
  }
  ...
}
```

Code 9.2: Periodic Task Configuration Template

To complete this example, the Mutex `mutex_leds` (of type `t_semaphore_resource*`) must be declared and created using `gu_SemaphoreCreateRecursiveMutex()`.

Infinite-Loop Task Template

Code 9.3 presents a template for an infinite-loop task, implemented as a C function called `task_infinite_loop`. These tasks are invoked at startup and must contain an infinite loop with a suspension for periodic repetition. Within the loop (implemented using `while(1)`), the task suspends its execution for a defined period using `gu_TaskDelay` or `gu_TaskDelayTime`. In this example, the task toggles the state of `LED 1` using the `XOR` function:

```
void task_infinite_loop(void* pdata)
{
  while(1) {
    /* Set a 600ms waiting timeout for LEDs Mutex */
    /* Define Mutex timeout (600ms) */
    G_UINT64 timeout_leds = gu_ConvertTime(0, 0, 0, 600);

    /* Request LEDs Mutex */
    if (gu_SemaphoreTake(mutex_leds, timeout_leds) == G_TRUE) {
      leds = leds ^ 2;              /* Invert LED 1                 */
      gm_WriteOutputs(leds);        /* Write LEDs to GeMRTOS output */
      gu_SemaphoreGive(mutex_leds); /* Release Mutex                */
    } else {
      /* mutex_leds timeout  */
      gu_printf("Semaphore not taken in \gls{task} periodic\n");
    }
    gu_TaskDelay( 0, 0, 1, 0);      /* Suspend \gls{task} 1 second        */
  }
}
```

Code 9.3: Infinite-Loop Task Template

Configuration for infinite-loop tasks is similar to periodic tasks, except tasks must be configured as `G_TCBType_OneShot` type and without configuring a period (Code 9.4).

```
...
ptcb = gu_TaskCreate((void *) task_infinite_loop,/* Task code         */
                     (void *) 2,                 /* Call argument     */
                     "infinite-loop \gls{task}");      /* \gls{task} description */

if (ptcb != (GS_TCB *) 0) { /* Task was created successfully */
  /* Set the \gls{task} as one-shot */
  gu_TaskTypeSet(ptcb, G_TCBType_OneShot);

```

```
10          gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 0);  /* Ready priority */
11          gu_TaskRunPrioritySet(ptcb,(G_UINT64) 0);     /* Run priority   */
12          gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0);    /* Starting time  */
13        } else {
14          gu_printf("Error when creating \gls{task}\n");
15          while(1);
16        }
17        ...
```

Code 9.4: Infinite-Loop Task Configuration Template

The Mutex `mutex_leds` is declared and created as described for periodic tasks.

Effective task management is paramount for building robust and responsive real-time systems. Understanding different task types (periodic vs. infinite-loop), and task configuration are fundamental for developers. By managing tasks, GEMRTOS can fully benefit from its multitasking engine, addressing many applications with high performance and scalable requirements. This leads to low overhead.

9.3 GeMRTOS Triggers: Event and Interrupt Management

Effective event and interrupt management is crucial for real-time systems. Triggers, a generalized interrupt handler in GeMRTOS, allows the system to respond effectively and predictably to internal and external stimuli. Interrupts provide a mechanism for hardware components to signal immediate attention, while events represent significant system occurrences.

Uniprocessor vs. Multiprocessor Handling:

- In a uniprocessor configuration, when an interrupt is triggered, the processor immediately executes the Interrupt Service Routine (ISR), saving the current context.
- In a multiprocessor GeMRTOS system, this architecture allows for more flexibility. A dedicated processor can be assigned to rapidly attend to high-priority events, while other external interrupts are handled as general system events by other processors. This distributes the interrupt load and improves overall responsiveness.

In GeMRTOS, Triggers represent a generic and flexible mechanism for responding to diverse events, encompassing traditional interrupt handling and extending it to include software-generated events. Triggers allow the system to react predictably and efficiently to both hardware and software stimuli, facilitating complex system-level coordination and enabling customized responses to specific situations.

The use of Triggers promotes modularity and separation of concerns within GeMRTOS. Components can be designed to react to specific Triggers without intimate knowledge of the underlying event sources. This simplifies system design, improves maintainability, and enables dynamic reconfiguration of event responses, leading to a more adaptable and robust operating system.

Note: Triggers are a fundamental GeMRTOS resource. The following sections provide detailed information on creating, configuring, and using triggers in your applications.

9.4 GeMTOS Resources

In GeMRTOS, resources are vital for managing shared data and peripherals, avoiding conflicts, and maintaining data integrity among multiple tasks. The system offers resource management mechanisms that enable developers

to build safe and efficient concurrent applications. These include controlled access, synchronization primitives, and safeguards against race conditions and deadlocks.

This section explores these mechanisms, detailing the use of semaphores, mutexes, message queues, and trigger to manage shared resources effectively and ensure the correct execution of concurrent operations. By carefully managing resources, developers can ensure that their applications are stable, efficient, and meet their real-time requirements.

9.4.1 Semaphores and Mutexes

Semaphores and Mutexes are fundamental resources in GeMRTOS, providing mechanisms for tasks to synchronize and prevent concurrent access to critical sections. This ensures data integrity and prevents race conditions. GeMRTOS supports binary and counting semaphores, as well as mutexes, offering flexibility for managing shared resources.

Mutexes, or mutual exclusion locks, are synchronization primitives that enforce exclusive access to shared resources, preventing data corruption and race conditions. GeMRTOS provides both non-recursive and recursive mutexes to support different synchronization needs.

Semaphores and mutexes store their state information within Resource Control Blocks (RCBs).

RCB Structure Extension Semaphores and mutexes share a similar structure, which is defined as an extension of the RCB:

```
struct T_SEMAPHORE_RESOURCE {
  TIMEPRIORITY  SEM_GrantedPriority;
  G_UINT32  SEM_Current_Count;
  G_UINT32  SEM_Maximum_Count;
  G_UINT32  SEM_Recurrence;
};
```

When you create a semaphore or mutex, the initial values of the RCB control structure are defined based on the resource type.

Counting Semaphores Counting semaphores manage resources with multiple instances or permit a certain number of concurrent accesses. The semaphore's count represents available resource instances or "takes". Tasks can acquire a "take" (decrementing the count) when available, or "give" a permit (incrementing `SEM_Current_Count`) when finished. If no permits are available, tasks can optionally wait. This makes counting semaphores suitable for shared resources such as buffers or multiple hardware peripheral instances.

Creating a Counting Semaphore A counting semaphore is created using the following function:

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

This function initializes the RCB fields as follows:

```
prcb->RCBType = GS_RCBType_SEM_COUNTING;
prcb->semaphore.SEM_GrantedPriority.i64 = (G_UINT64) G_LOWEST_PRIORITY;
// Initialize the counters
prcb->semaphore.SEM_Maximum_Count = (G_UINT32) max_count;
prcb->semaphore.SEM_Current_Count = (G_UINT32) initial_count;
```

The initial count sets the number of available permits at creation.

Binary Semaphore Binary semaphores synchronize two or more tasks by coordinating execution and ensuring proper access to shared resources. For example, Task A starts by performing a "take" operation on a binary semaphore. If the semaphore is 1, Task A proceeds; if 0, Task A is blocked until the semaphore is "given." Task B may "give" the semaphore when it requires Task A to execute its semaphored code.

Creating a Binary Semaphore A binary semaphore is created using:

```
GS_RCB *prcb = gu_SemaphoreCreateBinary(int initial_count);
```

The creation function initializes these RCB fields:

```
prcb->RCBType = GS_RCBType_SEM_BINARY;
prcb->semaphore.SEM_GrantedPriority.i64 = (G_UINT64) G_LOWEST_PRIORITY;
// Initialize the counters
prcb->semaphore.SEM_Maximum_Count = (G_UINT32) 1;
prcb->semaphore.SEM_Current_Count = (G_UINT32) initial_count;
```

where `initial_count` is either 0 (initially unavailable) or 1 (initially available).

Synchronization Example Consider two tasks: Task A produces data stored in memory, and Task B processes that data. After Task B processes the data, Task A produces new data, restarting the cycle.

```
GS_RCB *sem_produce = gu_SemaphoreCreateBinary(1);
GS_RCB *sem_process = gu_SemaphoreCreateBinary(0);
```

The `sem_produce` semaphore initializes available for taking, while `sem_process` is initially unavailable.

Tasks A and B are synchronized as follows:

```
void task_A(void *data)
{
  while (1) {
    gu_SemaphoreTake(sem_produce, G_LATEST_TIME); // Take sem_produce
    // or wait indefinitely

    // Code to produce data
```

```
    gu_SemaphoreGive(sem_process); // Signal data is ready
  }
}
```

Task A

```
void task_B(void *data)
{
  while(1) {
    gu_SemaphoreTake(sem_process, G_LATEST_TIME);  // Take sem_process, or wait

    // Code to process data

    gu_SemaphoreGive(sem_produce); // Signal data production is free
  }
}
```

Task B

Task A starts by taking `sem_produce`. Task B starts but waits for `sem_process` because it starts unavailable. Task B resumes when Task A finishes data production and gives `sem_process`. Task A waits for `sem_process` to become available again after produce a new data. The synchronization happen in the same way.

Warning: If the task granting the semaphore is a periodic task, it may be aborted, preventing successful subsequent takes because the semaphore will be held indefinitely. In such cases, configure periodic tasks without abortion or implement them as infinite-loop tasks.

Non-Recursive Mutex

A non-recursive mutex allows only one task to acquire the lock at a time and does not support recursive locking. Attempting to acquire the mutex again by the same task results in an error.

These mutexes are useful for ensuring exclusive access to shared resources and enforcing strict locking. They are suitable for situations where re-entrant locking is unnecessary.

Creating a Non-Recursive Mutex A non-recursive mutex is created with:

```
  GS_RCB *prcb = gu_SemaphoreCreateMutex(void);
```

The creation function initializes the following RCB fields:

```
  prcb->RCBType = GS_RCBType_SEM_MUTEX;
  prcb->semaphore.SEM_GrantedPriority.i64 = (G_UINT64) G_LOWEST_PRIORITY;
  // Initialize the counters
  prcb->semaphore.SEM_Maximum_Count = (G_UINT32) 1;
  prcb->semaphore.SEM_Current_Count = (G_UINT32) 1;
```

The initial count indicates that the mutex is initially available. The mutex priority is initially the lowest. When a higher-priority task acquires the mutex, its priority is elevated to match the task's priority, using the priority inheritance protocol. This discourages lower-priority tasks from preempting the mutex holder, reducing wait times and improving system efficiency.

Recursive Mutex

Recursive mutexes allow the same task to acquire the lock multiple times. The mutex tracks the number of acquisitions by the task. The task must call the "give" operation an equivalent number of times to fully release the mutex.

Recursive mutexes are useful in scenarios where a task enters multiple nested critical sections protected by the same mutex, preventing self-deadlock and simplifying design. However, they require careful usage to avoid increased complexity or contention.

Creating a Recursive Mutex A recursive mutex is created with:

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

The creation function initializes these RCB fields:

```
prcb->RCBType = GS_RCBType_SEM_MUTEX;
prcb->semaphore.SEM_GrantedPriority.i64 = (G_UINT64) G_LOWEST_PRIORITY;
// Initialize the counters
prcb->semaphore.SEM_Maximum_Count = (G_UINT32) 1;
prcb->semaphore.SEM_Current_Count = (G_UINT32) 1;
prcb->semaphore.SEM_Recurrence = 0;
```

The initial count indicates the mutex is initially available. The recurrence field tracks the number of "take" and "give" operations performed by the same task. The initial priority of the Mutex is also the lowest, and inherits the priority of the task. This is intended to reduce and avoid priority inversions.

Event Control Blocks (ECBs) and Semaphores/Mutexes Each task interacting with a semaphore or mutex has an associated resource event, potentially with a timeout. Figure 9.3 illustrates possible ECB states.

The resource event can be in one of three states:

- `G_ECBType_SEM_GRANTED`: The ECB is linked to the granted list of the mutex RCB (semaphore does not link the granted event). This event has no associated timed event.
- `G_ECBType_SEM_WAITING`: The ECB is linked to the waiting list of the semaphore/mutex RCB and has a timed event to define the timeout.
- `G_ECBType_SEM_WAITING_EXPIRED`: The ECB is not linked to any RCB lists. The resource event in `G_ECBType_SEM_WAITING` state changes to this state when a timeout occurs. The task resumes after determining the timed event occurred.

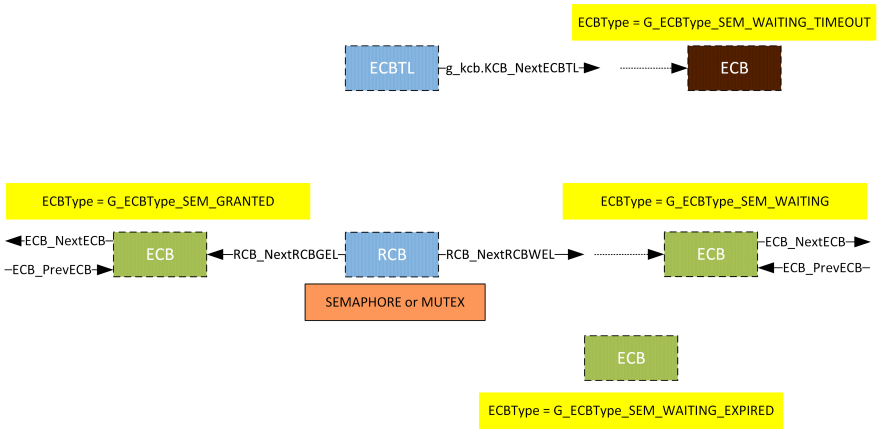

Figure 9.3: ECB States for Semaphore/Mutex Resources

The timed event is linked to the ECB Time List (ECBTL) when a resource event is linked to the waiting list. The timed event is of type `G_ECBType_SEM_WAITING_TIMEOUT`.

Events Associated with Semaphores and Mutexes The state of a semaphore or mutex changes due to runtime events. These events are now described:

- Execution of `gu_SemaphoreTake`: A task requests the semaphore/mutex.
  - If the resource is available (count > 0), the task takes it, and the count is decremented (non-recursive mutex/semaphore). Recurrence is also increased on recursive mutexs.
  - If unavailable, and it's not a recursive mutex: `G_FALSE` is returned if no timeout (timeout $= 0$). Otherwise, a timed event links to the ECB Time List (ECBTL).
- Execution of `gu_SemaphoreGive`: A task releases the resource.
  - For a non-recursive type, the count is incremented
  - For a recursive, the Recurrence decreases and the count decreases when recurrence equals 0. The first in the waiting events takes the semaphore/mutex resource.
- Timed Event Occurrence: When the timed event occurs:
  - The task, with the timeout is unlinked to waiting list(RCBWEL).
  - The event is set to `G_ECBType_SEM_WAITING_EXPIRED`.
  - The timed event is linked to the free ECB List (ECBFL).
  - The tasks, with the timeout unlinked, is linked to the ready list (TCBRDYÑ)
- Task Aborted: Aborted tasks require what is associated should be killed
  - In `G_ECBType_SEM_GRANTED`: Recurrence is set to 0, if the mutex is a non-recurrent one.
  - In `G_ECBType_SEM_WAITING`: The resource event and the associated event from time event are unlinked, and set free.
  - In `WAITING_EXPIRED`: The resource event is linked to the free ECB list.
  - In `G_ECBType_SEM_WAITING_EXPIRED`: The unlinked event, and the timed event are linked to free ECB list.
- A Task Waits for a Resource, and it becomes Available: The event is unlinked from the RCB waiting list, unlinked from the time list, set free, and granted to the task, so that it can resume.

When a task grants the resource:

- if it is a mutex:
  - If the task's priority is higher than the mutex's priority, the mutex's priority is elevated to match the task's priority. Otherwise, the task's priority is raised to match the mutex's priority.

  - If it is a non-recurrent mutex, the current count is decreased.
  - If it is a recursive mutex, the recurrence count is incremented. If the recurrence count was previously zero, the current count is decremented..
- if it is a semaphore, then no `G_ECBType_SEM_GRANTED` resource event type event is associated with the resource.

The events associated with semaphores/mutexes involve system primitives, timed events, and task abortion. These events may result in waiting tasks acquiring the semaphore/mutex when it is given.

Semaphore/Mutex Functions Table 9.1 lists the system primitives available for different kinds of semaphores and mutexes.

Table 9.1: System Primitives for Semaphore and Mutex Creation and Management.

Function	Description
`gu_SemaphoreCreateCounting`	Creates a counting semaphore with an initial and maximum count. The semaphore is available if the current count is greater than zero.
`gu_SemaphoreCreateBinary`	Creates a binary semaphore with an initial count of 1 (available) or 0 (unavailable).
`gu_SemaphoreCreateMutex`	Creates a non-recursive mutex.
`gu_SemaphoreCreateRecursiveMutex`	Creates a recursive mutex.
`gu_SemaphoreTake`	Requests a semaphore or mutex. If a timeout is defined, the task waits until the resource is available.
`gu_SemaphoreGive`	Releases a semaphore or mutex previously acquired.

These semaphore and mutex functions are simple yet flexible for task synchronization. The specific behavior of "take" and "give" operations depends on the resource type (semaphore or mutex).

Semaphore and Mutex Conclusions Semaphores—and mutexes—are fundamental synchronization tools in embedded real-time programming, providing essential mechanisms for managing access to shared resources and coordinating task execution. Semaphores, with their ability to serve as resource counters or signaling mechanisms, offer flexibility in designing synchronization strategies, while mutexes specifically ensure mutual exclusion, allowing only one task to access a critical section at a time. Understanding how to effectively utilize both primitives—and recognizing their appropriate applications—is key to developing systems that are both efficient and free of issues such as deadlocks and race conditions.

Moreover, proper implementation of semaphores and mutexes requires careful attention to initialization and operation within the system's context. When used correctly, these tools improve system stability, optimize resource utilization, and enhance predictable task scheduling. Mastery of their concepts and proper application is crucial for developers designing reliable, concurrent, and real-time systems, ultimately contributing to safer and more robust software solutions.

9.4.2 Message Queues

Inter-task communication is often necessary to exchange information between two or more tasks. This information transfer can be efficiently accomplished through message passing. A message typically consists of two components:

a pointer to data and a variable indicating the size of the referenced data. It's crucial that both the sender and receiver agree on the content and interpretation of the information within the message.

Figure 9.4 illustrates the flow of messages in a message queue system.

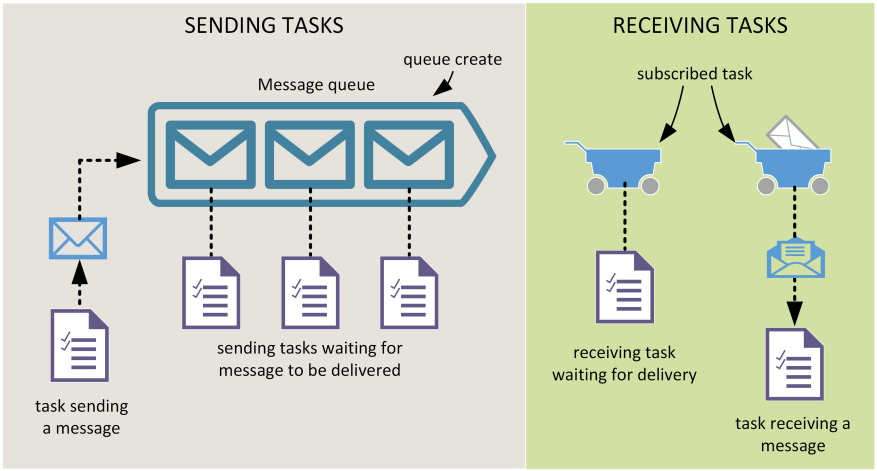

Figure 9.4: Message Queue flow

Message queues are system resources used for passing messages between tasks. In this communication model, there are tasks that send messages and tasks that receive them. The message queue stores all messages in the order they are sent.

When receiving tasks are ready, messages are read from the queue in sequence. After all designated receiving tasks have read a message, it is removed from the queue. The sending task is typically resumed once its message has been delivered by all receiving tasks.

It's important to note that a message remains in the queue until all intended receiving tasks have read it.

Key points about message queues:

- They facilitate asynchronous communication between tasks.
- Messages are queued in the order they are sent (FIFO - First In, First Out).
- Multiple tasks can send to and receive from the same queue.
- Messages persist in the queue until all receiving tasks have received them.

To manage message queues in GeMRTOS, there are five primary functions:

- `gu_MessageQueueCreate`: Configures a GEMRTOS resource as a message queue.
- `gu_MessageQueueSubscribe`: Subscribes a task to the messages sent to the message queue. Messages are intended to be delivered to all subscribed tasks.
- `gu_MessageQueueSend`: A task sends messages to the message queue, allowing other tasks to receive it.
- `gu_MessageQueuePrintf`: Similar to standard `printf`, a task sends formatted messages to the queue.
- `gu_MessageQueueReceive`: A task reads messages from the message queue to which it is subscribed.

Data Structures for Message Queues The following data structures are key to understanding how message queues are managed:

- `T_QUEUE_RESOURCE` is the main structure for managing the message queue.

```
struct T_QUEUE_RESOURCE {
  G_UINT32 MQ_subscribers; // Number of receive subscribers
  G_UINT32 MQ_seq_send;    // Sequence number for the next ECB to send
```

```
  G_UINT32 MQ_msg_seq;      // Sequence number of the current message
};
```

- Message queue information is linked to an RRDS control block, it is also needed in order to identify what the source and consumers are. The structure for its operation depends on if it is for senders, either receivers:
  - Data Structure of the sender is:

```
struct rrds_mq_sender {
  G_UINT32 deliveries;    // Times the message was delivered
  G_UINT32 consumers;     // Number of consumer should be delivered
  G_UINT32 MQ_msg_seq;    // Identification of the message
  char     *pmsg;         // Pointer to the message
  G_UINT32 msg_length;    // Length of the message
};
```

  - Data Structure for consumers:

```
struct rrds_mq_receiver {
  G_UINT32 MQ_msg_seq;    // Last of sequence of sender received
  GS_ECB *pproducer;      // Pointer to the event of the current sender
  char *buffer_msg;       // Pointer to the receive buffer
  G_UINT32 buffer_length;// Length of the receive buffer
  G_UINT32 msg_length;    // Length of the producer message (0 if not message
      produced)
};
```

ECBs always store the information by linking the corresponding RRDS control block..

Creating a Message Queue To set up a message queue, an RCB control block is allocated to manage it. The waitlist of the RCB is used for sender tasks, while the grant list applies to receiver tasks. The function to create a message queue is:

```
GS_RCB *prcb = gu_MessageQueueCreate(void);
```

When executed, the RCB is configured:

```
prcb->RCBType = (G_UINT32) GS_RCBType_QUEUE;
prcb->queue.MQ_subscribers = (G_UINT32) 0;
prcb->queue.MQ_msg_seq = (G_UINT32) 0;
prcb->queue.MQ_seq_send = (G_UINT32) 0;
```

The RCB block is a queue and contains a sequence number to determine what will be sent and what has been sent.

The gu_MessageQueueCreate function returns a GS_RCB * pointing to the queue. All operations with the queue requires this pointer, but a NULL return indicates that the messaga queue was not created. This will occur when there is insufficient memory.

Subscribing to a Queue Every task that wants to receive messages from a message queue must subscribe to the queue. Upon subscription, the queue increments the count of required message deliveries by updating the subscriber count (prcb->queue.MQ_subscribers++).

The function used for subscribing is defined as follows:

```
GS_ECB *gu_MessageQueueSubscribe(GS_RCB *presource, void *buffer_msg, G_UINT32
    buffer_length)
```

This function links an ECB control block of type G_ECBType_QUEUE_RECEIVER to the granting event list (RCBEGL) of the RCB control block of the message queue. It returns a pointer to the ECB control block associated with the granting ECB list (RCBEGL) of the message queue. A NULL return value indicates an error. The receiving buffer is passed as a pointer along with its length.

Sending a Message To send a message, the sender task should execute the following function:

```
gu_MessageQueueSend(GS_RCB *prcb, char *pmsg, int msg_length, G_UINT64 timeout)
```

The gu_MessageQueueSend function transmits the message to all subscriber tasks of the message queue. It returns the count of successful deliveries. If no tasks are subscribed, the function returns zero.

When there are subscribers, an ECB control block is linked to the RCB waiting list (RCBWEL) along with a rrds_mq_sender structure that holds information related to the message. A sequence number is assigned, and the current sequence number in the RCB control block is incremented.

If a timeout is specified (i.e., timeout is not zero), an ECB of type G_ECBType_MESSAGE_SEND_TIMEOUT is linked to the timed event list (ECBTL). Subsequently, all ECB control blocks in the granted list of the RCB control block waiting to receive a message (G_ECBType_MESSAGE_RECEIVE_WAIT) change their associated task state to ready.

The sender task is then suspended until the message is delivered to all receivers or until the timeout occurs. The function returns the number of deliveries produced. A positive return indicates delivery to all subscribed receivers, while a negative return indicates partial delivery.

An alternative function for sending messages is:

```
gu_MessageQueuePrintf(GS_RCB *prcb, char *format, ...);
```

This function accepts a message in a format similar to the C printf function. It's useful for sending messages that include variable values within a string, without requiring string manipulation functions.

Receiving Messages Subscribed tasks can receive messages from the message queue by executing the following function:

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

The gu_MessageQueueReceive function retrieves messages from the previously registered queue. If the buffer size provided during registration is smaller than the incoming message, the message will be received in multiple segments. Once the final segment is received, the sender task is resumed by setting its state to ready.

When the last segment of the message is received, the sender task is resumed by changing its state to ready.

Return Values and Error Codes

Table 9.2: Message Queue API return values.

Function	Return type	Meaning
gu_MessageQueueCreate	GS_RCB *	Non-NULL pointer to the new queue; NULL if the RCB pool is exhausted.
gu_MessageQueueSubscribe	GS_ECB *	Non-NULL receiver ECB; NULL if the ECB or RRDS pool is exhausted.
gu_MessageQueueSend	int	Number of subscriber tasks to which the message was successfully delivered. Returns 0 if no tasks are subscribed or if the ECB pool is exhausted.
gu_MessageQueueReceive	int	Number of bytes copied into the subscriber buffer on this call. Returns 0 if the queue was destroyed while the task was waiting.
gu_MessageQueuePrintf	int	Same semantics as gu_MessageQueueSend applied to the formatted string.
gu_MessageQueueDestroy	G_UINT32	G_TRUE on success; G_FALSE if active subscribers or senders still reference the queue.

Message Queue conclusion Message queues provide a powerful mechanism for inter-task communication within GeMRTOS, offering a flexible and reliable way to exchange data between different parts of your application. By understanding the core functions and data structures involved in creating, subscribing to, sending, and receiving messages, developers can effectively harness message queues to build complex and well-coordinated systems.

Proper message queue management involves careful consideration of factors such as buffer sizes, synchronization requirements, and potential error conditions. Sufficient buffer space should be allocated to prevent message loss. Handling potential errors, such as queue overflow or timeout situations, is also essential for creating robust and dependable communication pathways.

In designing your applications, consider carefully whether a message queue or other inter-process communication mechanisms, such as semaphores or mutexes, are the most appropriate for your specific needs.

9.4.3 GeMRTOS Triggers: Event and Interrupt Management

In real-time embedded systems, efficient event handling is paramount. GeMRTOS provides a powerful mechanism for this purpose called triggers. Triggers act as generalized interrupt handlers, allowing system components to react promptly and predictably to both internal and external events.

Triggers enable a decoupling of event detection and response. A trigger is essentially a resource that tasks can register with. When the associated event occurs, all registered tasks are signaled, allowing them to execute the appropriate response logic. This eliminates the need for continuous polling and promotes efficient resource utilization.

GeMRTOS offers a comprehensive set of functions for creating, configuring, and managing triggers. These tools allow developers to define specific event conditions, associate tasks with those conditions, and enable or disable trigger responses as needed. This flexible framework empowers the creation of robust and responsive real-time applications.

Figure 9.5 illustrates the basic trigger structure, featuring two lists: a waiting list that associates all events awaiting the trigger and a released list that associates events that have been triggered but whose tasks haven't fully executed; upon a trigger event, all tasks with events in the waiting list are moved to the released list, and as tasks respond to the trigger and complete execution, they are re-linked to the waiting list; once the last task on the released list finishes, the trigger is re-enabled for the next event. The trigger may be associated with an external event (hardware IRQ), or to internal events.

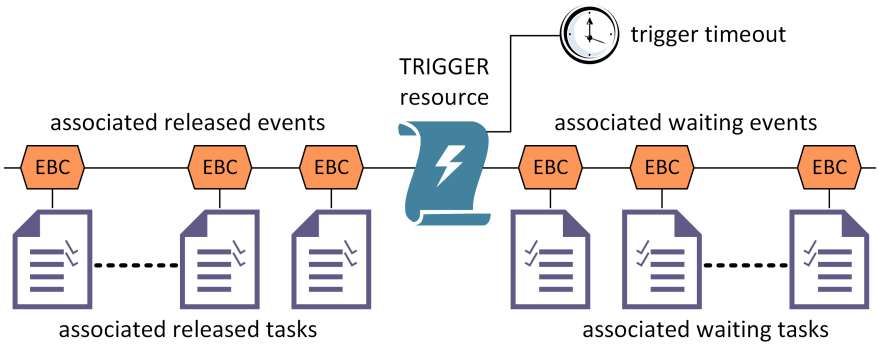

Figure 9.5: The Trigger Structure

There are four main functions to manage message queues:

- gu_TriggerCreate: a GEMRTOS resource is configured as trigger.
- gu_TriggerRegisterTask: a task is registered in a trigger to be invoked when trigger is released.
- gu_TriggerWait: a registered task changes to wait for the next released of the trigger.
- gu_TriggerRelease: the trigger is released.

The specific fields for a trigger structure is the following::

```
struct T_TRIGGER_RESOURCE {
  G_UINT32 TRG_IRQ_ID;         //Number of IRQ associated with
                               //(G_NUMBER_OF_IRQ_TCB + 1 if it is not
                               // associated with hardware interrupt).
  G_UINT32 TRG_Enabled;        // G_TRUE if trigger is enabled
  GS_TRGStatus TRGStatus;      // Status of the trigger
  GS_TRGTimeOutType TRGType;   // Type of the trigger (enum GS_TRGTimeOutType)
  G_UINT64 TRGTimeout;         // Trigger timeout in number of ticks
```

```
  GS_ECB *TRG_NextECB;         // Pointer to the timeout ECB
  void *(*enable_code)(void *); // Pointer to the function to be executed
                                // to re-enable trigger
  void *enable_arg;             // Pointer to the argument of the enable call
  void *(*disable_code)(void *);// Pointer to the function to be executed
                                // to reenable trigger
  void  *disable_arg; // Pointer to the argument of the disable call
};
```

The trigger structure stores all the information required to preserve the status of the resource.

Creating a Trigger A trigger assigns a RCB control block to hold all the trigger structure. The waiting ECB list of the RCB control block is used for waiting tasks and the granting ECB list is used for released tasks. A trigger is created with the following function:

```
  GS_RCB *prcb = gu_TriggerCreate(int irq_id, G_UINT64 ticks_to_wait)
```

The gu_TriggerCreate function returns the pointer to the RCB control block of the resource that implements the trigger. The irq_id argument specifies the hardware interrupt to with this trigger will be associated with. If the trigger is internal (not associated with external hardware interrupt) then it must be set equal to -1. The ticks_to_wait is the timeout for the trigger when it is waiting to be released. If ticks_to_wait is set equal to zero (0), the release of the trigger will be waited forever until it happens. There exist different kind of timeouts supported, and they are explained in section 9.4.3.

The process of creating a trigger, as shown in Figure 9.6, begins checking if the specified IRQ ID is within the valid range for hardware interrupts. If it is, the system verifies whether a trigger resource already exists for that IRQ. If not, it allocates a new resource control block (RCB), assigns the IRQ ID, and disables the associated hardware interrupt to prevent unintended activations. If an existing trigger resource is found, it confirms that the IRQ ID matches the current configuration. In cases where the IRQ ID indicates no hardware association, a new RCB is allocated with a special identifier, indicating it is not linked to a specific hardware interrupt.

Once the trigger resource is identified or created, its type is set to handle timeout events, and callback functions for enabling and disabling are initialized to null. The system then creates an event control block (ECB) to manage the trigger's timeout behavior and links it to the RCB. The ECB is initialized with a timeout value and associated with the RCB, with its type set accordingly. Depending on the specified time delay (ticks_to_wait), the trigger is configured either as disabled or set to restart when enabled. Finally, the trigger's status is marked as not valid, and the process exits the critical section, returning a pointer to the newly created trigger resource.

Registering in a Trigger Task wanted to be notified when a trigger takes place requires to be registered in advance. Registration is achieved by execution the function:

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

where ptcb is the handle of the task to be registered and ptrigger is the handle to the trigger to be registered to.

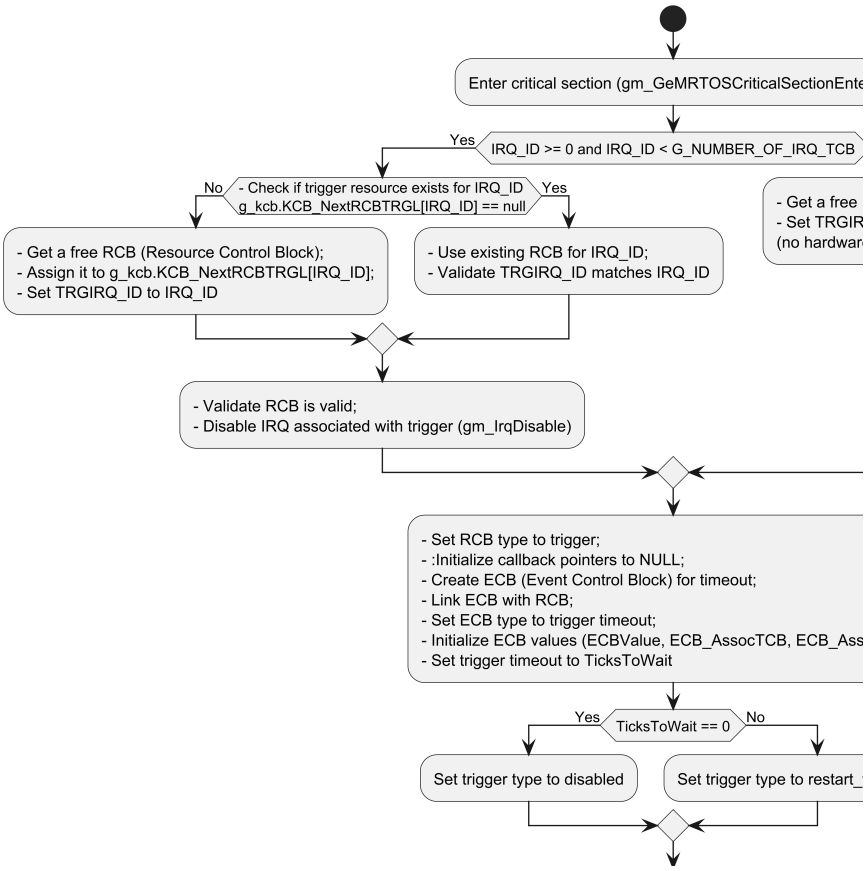

Figure 9.6: Trigger creation process

Registering tasks is essential to determine when the trigger should be enabled, as all registered tasks are waiting for the trigger to occur. The code 9.7 shows a sketch of the function.

The `gu_TriggerRegisterTask` function begins by entering in the critical section to ensure processor mutual exclusion during the registration process. The function sets the task's type to ISR and records the associated `IRQ` number. Next, it determines the trigger resource is valid. After establishing the trigger, the function allocates a new Event Control Block (ECB) for the task, initializing its values such as the association with the trigger and task, and setting its type based on the task's state. If the task is in a waiting state, the ECB is set to indicate that the task is waiting for a trigger event and link to the waiting linked list of the RCB control block; otherwise, it is set to a state indicating the task is executing the Trigger and the ECB control block is linked to the granting linked list of the trigger control block. Ultimately, the function exits the critical section and returns success.

Waiting for a Trigger Task should be waiting for the trigger to be released when the trigger occurs. The trigger is enabled when all the registered tasks are waiting for the trigger. A task waits for the trigger when executes the function:

```
GS_TRGStatus gu_TriggerWait(void);
```

When `gu_TriggerWait` function is executed, the task is suspended until the next released of the trigger or the occurrence of the trigger timeout. Consequently, the function may return:

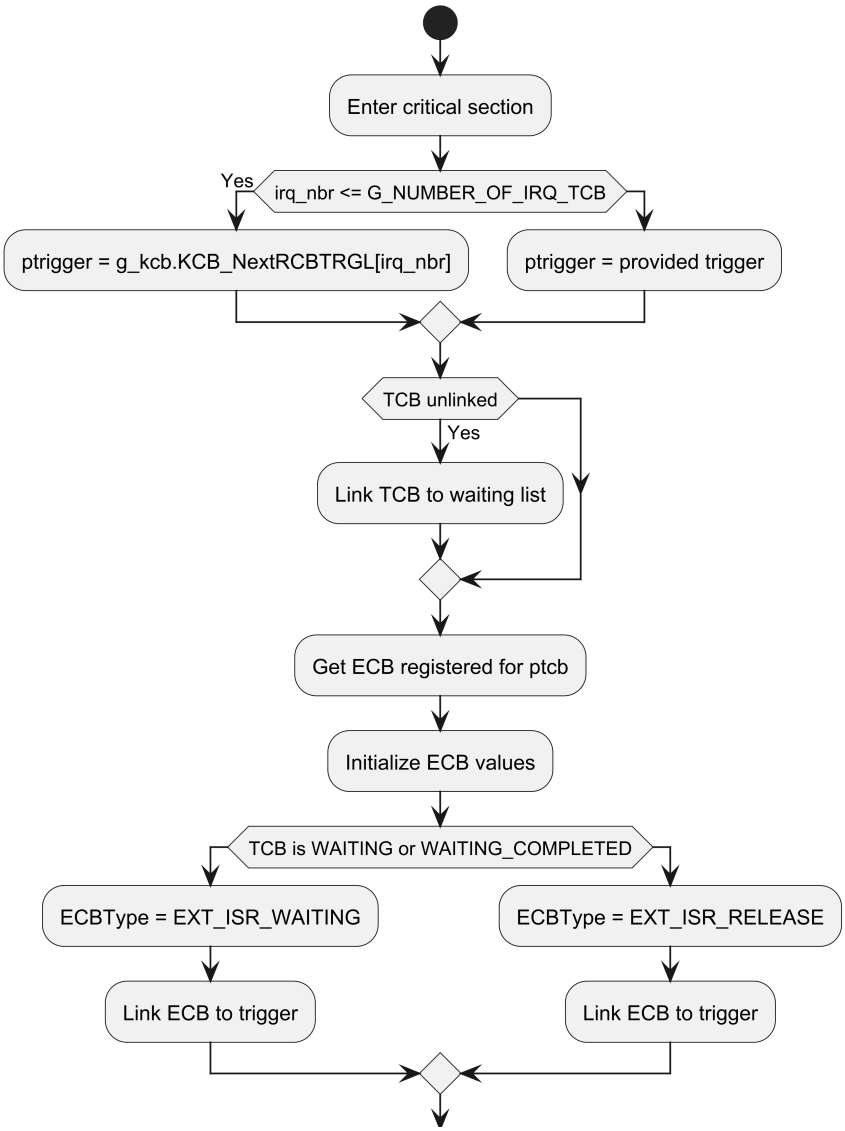

Figure 9.7: Task trigger registration process

- `G_TRGStatus_triggered`: the task is resumed because the trigger was released.
- `G_TRGStatus_timeout`: the task is resumed because the timeout elapsed without the trigger being released.

GeMRTOS trigger scheme offers a flexible mechanism for event management avoiding complex synchronization mechanism between event and timing management required in traditional operating systems. An event management with timing requirement is easily implemented as shown in code 9.5:

```
if (gu_TriggerWait()== G_TRGStatus_triggered) {
  ...
  // action to be performed when trigger is released
  ...
} else {  // returned status equal to G_TRGStatus_timeout
  ...
  // actions to be performed when timeout
  ...
```

```
}
```

Code 9.5: Event management with trigger timing requirement

Traditional operating systems require enabling the ISR routine for event management while a time interrupt should be set for the trigger timeout. The first occurrence, timeout or event, should manage the other interrupt in order to avoid race conditions. Synchronizations resources should be carefully configured to avoid these race conditions often difficult to debug. GeMRTOS offers a be easy and robust mechanism to manage events with timing requirements.

Releasing a Trigger A trigger is released when, upon being enabled, its associated event occurs. When released, all tasks waiting for that trigger are resumed, changing their state from waiting to ready. Triggers linked to hardware IRQs are released when an external interrupt occurs. Internal triggers, not connected to external IRQs, are released when the following function is executed:

```
G_UINT32 gu_TriggerRelease (ptrigger);
```

where `ptrigger` is the handle of the trigger to be released.

The `gu_TriggerRelease` function starts by entering a critical section to avoid race conditions when updating shared trigger data. It then obtains the trigger resource (RCB) linked to the provided handle and marks the trigger's status as "triggered," signifying that the associated event has occurred. Subsequently, it invokes an internal routine to manage the trigger release, which involves unblocking all tasks waiting on that trigger and transitioning their states from waiting to ready. This guarantees that all registered tasks are properly notified and can continue their operations.

Once the internal routine unblocks the tasks, the `gu_TriggerRelease` function exits the critical section. It then returns a success code, confirming that the trigger has been properly activated. This process ensures timely notification of multiple tasks waiting for the same event, enabling efficient synchronization and coordination within the real-time operating system.

Trigger Timeouts The creation of a trigger initializes an RCB control structure as a `GS_RCBType_TRIGGER` type for a trigger resource. If a timeout value is specified, the timeout is set as a `restart_when_enable` type, while it is disable if timeout argument is set equal to zero.

Triggers allow including timeouts while waiting for their release. This introduces a very efficient mechanism to handle events, internal or external, that requires a timed mechanism. For instance, is very frequent to wait for a message from an Ethernet card, but, at the same time, it is required to send messages in a while to keep the communication link alive. Pooling the receive buffer frequently is a waste of time and energy. It is more efficient to wait for the received interrupt that happens when a frame is received. Most used mechanism in traditional RTOS enables a semaphore to synchronize the interrupt routine with a timed event: two routines should be synchronized and an additional resource created to manage event with time constraints.

GeMRTOS introduces the trigger timeout. When the `gu_TriggerWait` is execute, the task is block until the first of two events takes place:

- The trigger is released because the event took place. Then the function returns the value `G_TRGStatus_triggered`.

- The timeout expired because no event took place. Then the function returns the value `G_TRGStatus_timeout`.

Code 9.5 shows how easily it is to define events with timeouts in GeMRTOS. Timeout is enabled when the trigger is created by defining the value of the timeout. The feature of the timeout is specified by the function:

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

where `ptrigger` is the handle to the trigger resource, and type defines the behavior of the timeout according to the following alternatives:

- `GS_TRGTimeOutType_disabled`: the timeout is disabled and consequently the `gu_TriggerWait` function returns only when the event associated with the trigger takes place.
- `GS_TRGTimeOutType_restart_when_enable`: the timeout is enabled each time that the `gu_TriggerWait` function is executed and the timeout interval is measured starting when the `gu_TriggerWait` function is executed (Figure 9.8).

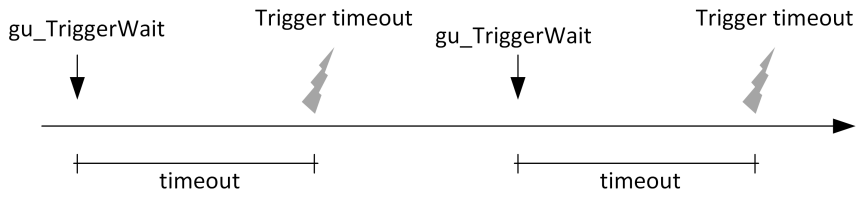

Figure 9.8: Trigger restart when enable timeout

- `GS_TRGTimeOutType_restart_when_timeout`: the timeout was reestablished when the previous timeout took place. In this way, the timeout takes place periodically, independently when the `gu_TriggerWait` function is executed (Figure 9.9).

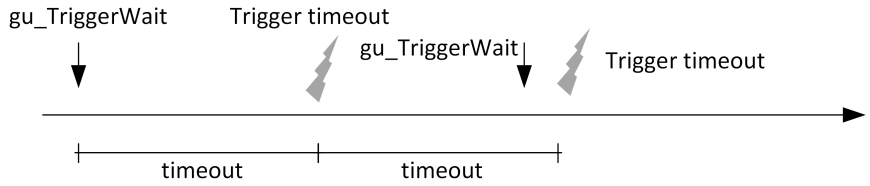

Figure 9.9: Trigger restarts when timeout.

- `GS_TRGTimeOutType_no_restart`: the timeout is not reestablished after it took place. In this way, it is enabled only for the next execution of the `gu_TriggerWait` only (Figure 9.10).

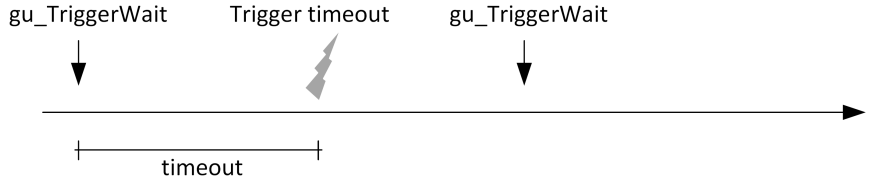

Figure 9.10: Trigger no-restart timeout.

All these types of alternatives allow flexibility and easiness suitable for different kinds of applications.

The GeMRTOS trigger mechanism provides a flexible and efficient way to manage event-driven operations within the system. By allowing tasks to register for specific trigger events and enabling external or internal stimuli to activate these triggers, the system can achieve precise synchronization and coordination. The functions for creating, enabling, disabling, and releasing triggers facilitate modular and dynamic management of system events, which is essential in complex, real-time environments. Additionally, triggers include time specifications through various timeout options, allowing for greater control over how and when tasks respond to events.

Overall, these trigger utilities enhance system responsiveness and resource utilization by decoupling event detection from task execution. They allow multiple tasks to wait for, or respond to, specific triggers with predefined timeout behaviors, ensuring that system actions are both timely and predictable. This approach significantly contributes to developing scalable and reliable systems, where event handling is streamlined and task interactions are well-controlled, leading to more robust and maintainable applications that can adapt to different timing requirements.

Trigger conclusions Triggers serve as essential mechanisms for event-driven programming, allowing systems to respond proactively to specific conditions or changes. They enable immediate and automatic execution of predefined actions without the need for continuous polling, thereby enhancing efficiency and responsiveness.

Moreover, triggers facilitate decoupling between event sources and handlers, promoting a modular and flexible architecture. This separation simplifies maintenance and scalability, as triggers can be modified or extended independently of the core system logic.

Finally, the effective implementation of triggers can significantly improve system responsiveness and resource utilization. When properly designed, they contribute to a more reactive and efficient system, capable of handling complex event patterns with minimal latency and overhead.