



GeMRTOS: Multiprocessor RTOS

User and Development Manual

gemrtos.com

September 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.

Contents

1	Introduction	1-1
1.1	Purpose of this Manual	1-1
1.2	Overview of GeMRTOS	1-1
1.2.1	Key Features of GeMRTOS	1-1
1.2.2	Benefits of Using GeMRTOS	1-2
1.2.3	Hybrid Partition Scheduling	1-2
1.3	Target Audience	1-2
2	Getting Started for Evaluators	2-1
2.1	Who This Chapter Is For	2-1
2.2	What You Need	2-1
2.3	The Example Catalogue	2-1
2.3.1	Listing the Examples	2-1
2.3.2	Building, Programming, and Running an Example	2-2
2.3.3	Reading the Output	2-2
2.4	The Example Catalogue at a Glance	2-2
2.5	Licensing for Evaluation	2-4
2.6	Where to Go Next	2-4
3	GeMRTOS Architecture	3-1
3.1	System Overview	3-1
3.2	GeMRTOS Controller	3-2
3.2.1	Mutual Exclusion Among Processors	3-2
3.2.2	Time Management	3-3
3.2.3	Interrupt Management	3-3
3.3	System Components and Control Blocks in GeMRTOS	3-3
3.3.1	Task	3-4
3.3.2	Events	3-5
3.3.3	Time	3-5
3.3.4	Resources	3-6
3.3.5	Signals	3-6
3.3.6	Processors	3-6
3.3.7	Hybrid Scheduling Lists	3-6
3.4	GeMRTOS Code Structure	3-7
3.4.1	Initialization Sections	3-9

3.4.2	Task Start Section	3-9
3.4.3	Interrupt Handler Section	3-10
3.4.4	System Call Section	3-10
3.4.5	Task Suspend Section	3-10
3.4.6	Task Complete Section	3-11
3.5	GeMRTOS Linked Lists	3-11
3.5.1	Hybrid Scheduling List Structure	3-11
3.5.2	Resource Management Lists	3-12
3.5.3	ECB Time List (ECBTL)	3-13
3.6	Conclusions	3-14
4	GeMRTOS Controller	4-1
4.1	Introduction to the GeMRTOS Controller	4-1
4.2	GeMRTOS Global Region	4-1
4.2.1	Reset Code for CPU_ID $\neq$ 1	4-4
4.2.2	Interrupt Return Code	4-5
4.2.3	Status Register (STATUS)	4-5
4.2.4	Control Register (CTRL)	4-6
4.2.5	PRC_IRQ_SECTION Register	4-8
4.2.6	SYS_TM_HGH Register	4-8
4.2.7	FRZ_THR Register	4-9
4.2.8	FRZ_TM Register	4-9
4.2.9	TM_CNT Register	4-10
4.2.10	NXT_OCC_TM Register	4-10
4.2.11	INOUTS Register	4-10
4.2.12	RST_CNT Register	4-11
4.2.13	SMP Register	4-11
4.2.14	MTX_GRN (Mutex Grant) Register	4-12
4.3	GeMRTOS Processor Region	4-12
4.3.1	ADDR_REG_0 Register	4-13
4.3.2	ADDR_REG_1 Register	4-14
4.3.3	ADDR_REG_2 Register	4-15
4.4	GeMRTOS Controller Mutex Architecture	4-16
4.4.1	Mutex Structure	4-16
4.4.2	Mutex Acquisition	4-16
4.4.3	Mutex Release	4-17
4.4.4	Mutex Release Decision Tree	4-17
4.4.5	R1_MTX_NESTED Synchronization Protocol	4-18
4.4.6	Nested Mutual Exclusion Terminology	4-20
4.5	GeMRTOS Watchdog Architecture	4-21
4.5.1	Timeout and Arming	4-22
4.5.2	The Halt/Run Pet Mechanism	4-22
4.5.3	Recovery on Reset	4-22
4.6	Time Management in the GeMRTOS Architecture	4-23
4.7	GeMRTOS Controller Device Interrupt Management	4-25
4.8	Conclusions	4-26

5	GeMRTOS: An Event-Driven RTOS	5-1
5.1	Introduction	5-1
5.2	Timed Events	5-2
5.2.1	Interrupt Handling	5-2
5.2.2	Event Processing	5-2
5.2.3	System Behavior	5-2
5.2.4	Example: Delay Event Processing	5-2
5.2.5	Flexibility and Extensibility	5-2
5.2.6	Frozen Mode	5-3
5.3	External Events	5-3
5.4	Internal Events in GeMRTOS	5-4
5.5	GeMRTOS Linked List Management	5-4
5.6	Callback Functions	5-5
5.6.1	gk_ResourceECBKillCallback function	5-5
5.6.2	gk_TimeCallback function	5-5
5.6.3	gk_FrozenCallback	5-5
5.6.4	gk_UnFrozenCallback	5-6
5.6.5	gk_TaskPrioritySetCallback	5-6
5.6.6	gk_StartTaskCallback	5-6
5.7	Conclusions	5-6
6	GeMRTOS IP for Platform Designer	6-1
6.1	Introduction	6-1
6.2	GeMRTOS IP Architecture	6-1
6.3	GeMRTOS IP Configuration Parameters	6-2
6.4	GeMRTOS IP Information	6-5
6.5	GeMRTOS IP Memory Map	6-6
6.5.1	Memory Map Organization	6-7
6.5.2	Conclusion	6-7
7	GeMRTOS Hardware and Software Design Flow	7-1
7.1	Introduction	7-1
7.2	Quartus Prime / Platform Designer Hardware Design Flow	7-2
7.3	The Software Design Flow	7-3
7.3.1	The settings.bsp file	7-4
7.4	gemrtos_build Script for Software Design Flow	7-5
7.5	Conclusions	7-6
8	Development Guide	8-1
8.1	Introduction	8-1
8.2	Licensing GeMRTOS	8-1
8.2.1	Hardware IP License Configuration	8-2
8.3	Hardware Design Guide	8-2
8.3.1	Integrating and Designing with the GeMRTOS Multiprocessor IP Component in Platform Design Projects	8-2
8.3.2	Step 3: Open Quartus Prime Project	8-3
8.3.3	Step 4: Open the Altera's Platform Designer Project	8-4
8.3.4	Step 6: Design the Altera's Platform Designer System	8-6

8.3.5	Step 7: Generate the Altera's Platform Designer System and Compile the Quartus Prime Project	8-7
8.4	Designing the GeMRTOS Multiprocessor Software Application	8-7
8.4.1	Alternative 1: Building the Application with the <code>gemrtos_build</code> Script	8-9
8.4.2	Beyond the Default Application: Building a Catalogue Example on Your Board	8-11
8.4.3	Alternative 2: Executing All the Steps to Create the Software Application	8-12
8.4.4	Alternative 3: Creating Your First GeMRTOS Project with the <code>gemrtos_build</code> Script	8-17
8.5	Conclusion	8-18
9	GeMRTOS Programming Guide	9-1
9.1	Introduction	9-1
9.2	Task Management	9-1
9.2.1	Periodic and Infinite-Loop (One-Shot) Tasks	9-2
9.2.2	Task Creation and Configuration	9-3
9.3	Recovering from a Hung Task: the Per-Processor Watchdog	9-6
9.3.1	Arming a Task's Own Watchdog	9-6
9.3.2	Registering a Recovery Handler	9-6
9.4	GeMRTOS Triggers: Event and Interrupt Management	9-7
9.5	GeMRTOS Resources	9-8
9.5.1	Semaphores and Mutexes	9-8
9.5.2	Message Queues	9-14
9.5.3	GeMRTOS Triggers: Event and Interrupt Management	9-18
10	Code Examples	10-1
10.1	Introduction	10-1
10.2	Example 1: Creating Infinite-Loop Tasks	10-1
10.3	Example 2: Creating Periodic Tasks	10-3
10.4	Example 3: Trigger-Driven JTAG-UART Management Tasks	10-5
10.4.1	The JTAG-UART Device	10-5
10.4.2	Using Trigger for Easy and Efficient JTAG UART Management	10-7
10.5	Example 4: Ethernet server with LwIP with TSE	10-23
10.5.1	TSE initialization for LwIP	10-24
11	GeMRTOS Example Applications	11-1
11.1	Introduction	11-1
11.2	01_hello_task — The Minimum Viable Application	11-2
11.3	02_tasks_and_delays — Periodic and One-Shot Tasks, Side by Side	11-3
11.4	03_semaphores — Every Synchronization Primitive GeMRTOS Offers	11-4
11.5	04_dining_philosophers — Contention and Deadlock Avoidance	11-4
11.6	05_message_queue_pubsub — Publish/Subscribe Messaging	11-5
11.7	06_triggers_and_isr — Hardware Triggers as a Real Interrupt Service Routine	11-6
11.8	07_multiprocessor_scheduling — Partitioned, Global, and Exclusive Scheduling	11-6
11.9	08_signals_and_frozen_mode — Fault Handling and Robustness	11-7
11.10	09_network_httpd — A Networked Status and Configuration Page	11-8
11.11	10_full_demo — The Complete Reference Application	11-8
11.12	11_watchdog_recovery — Per-Processor Watchdog Fault Recovery	11-9
11.13	Conclusion	11-9

12 Debugging GeMRTOS	12-1
12.1 General Recommendations	12-1
12.2 Debugging Multiprocessor Applications	12-2
12.3 Console Output Guarantees	12-2
12.3.1 Why Delivery Is Not Guaranteed	12-3
12.3.2 What Is Discarded, and When	12-3
12.3.3 Steady-State Output Versus Bursts	12-3
12.3.4 Detecting Lost Output	12-3
12.4 System Console	12-4
12.5 HPS Compatibility	12-4
12.6 Common Failure Modes	12-4
12.7 Production Build Configuration	12-5
12.7.1 Two different “production” configurations exist today	12-6
13 API Reference	13-1
13.1 API Naming Conventions	13-1
13.2 Critical Section Macros - Detailed Semantics	13-1
13.2.1 Critical Section Enter Macro Semantics	13-1
13.2.2 Critical Section Exit Macro Semantics	13-2
13.2.3 Mutex Nested Value Synchronization	13-3
13.3 GeMRTOS API Functions categories	13-4
13.3.1 Critical Section category	13-4
13.3.2 Frozen Mode category	13-5
13.3.3 IRQ Management category	13-5
13.3.4 Message Queue category	13-5
13.3.5 Processor category	13-6
13.3.6 Scheduling List category	13-6
13.3.7 Semaphore and Mutex category	13-6
13.3.8 Signal category	13-7
13.3.9 System category	13-7
13.3.10 Task category	13-7
13.3.11 Time Management category	13-8
13.3.12 Trigger category	13-8
13.4 GeMRTOS Functions	13-9
13.4.1 gm_GeMRTOSCriticalSectionEnter	13-9
13.4.2 gm_GeMRTOSCriticalSectionExit	13-9
13.4.3 gm_FrozenModeDisable	13-10
13.4.4 gm_FrozenModeEnable	13-11
13.4.5 gm_GatedClockDisable	13-12
13.4.6 gm_GatedClockEnable	13-12
13.4.7 gm_StatusFrozenModeEnableGet	13-13
13.4.8 gm_SystemReset	13-14
13.4.9 gm_TimeCountersReset	13-15
13.4.10 gm_TimeCountersHold	13-16
13.4.11 gm_TimeCountersUnhold	13-16
13.4.12 gm_TimeCoutersUnreset	13-17
13.4.13 gu_FrozenModeThresholdGet	13-18

13.4.14	gu_FrozenModeThresholdSet	13-19
13.4.15	gu_FrozenModeTimeGet	13-20
13.4.16	gm_IrqDisable	13-21
13.4.17	gm_IrqEnable	13-21
13.4.18	gk_NextOccTimeProcessor	13-22
13.4.19	gm_TriggerPrcIRQ	13-22
13.4.20	gu_MessageQueueCreate	13-23
13.4.21	gu_MessageQueuePrintf	13-24
13.4.22	gu_MessageQueueReceive	13-24
13.4.23	gu_MessageQueueSend	13-25
13.4.24	gu_MessageQueueSubscribe	13-26
13.4.25	gm_PrcInterruptDisable	13-27
13.4.26	gm_PrcInterruptEnable	13-27
13.4.27	gm_ProcessorHalt	13-28
13.4.28	gm_ProcessorId	13-29
13.4.29	gm_ProcessorInterrupt	13-29
13.4.30	gu_ProcessorWaitForIrq	13-30
13.4.31	gu_SchedulingListAssociateProcessor	13-31
13.4.32	gu_SchedulingListAssociateTask	13-31
13.4.33	gu_SchedulingListCreate	13-32
13.4.34	gu_SchedulingListExclusionSet	13-33
13.4.35	gu_SemaphoreCreateBinary	13-33
13.4.36	gu_SemaphoreCreateMutex	13-34
13.4.37	gu_SemaphoreCreateRecursiveMutex	13-35
13.4.38	gu_SemaphoreGetCount	13-35
13.4.39	gu_SemaphoreGive	13-36
13.4.40	gu_SemaphoreTake	13-37
13.4.41	gu_SignalCreate	13-37
13.4.42	gu_SignalDestroy	13-38
13.4.43	gm_EREAddressSet	13-39
13.4.44	gm_MutexNestedValue	13-39
13.4.45	gm_ReadInputs	13-40
13.4.46	gm_StatusFrozenModeActiveGet	13-41
13.4.47	gm_WriteOutputs	13-41
13.4.48	gu_fprintf	13-42
13.4.49	gu_printf	13-43
13.4.50	gu_TaskCreate	13-44
13.4.51	gu_TaskDelay	13-45
13.4.52	gu_TaskDelayTime	13-45
13.4.53	gu_TaskGetCurrentTCB	13-46
13.4.54	gu_TaskKill	13-47
13.4.55	gu_TaskPeriodSet	13-48
13.4.56	gu_TaskReadyPrioritySet	13-48
13.4.57	gu_TaskResume	13-49
13.4.58	gu_TaskRunPrioritySet	13-50
13.4.59	gu_TaskStartWithOffset	13-51
13.4.60	gu_TaskSuspend	13-52

13.4.61	gu_TaskTypeSet	13-52
13.4.62	gm_StatusCountingTimeGet	13-53
13.4.63	gm_StatusResetCountingTimeGet	13-54
13.4.64	gm_SystemTimePrescaleGet	13-54
13.4.65	gm_SystemTimePrescaleSet	13-55
13.4.66	gu_SystemTotalTimeGet	13-56
13.4.67	gu_TriggerCreate	13-56
13.4.68	gu_TriggerDisable	13-57
13.4.69	gu_TriggerDisableHook	13-58
13.4.70	gu_TriggerEnable	13-58
13.4.71	gu_TriggerEnableHook	13-59
13.4.72	gu_TriggerRegisterTask	13-60
13.4.73	gu_TriggerRelease	13-61
13.4.74	gu_TriggerSetTimeoutType	13-61
13.4.75	gu_TriggerWait	13-62
13.4.76	gu_ConvertTime	13-63
13.4.77	gu_SemaphoreCreateCounting	13-63
13.4.78	gu_SemaphoreDestroy	13-63
13.4.79	gu_MessageQueueDestroy	13-64
13.4.80	gu_TriggerDestroy	13-64
13.4.81	gm_WatchdogTimeoutSet	13-64
13.4.82	gm_PrcWatchdogEnable	13-65
13.4.83	gm_PrcWatchdogDisable	13-66
13.4.84	gu_WatchdogHandlerSet	13-66
14	Windows Emulation Layer	14-1
14.1	Overview	14-1
14.2	Known Limitations	14-1
14.2.1	ERET Instruction Behavior	14-1
14.2.2	Interrupt Timing	14-2
14.2.3	Clock Accuracy	14-2
14.2.4	Processor Reset	14-3
14.3	Installing the Required Tools	14-3
14.3.1	Installing Visual Studio Code	14-3
14.3.2	Installing the MinGW GCC Toolchain	14-3
14.4	Windows Build and Testing	14-4
14.4.1	Package Composition	14-4
14.4.2	First-Run Setup	14-4
14.4.3	Opening and Configuring the VS Code Workspace	14-4
14.4.4	Build Commands	14-5
14.4.5	Run Commands	14-5
14.4.6	Debugging	14-5
	Glossary	1

Introduction

1.1 Purpose of this Manual

As a detailed technical reference, this manual provides developers with the essential knowledge required to design, implement, and manage real-time embedded systems on Altera Field-Programmable Gate Array (FPGA) devices, all leveraging the power of Generic eMbedded Real-Time Operating System (GeMRTOS). This includes understanding GeMRTOS's core capabilities, such as task management, inter-process communication, and interrupt handling. It also details how to seamlessly integrate GeMRTOS with Altera FPGA hardware and development tools, enabling the creation of sophisticated and efficient embedded solutions.

Note: To facilitate understanding, a glossary of technical terms specific to GeMRTOS is included at the end of this manual. Readers less familiar with these terms are encouraged to consult the glossary as needed.

1.2 Overview of GeMRTOS

GeMRTOS is a powerful yet user-friendly Real-Time Operating System (RTOS) tailored for demanding multiprocessor embedded systems on Altera FPGAs. It enables the creation of complex, real-time applications through simplified processor management, synchronization, and resource allocation. Its features include a timer-tickless scheduler for minimal overhead, floating scheduling to prioritize critical tasks, and hybrid partitioning for adaptable task allocation. Using GeMRTOS, developers can achieve enhanced performance, reduced development time, and a modular, maintainable codebase.

1.2.1 Key Features of GeMRTOS

GeMRTOS boasts the following key features:

- Modular design for straightforward integration within Altera's Platform Designer.
- Broad processor support, compatible with both Nios II and Nios V soft processors.
- Timer-tickless scheduler, minimizing overhead and maximizing responsiveness.
- Floating scheduling, ensuring efficient management of high-priority tasks.
- Support for event-driven programming, enabling reactive system designs.

- Hybrid partition scheduling, for adaptable task management.

1.2.2 Benefits of Using GeMRTOS

Adopting GeMRTOS offers several significant benefits:

- Simplified multiprocessor system design, streamlining development efforts.
- Enhanced performance for embedded applications, achieving highly efficient resource utilization.
- Reduced development time, accelerating project completion.
- Modular and maintainable codebase, promoting long-term system stability and adaptability.

1.2.3 Hybrid Partition Scheduling

GeMRTOS introduces hybrid partition scheduling, an innovative approach that overcomes the limitations of traditional global and partitioned scheduling. Offering unparalleled efficiency and flexibility, this method uses scheduling lists to dynamically assign tasks and processors, adapting to the evolving needs of application subsystems. This allows dedicating processors to specific roles, such as sensor data collection or user interface management, maximizing system responsiveness and optimizing resource utilization.

1.3 Target Audience

This manual is tailored for embedded systems engineers, software developers, and hardware designers working with GeMRTOS on Altera FPGA platforms.

Getting Started for Evaluators

2.1 Who This Chapter Is For

This chapter is for someone with a **pre-built** GeMRTOS board project in hand — GeMRTOS already integrated into a Quartus® Prime / Platform Designer system, ready to compile — who wants to see it running in minutes, not build that integration from scratch. If you are integrating GeMRTOS into your *own* new hardware design instead, see Chapter 8 (Development Guide).

Note: This chapter uses `m10_rgmii_project_msgdma` as its reference board project throughout. The same flow applies to any board project GeMRTOS has already been integrated into — only the board directory name changes.

2.2 What You Need

- Quartus Prime (Standard or Pro, matching your board project) with the Nios V or Nios II toolchain installed.
- The board itself, connected via USB-Blaster/JTAG.
- A plain command shell (`cmd.exe`) — no special environment setup is required beyond what Quartus Prime installs.
- For `09_network_httpd` specifically: an Ethernet cable into a DHCP-capable network. Every other example needs only the JTAG connection above.

2.3 The Example Catalogue

GeMRTOS ships eleven runnable example applications, each demonstrating one coherent slice of the API — from a one-task smoke test through a full multiprocessor, networked demo. They live in `ip/GeMRTOS_Multiprocessor/examples/` and are selected at build time, without editing any source file yourself.

2.3.1 Listing the Examples

From the repository root:

```
new_script.bat -ex-list --board m10_rgmii_project_msgdma
```

This prints every example's name, the application directory it deploys into, and a one-line description, then exits — nothing is built.

2.3.2 Building, Programming, and Running an Example

One command builds the selected example, programs the FPGA, downloads the compiled application, and opens a JTAG UART terminal window per processor:

```
new_script.bat -std -ex 01_hello_task --board m10_rgmii_project_msgdma ^
-qpf top -qsys q_sys -dir software -sd e -fat f -debug
```

- `-std` selects Quartus Prime Standard (use `-pro` for Quartus Prime Pro).
- `-ex <name>` selects the example — replace `01_hello_task` with any name from the catalogue listing.
- `-qpf/-qsys` name the Quartus/Qsys project files (without extension) already present in the board directory — these do not change per example, only per board.
- `--board` is the board directory at the repository root.

This is the same script used to build every example throughout this manual's own development and verification history — nothing about it is example-specific machinery you need to understand, only the `-ex` value changes.

Note: Omitting `-ex` entirely builds `10_full_demo` — the original, everything-combined demo this catalogue was extracted from. Existing workflows that predate the example catalogue keep working completely unchanged.

Note: Programming and downloading here are the same `quartus_pgm/niosv-download` (or `nios2-download`) steps described in detail in Section 8.4.3 and Section 8.4.3 — this script performs them for you; nothing new to learn if you have already read the Development Guide chapter.

2.3.3 Reading the Output

Each JTAG UART terminal window corresponds to one processor instance (`juart-terminal/nios2-terminal --instance=N`). Every example's own `README.md` (in its directory under `ip/GeMRTOS_Multiprocessor/examples/`) documents exactly what to expect on the UART, which lines are normal and which would indicate a problem, and a short "Things to try" list for exploring further — this manual does not duplicate that content here, since it is authored and kept current alongside each example's own source.

2.4 The Example Catalogue at a Glance

Figure 2.1 groups the same eleven examples by theme, as a quick starting point; Table 2.1 below gives the full detail for each.

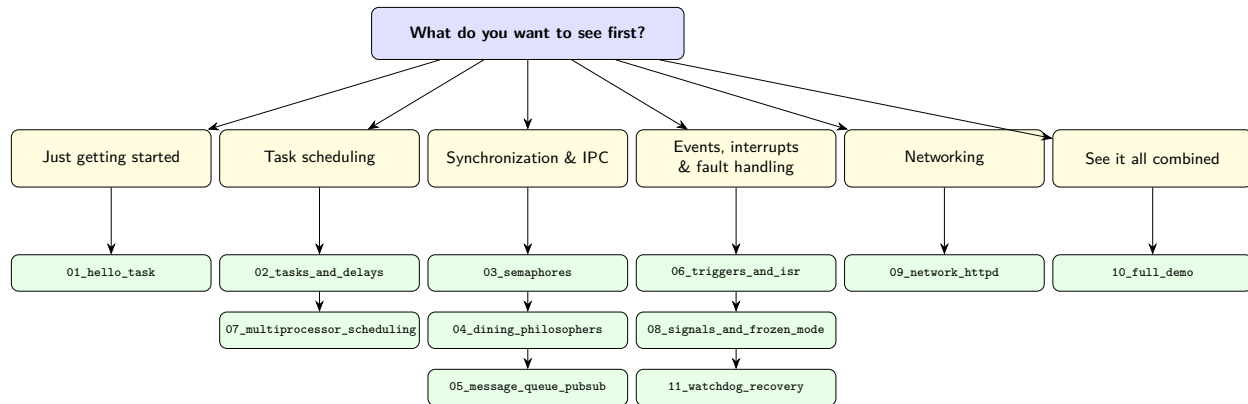


Figure 2.1: Which example do I need?

Table 2.1: The example catalogue.

Name	Demonstrates	Extra hardware
01_hello_task	Minimum viable application; board smoke test	—
02_tasks_and_delays	Periodic vs. one-shot tasks, ready/run priority, start offsets	—
03_semaphores	All four semaphore flavours, acquisition timeouts, priority ceiling	—
04_dining_philosophers	Classic contention/deadlock-avoidance stress demo	LEDs
05_message_queue_pubsub	Publish/subscribe message queues, fan-out, late subscribers	—
06_triggers_and_isr	Hardware triggers as a real interrupt service routine	—
07_multiprocessor_scheduling	Partitioned vs. global scheduling, CPU affinity, mutual exclusion between scheduling lists	2+ processors
08_signals_and_frozen_mode	Self-recovering fault handling: deadline overrun, timed-event backlog	—
09_network_httpd	LwIP TSE + HTTP server, live status page, editable parameters	Ethernet/DHCP
10_full_demo	Everything combined — the long-standing regression reference	Ethernet/DHCP, LEDs
11_watchdog_recovery	Per-processor watchdog fault recovery: a hung task is caught and its processor reset and recovered automatically, while the rest of the system keeps running	2+ processors

Every example needs a JTAG connection for its UART output; the "Extra hardware" column above lists only what it needs *beyond* that.

2.5 Licensing for Evaluation

The example catalogue is distributed under the PolyForm Noncommercial License 1.0.0 (see the LICENSE file at the repository root). Building, running, and exploring these examples for evaluation purposes is free under that license; a separate commercial license from GeMRTOS is required before shipping a product built on GeMRTOS.

2.6 Where to Go Next

- **Integrating GeMRTOS into your own board design:** Chapter 8, Development Guide.
- **Understanding how GeMRTOS is put together:** Chapter 3, GeMRTOS Architecture.
- **Common code patterns for your own application:** Chapter 10, Code Examples.
- **The full task/synchronization/IPC API:** Chapter 9, GeMRTOS Programming Guide.

GeMRTOS Architecture

3.1 System Overview

GeMRTOS is designed to facilitate real-time multitasking on multiprocessor systems. A key aspect is the management of hardware resources and the software architecture. Figure 3.1 shows a layout of the GeMRTOS architecture that comprises hardware and software. This innovative architecture adapts to evolving system requirements, ensuring that complex, real-time applications can be created through simplified processor management, synchronization, and resource allocation. Leveraging an event-driven design and hybrid partition scheduling, GeMRTOS empowers developers to build responsive and scalable applications for demanding environments, all while maintaining a clear separation between hardware and software components.

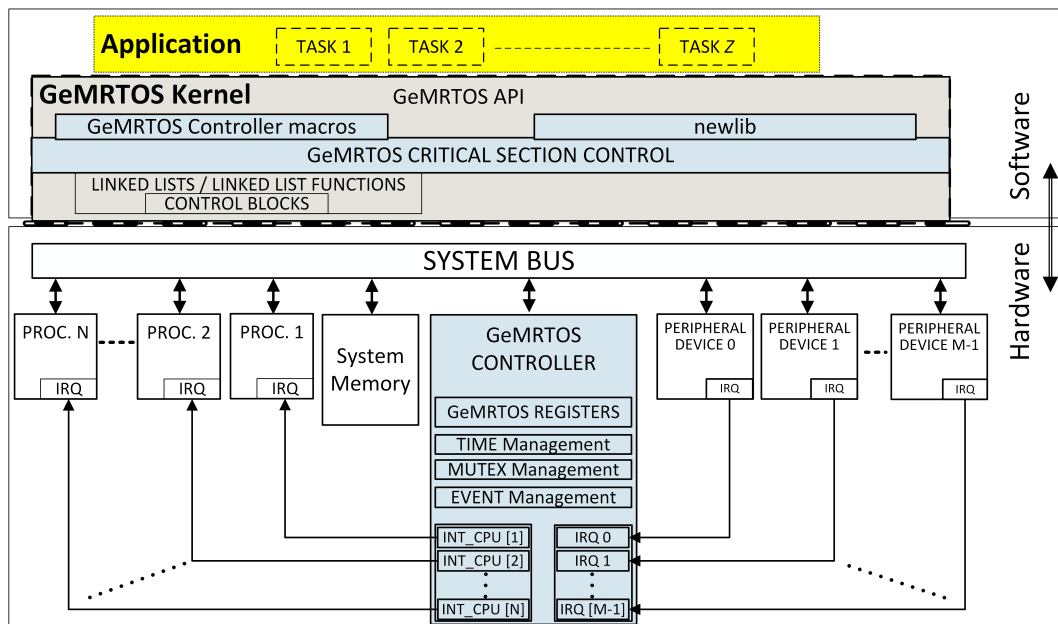


Figure 3.1: GeMRTOS architecture layout.

The hardware architecture of a GeMRTOS-based system includes the following core interconnected elements:

- **Processors:** Multiple processors (PROC 1, PROC 2, ..., PROC N) with unique CPU_IDS. These processors are the primary execution units where tasks run. Each processor has a unique CPUID (Central Processing

Unit Identifier) for identification on the bus and for use within the GeMRTOS itself. Current GeMRTOS version supports both the Nios II and the Nios V soft-processors for the Altera FPGA families.

- **System Bus:** This component is essential to allow communication between the multiple elements. Altera Nios II and Nios V processors utilize the Avalon bus.
- **Central Memory:** Accessible through the System Bus. This is the main memory used by all processors for code, data, and stacks. The size is dictated by the embedded application and specific hardware implementation.
- **GeMRTOS Controller:** Manages all the processor synchronization, time management, and Centralized Interrupt Management.
- **I/O Devices with Interrupt Requests (IRQs):** Connected to the Interrupt lines of the GeMRTOS controller. This provides a centralized, scalable and managed way to handle peripherals.

The software architecture of GeMRTOS relies on Control Blocks, linked to maintain system state, that store information about tasks, events, signals, resources, times, and processors. To ensure consistency, only functions holding the GeMRTOS Mutex in the GeMRTOS controller can modify these Control Blocks and their relationships. User applications are built from tasks, implemented as either periodically released functions or infinite-loops that suspend themselves to achieve regularly timed execution.

GeMRTOS maintains the system state using its controller registers and highly linked lists of control blocks. System state transitions occur in response to events. The GeMRTOS controller manages external events originating from peripherals via interrupts and maintains system time for generating timed events. The GeMRTOS controller then interrupts the most appropriate processor to service the event, updating system state accordingly. For instance, when a timed event occurs, the processor executing the task associated with the timed event, if any, is interrupted; otherwise, if the associated task is not executing, the processor executing the lowest priority task is interrupted.

In GeMRTOS, system processors are dedicated to executing application tasks and updating system state in response to events. System functions are executed only when system state needs to be updated, avoiding unnecessary periodic execution. The GeMRTOS controller and Control Blocks are essential for preserving system state and managing system events, a design that makes GeMRTOS efficient and flexible for real-time applications.

Before delving into the implementation details, it's essential to grasp the core concepts upon which GeMRTOS is built. This section provides an overview of these foundational elements.

3.2 GeMRTOS Controller

The GeMRTOS controller's primary function is to provide innovative and adaptable management of time and external events, as well as ensuring mutual exclusion among system processors to guarantee coherent access to the system state. This is achieved through a combination of hardware and software mechanisms, allowing for efficient and reliable coordination in a multiprocessor environment. The controller plays a critical role in maintaining system-wide consistency and enabling predictable real-time behavior.

3.2.1 Mutual Exclusion Among Processors

To preserve system state consistency, the GeMRTOS controller employs a hardware Mutex resource, an innovative approach that surpasses traditional locking mechanisms in efficiency. As a critical section, the system state requires that only one procedure be executed at a time, preventing conflicting modifications. Consequently, only one processor is permitted to modify the system state at any given time. The GeMRTOS controller implements this hardware Mutex resource, allowing only one processor at a time to execute system update procedures.

3.2.2 Time Management

Time management is crucial in real-time systems, dictating when events occur. The GeMRTOS controller leverages two primary registers to manage time with exceptional precision and adaptability. First, the System Time register increments based on the system clock and Prescale register configuration, typically set to count in 10MHz increments. This register tracks time in non-frozen mode. Second, the Frozen Mode Time register increments when the system switches to frozen mode, which is activated when the frozen mode is enabled and the next timed event exceeds the System Time by a threshold predefined by the user.

3.2.3 Interrupt Management

GeMRTOS employs an adaptable interrupt management architecture that allows the system to respond effectively to external events. In most cases, an application is implemented as a set of tasks executed by system processors, triggered by external events. The system includes both processors and peripherals; the processors interact with the external environment via these peripherals to receive stimulus. Interrupts are usually implemented to force a processor to execute the task that services the peripheral. GeMRTOS handles interrupt requests classified as:

- **Timed event interrupts:** These events are configured to happen at a certain system time. Time events are sorted according to the occurrence time in a total order relationship. The next time event is the earliest time event.
- **External interrupts:** These events are produced by the input/output peripheral devices. Most of these events require the execution of the associated interrupt service routine (ISR) in order to satisfy the demand of the input/output peripheral. Interrupt requests may be enabled or disabled by setting or clearing, respectively, each interrupt request in the GeMRTOS controller. Therefore, an interrupt request from the peripherals will produce a processor interrupt request when the interrupt request is enabled.
- **Internal interrupts:** A processor may interrupt another processor. However, this method should be limited only to internal GeMRTOS kernel procedures.

This innovative approach enables the GeMRTOS controller to effectively manage interrupt requests from timed events, system peripherals, and interval interrupts. By selecting the most adequate processor to execute the service task(s) associated with the interrupt, GeMRTOS optimizes system performance and responsiveness.

Furthermore, GeMRTOS implements triggers, a sophisticated mechanism that expands upon traditional interrupt handling. Triggers are associated with both internal and external events, releasing one or several tasks in response. This adaptable design allows an event to trigger the execution of multiple associated tasks, while also supporting timeout settings to handle events that do not occur within a predefined time interval.

3.3 System Components and Control Blocks in GeMRTOS

GeMRTOS achieves its real-time capabilities and adaptable behavior through a set of carefully designed system components and control blocks. In contrast with traditional methods, GeMRTOS generalizes all the functions and system components into a set of structures whose information is held into control block highly linked among them. GeMRTOS relies on control blocks to maintain the system's state. Key control blocks include:

- **Kernel Control Block (KCB):** Centralizes operating system information.
- **Task Control Block (TCB):** Stores runtime information for an application task.
- **Event Control Block (ECB):** Manages event-related information.
- **Scheduling List Control Block (LCB):** Supports hybrid partition scheduling.
- **Processor Control Block (PCB):** Tracks the status of each system processor.
- **Resource Control Block (RCB):** Manages system resources like semaphores and message queues.
- **Signal Control Block (SCB):** Defines internal signals.

This innovative design, comprised of carefully linked control blocks, maintains the system's state throughout runtime. Only the processor that currently holds the GeMRTOS Mutex is authorized to modify the information held within these control blocks, ensuring system consistency and data integrity. Each control block is associated with a key entity within GeMRTOS, contributing to the system's modularity and adaptability.

3.3.1 Task

In GeMRTOS, a task is defined as a fundamental unit of execution, designed for flexible and adaptable management within a multiprocessor environment. As illustrated in Figure 3.2, each task consists of the following elements:

- **Task Control Block (TCB):** A control block storing essential runtime information for task management, enabling dynamic scheduling and resource allocation.
- **Task Code:** The executable code loaded into system memory, implementing the task's function when executed by a system processor.
- **Task Stack:** Data storage for preserving the task's status when suspended, enabling seamless resumption and efficient resource management.

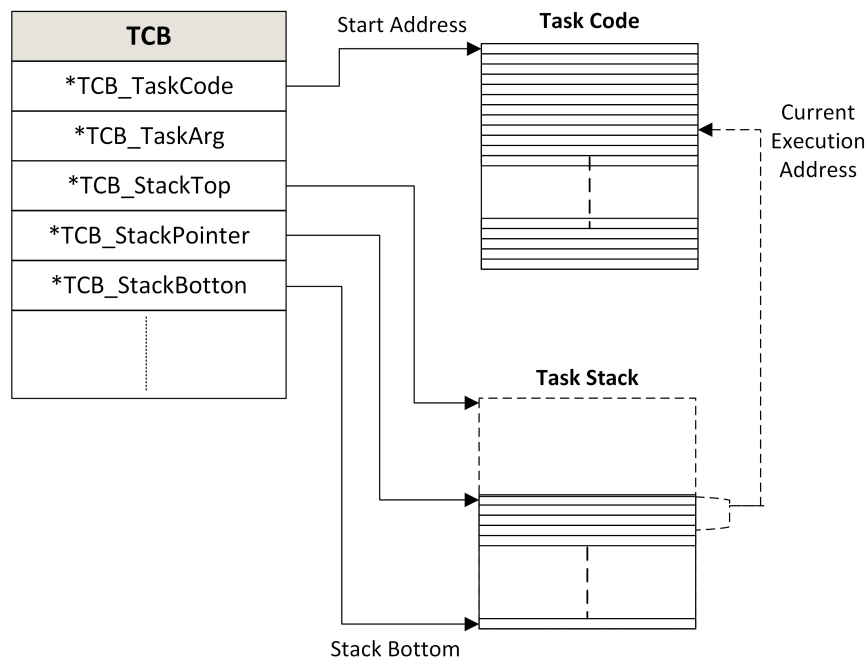


Figure 3.2: GeMRTOS task main components.

GeMRTOS tasks can exist in different states, each reflecting a particular stage in their lifecycle:

- **Waiting State:** The task is awaiting the release of an event, enabling efficient synchronization and event-driven behavior. A task in this state is referred to as a "waiting task."
- **Ready State:** The task is prepared for execution, awaiting a processor to grant access. A task in this state is known as a "ready task," enabling adaptable scheduling based on system demands.
- **Running State:** The task is actively executing on a processor. Task can be executed to different states: suspended, finishes the execution and it is preempted by a higher priority task. A task in this state is referred to as an "executing task."

Tasks transitions states according to the events that change states. To support adaptable system behavior, GeMRTOS defines a set of event categories that trigger task state changes:

- **Create Events:** Create a task and set its state to waiting, initializing its stack for execution.

- **Release Events:** Change the task state from waiting to ready, enabling processor allocation based on task priority.
- **Suspend Events:** Change the task state from ready to waiting, preserving its context for later resumption.
- **Blocking Events:** Change the task state from running to waiting, typically due to resource contention or synchronization requirements.
- **Execution Events:** Change the task state from ready to running, allowing a processor to execute the task code.
- **Deallocate Events:** Change the state of a task from the running state to the ready state.
- **Finish Events:** Destroy a task. A finish event destroys a task-independent of its state.

Tasks may be classified as:

- **User Task:** Tasks released for execution according to user-defined configurations.
- **Idle Task:** Tasks executed when no other user tasks are ready.

This innovative task management framework, leveraging flexible state transitions and event-driven mechanisms, enables GeMRTOS to efficiently orchestrate complex applications within a multiprocessor environment. By dynamically adapting to changing system conditions and prioritizing critical tasks, GeMRTOS ensures responsive and reliable performance in demanding real-time scenarios.

3.3.2 Events

GeMRTOS leverages an event-driven architecture, enabling adaptable and efficient responses to both internal and external stimuli. Events are the cornerstone of system activity, triggering task state transitions and coordinating resource allocation. By responding to events rather than relying on constant polling, GeMRTOS minimizes overhead and maximizes system responsiveness. GeMRTOS supports three primary categories of events:

- **Timed Events:** Occur when the system time reaches a specific, pre-defined value. These events enable precise scheduling and control of time-critical operations, offering a predictable and reliable mechanism for tasks that must execute at specific intervals.
- **External Events:** Generated by peripheral devices, signaling asynchronous interactions with the external environment. These events enable the system to react promptly to changes in its surroundings, facilitating responsive control and monitoring of external processes.
- **Internal Events:** Occur based on runtime conditions within the system, including synchronizations and signals. These events allow for adaptable responses to unforeseen circumstances and dynamic system adjustments, ensuring robust and reliable operation even under unexpected conditions.

Through this comprehensive event management framework, GeMRTOS provides a highly adaptable and responsive platform for building complex, real-time embedded systems. By carefully defining and managing event triggers, developers can create systems that react efficiently to changing conditions, optimize resource utilization, and maintain reliable performance in demanding environments.

3.3.3 Time

Time is a fundamental element in GeMRTOS, providing the crucial foundation for time-aware decision making and efficient task scheduling. In an innovative approach to time management, the GeMRTOS controller utilizes two key registers: the `R_TM_CNT` register, which holds the system time in non-frozen mode, and the `R_FRZ_TM_CNT` register, which holds the length of time the system stayed in frozen mode.

This dual-register system enables adaptable time management in the face of varying system loads. The GeMRTOS controller intelligently produces a time event each time the `R_TM_CNT` register reaches the value stored in the `R_NXT_EVN_CNT` register. Upon reaching this value, GeMRTOS interrupts either the processor executing the task associated with the timed event (if it is executing) or the processor executing the lowest priority task (if no task

is associated with it). This adaptable approach ensures that timed events are efficiently processed by choosing the most suitable processor to interrupt, balancing responsiveness with overall system load.

Through this innovative time management system, GeMRTOS provides a robust and adaptable foundation for building real-time applications. By precisely tracking system time and intelligently scheduling time events, GeMRTOS empowers developers to create systems that meet strict timing deadlines and maintain reliable performance under dynamic conditions.

3.3.4 Resources

In a multitasking system, tasks often share access to system resources, such as data memory, code memory, I/O devices, and specialized hardware components. To manage the utilization of these resources efficiently and fairly among tasks, GeMRTOS provides an innovative resource management mechanism that enables adaptable resource sharing and prevents resource contention. This mechanism offers data structures and functions designed to handle the sharing of system resources among the tasks that require their utilization.

GeMRTOS implements the Resource Control Block (RCB) to manage the events associated with resources, supporting a variety of resource management strategies. The Resource Control Block provides a flexible approach to assigning resources. In fact, when a task requests a resource, one of two approaches can be applied (1) the resource may be granted to the task if it is available or (2) the task may be blocked if it must wait until the resource becomes available. Moreover, a timeout may need to be specified in order to avoid a task either granting or waiting for a resource for a long time, which would impede the adequate sharing of resources among tasks.

3.3.5 Signals

In operating systems, signals provide a powerful mechanism for handling exceptional conditions that may arise during runtime, offering an adaptable and efficient way to respond to unforeseen events. GeMRTOS leverages signals to enable tasks to execute associated code when a specific condition occurs, facilitating flexible error handling and dynamic system adjustments. These conditions may be predefined, such as task creation, resource allocation, or task blocking. The Signal Control Block (SCB) control structure is defined to store signal information, and is linked to control block structures accordingly. The SCB control structure points to the code to be executed when the condition of the signal occurs. This code is executed prior to the execution of the code of the task with which the signal is associated. This innovative approach enhances system robustness and adaptability by providing a structured way to handle exceptional circumstances.

3.3.6 Processors

The system processors are the workhorses of GeMRTOS, executing the code of the operating system, tasks, and signals. The operating system is in charge of sharing the utilization of the processors among the system tasks. This adaptable processor allocation is achieved through an innovative approach that ensures efficient resource utilization and responsiveness. Each processor is assigned to execute the highest priority tasks in the ready state. The Processor Control Block (PCB) stores the state of each processor, enabling the system to dynamically track processor availability and workload. Each processor may serve several scheduling lists, and several processors may serve the same scheduling list, allowing for a highly flexible and scalable task distribution.

3.3.7 Hybrid Scheduling Lists

GeMRTOS introduces an innovative hybrid scheduling approach, offering exceptional configuration flexibility for diverse application requirements. Unlike traditional scheduling methods, GeMRTOS allows application

configuration using hybrid scheduling lists, enabling dynamic adaptation to changing system conditions. Each task is assigned to a hybrid scheduling list but may migrate among lists during runtime, offering unprecedented control over task placement. Processors are assigned to one or more hybrid scheduling lists, allowing each scheduling list to be served by multiple processors, and each processor to serve multiple lists, creating a highly flexible and adaptable scheduling environment. This configuration enables efficient and sophisticated load balancing strategies to be easily implemented. To avoid overload, each hybrid scheduling list allows defining an exclusion parameter that bound the number of processors that may simultaneously execute tasks assigned to that hybrid scheduling list. The List Control Block (LCB) structures hold all the information relative to each hybrid scheduling list, and several hybrid scheduling lists may be created. Fixed-priority and Earliest-deadline-first priority disciplines may be defined for each hybrid scheduling.

3.4 GeMRTOS Code Structure

GeMRTOS code structure defines how tasks are executed during runtime, ensuring a synchronized and adaptable system operation. All critical sections (shadow containers) of GeMRTOS are executed by granting the GeMRTOS Mutex, maintaining consistency when system status is updated. This innovative approach provides a predictable and reliable framework for managing task execution and preserving data integrity. As depicted in Figure 3.3, the code structure comprises several key sections.

The main sections of the code structure include:

- **Initialization sections:** During initialization, the GeMRTOS controller manages processor startup, keeping all but the processor with CPUID 1 in reset. This processor initializes core data structures, interrupt handlers, and executes the user-defined `main()` function. The remaining processors are then sequentially released from reset after being granted the GeMRTOS Mutex, guaranteeing an orderly initialization process.
- **Task start section:** After acquiring the GeMRTOS Mutex, the Task Start Section initializes task execution on a processor by clearing its status, selecting the highest-priority ready task or the idle task, updating the PCB, setting the processor priority in the GeMRTOS controller, handling priority inversions with trigger interrupts, updating the processor of the next time event and calling the `START_TASK` macro to begin execution.
- **Interrupt handler section:** When a processor receives an interrupt request from the GeMRTOS controller, it saves its state, disables the PIRQ, acquires the GeMRTOS Mutex, and then reads the event register to determine the appropriate action: execute the `gk_UnFrozenCallback`, the `gk_FrozenCallback` or `gk_TimeCallback` functions, or disable a DIRQ and release its trigger, and call the start section to proceed with the next task.
- **System call section:** To ensure consistent operation, system calls invoked by tasks execute within the GeMRTOS controller's critical section, releasing the GeMRTOS Mutex if no context switch is needed, otherwise invoking the start task section and guaranteeing completion upon task resumption.
- **Tasks codes:** Are executed by system processors outside the critical section of the GeMRTOS controller, but the critical section may be requested when a task code calls a system call. The task stack stores the data and information required for the task code to be executed during runtime. A TCB control block is associated with each task to hold all the task status required by the task for execution.
- **Task suspend section:** Upon task suspension, the system removes the Task Control Block (TCB) from the executing list and adds it to the waiting list, selects a new ready task for execution by the current processor, and then calls the start task section to initiate the new task.
- **Task complete section:** When a task completes, its TCB is moved to the waiting list, and if it was an Interrupt task, its DIRQ is re-enabled.

This carefully orchestrated code structure, encompassing distinct sections for initialization, task management,

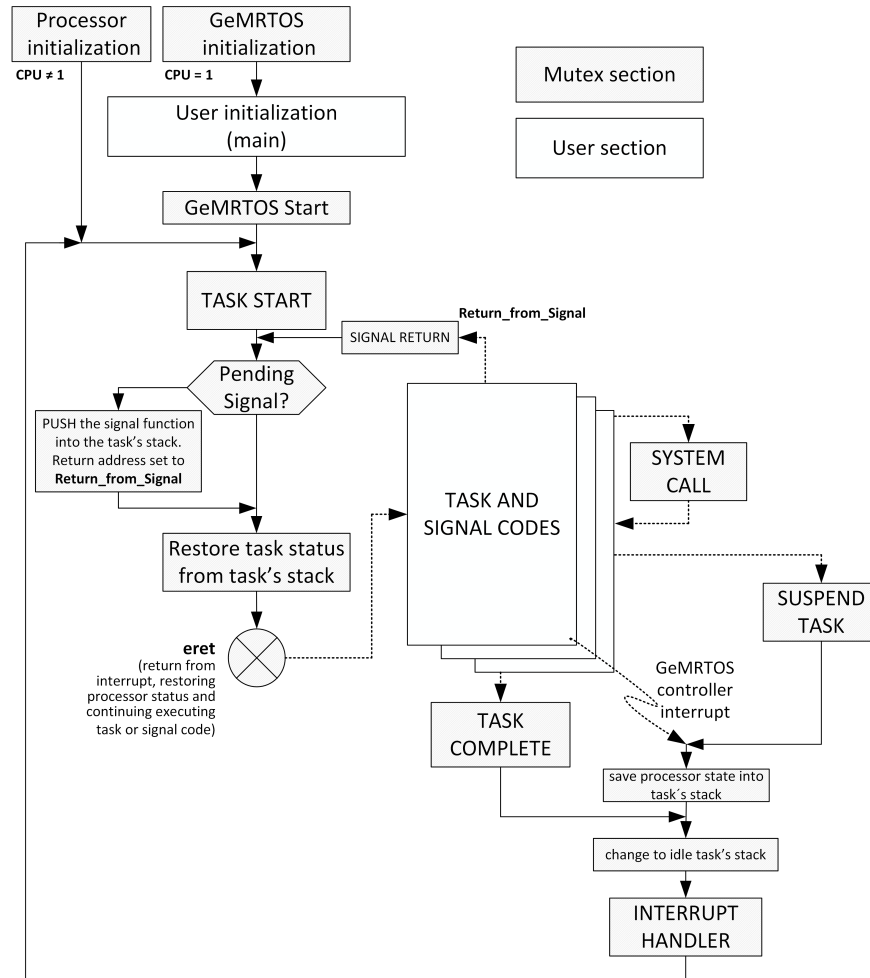


Figure 3.3: GeMRTOS code structure.

interrupt handling, and system calls, provides GeMRTOS with a robust and adaptable framework for executing real-time applications on multiprocessor systems. The subsequent sections detail each of these components and the flow of each of them.

3.4.1 Initialization Sections

During initialization, GeMRTOS creates and configures its core Control Blocks, establishing the foundation for adaptable and efficient system operation. The GeMRTOS controller manages processor startup, innovatively keeping all processors except the one with CPUID 1 in a reset state. After each processor initializes successfully, the GeMRTOS controller releases the next processor in sequence, ensuring a synchronized and orderly startup process. This approach leverages a controlled release mechanism to guarantee consistent initialization across all processors, a key innovation in GeMRTOS's multiprocessor design.

The processor with CPUID 1 is responsible for system-wide initialization:

- It initializes the GeMRTOS controller's internal registers, including the system time.
- It creates and configures all core Control Blocks.
- It sets up the initial state of all linked lists.
- It creates interrupt handlers by associating a trigger resource with each peripheral interrupt request (DIRQ), and an idle task for each processor.
- Finally, it executes the user-defined `main()` function, allowing the application to initialize its own tasks and resources.

To ensure synchronized startup, only the processor with CPUID 1 performs this configuration. All other processors remain in reset until they are granted the GeMRTOS Mutex by the previously initialized processor, guaranteeing orderly initialization before application tasks begin execution.

3.4.2 Task Start Section

The Task Start Section, executed after acquiring the GeMRTOS Mutex, plays a crucial role in initiating task execution on a processor with an innovative approach to managing the resources. In this section, the following steps are executed:

- The processor's status in its Processor Control Block (PCB) is cleared, indicating no assigned task.
- The processor searches its assigned hybrid scheduling lists in priority order to find the highest-priority task in the ready state. If no ready task is found, the processor's idle task is selected.
- The processor's PCB is updated with the selected task as the currently executing task. The task's Task Control Block (TCB) is linked to the running list of its hybrid scheduling list, based on the task's running priority.
- To support preemption, the processor priority equal to the running task priority is then updated in the GeMRTOS controller, for handling external or internal events.
- Priority inversions are checked and handled by enabling trigger interrupts on processors running lower-priority tasks blocking higher priority one with the same LCB control block.
- The processor that will execute the next timed event in the register of the GeMRTOS controller is updated, which processor associated with the TCB control block linked to the first ECB control block of the GeMRTOS time list.

This innovative approach facilitates dynamic adaptation to changing system demands. By selecting the highest-priority ready task and configuring the processor accordingly, GeMRTOS ensures efficient resource utilization and predictable real-time performance, all while maintaining a clear and structured task execution framework.

3.4.3 Interrupt Handler Section

The Interrupt Handler Section contains the code executed when a processor receives a Processor Interrupt Request (PIRQ) from the GeMRTOS controller, resulting in a processor interrupt signal. This innovative section ensures timely response to system events while maintaining system integrity. In this section, the processor performs the following steps: saves the processor state into the current executing task stack, disables its interrupts, request the GeMRTOS Mutex and reads then Event register of the GeMRTOS controller to get the event code. Depending on the event code, the following occurs:

- **EVN_CODE_FROZEN:** The `gk_FrozenCallback` function is called to handle the switching of the system to the frozen mode.
- **EVN_CODE_UNFROZEN:** The `gk_UnFrozenCallback` function is called to handle the switching of the system to the non-frozen mode.
- **EVN_CODE_TIMED:** The `gk_TimeCallback` function is called to process the interrupt associated with the timed event.
- **DIRQI_CODE:** The requesting peripheral interrupt request (DIRQ) is disabled in the GeMRTOS controller, and the associated trigger is released.

This structure allows GeMRTOS can support processors lacking native interrupt functionality. The processor checks for interrupt requests in the GeMRTOS controller when tasks are suspended or completed. To reduce interrupt latency, any processor can be assigned to wait state mode for such events, reducing the latency to near zero.

In summary, the interrupt handler section ensures that the processor safely responds to system events by saving its state, processing the appropriate routine based on the interrupt cause, disabling further interrupts, and relinquishing control to the next scheduled task, ensuring system stability and responsiveness.

3.4.4 System Call Section

GeMRTOS ensures the integrity and consistency of system operations through its unique approach to handling system calls. Instead of letting tasks directly modify sensitive system components, GeMRTOS requires that all system calls execute within the protected "critical section" of the GeMRTOS controller. Imagine it as a specialized processing zone where tasks must temporarily venture to request services from the system core. This innovative design choice guarantees that all modifications to GeMRTOS data structures and resources are performed in a tightly controlled and synchronized manner, shielding the system from data corruption and maintaining overall stability.

The journey a system call takes depends on its specific purpose. Some calls are simple and direct, requiring no change in context. In these cases, the GeMRTOS Mutex is released before returning control to the calling task, minimizing overhead. However, when a system call requires a more complex operation, such as switching to a different task, the "start task section" is invoked. This innovative process ensures that when the original task resumes, the system call will seamlessly complete its execution, preserving the integrity of the overall operation. By mandating that all state-modifying actions happen via this protected path, GeMRTOS ensures that resources, macros, and system primitives are managed reliably and predictably.

3.4.5 Task Suspend Section

The Task Suspend Section defines the adaptable process by which GeMRTOS manages task suspension, ensuring responsiveness and maximizing processor utilization. When a task's execution must be temporarily paused, GeMRTOS initiates a controlled transition. The Task Control Block (TCB) of the currently executing task

is removed from the executing list and added to the waiting list. This action effectively places the task in waiting state, awaiting a future event to trigger its resumption.

To maintain system momentum, the current processor selects a new task from the ready tasks eligible for that processor. This ensures processors always have tasks to serve and allows a seamless transition to the start task section to initiate the new task.

3.4.6 Task Complete Section

The Task Complete Section outlines the systematic process executed when a task finishes its execution, guaranteeing resource cleanup and enabling a streamlined transition to subsequent tasks. The point in which this section is reached at task completion is when a task finishes and is removed. The following key steps are then executed:

- The task's TCB is removed from the executing list, allowing the processor to be reallocated.
- The task's TCB is linked to the waiting list, preserving the task's state for potential future use.
- If the task was an Interrupt task, the corresponding peripheral interrupt request (DIRQ) is re-enabled to allow for further interrupts.
- The scheduler selects the highest priority ready task for the current processor, ensuring continuous processor utilization.
- The start task section is invoked to execute the selected task, initiating the next task's execution cycle.

This structured process ensures that upon task completion, the system removes the task from the execution list, returns it to the waiting list, re-enables the DIRQ for Interrupt tasks, and then initiates the start task routine to select and execute the next ready task. This contributes to the GeMRTOS efficient use of the resources.

3.5 GeMRTOS Linked Lists

GeMRTOS leverages a sophisticated system of linked lists to maintain system information and preserve system consistency during runtime. This approach is a major component since they enable a highly adaptable, event-driven architecture, allowing the system to respond efficiently to changing conditions. Several linked list structures are established among control block structures, forming the backbone of GeMRTOS's dynamic operation. The control block linked lists are basic in GeMRTOS and crucial in all applications. These linked lists help to preserve the status of the system.

The following subsections provide a brief overview of several key linked list structures, illustrating how GeMRTOS maintains system status and enables adaptable operation.

3.5.1 Hybrid Scheduling List Structure

GeMRTOS offers a very flexible scheduling mechanism based on hybrid scheduling lists. This innovative approach allows for dynamic task allocation and efficient processor utilization, adapting to changing system conditions and application requirements. Figure 3.4 shows an overview of the main links involved in the hybrid scheduling list structure.

Key components of this structure include:

- **LCB Control Block:** The base of the hybrid scheduling list architecture, holding information about the hybrid scheduling list. Each hybrid scheduling list will have a LCB assigned to it in order to hold all the information about the hybrid scheduling list. The root of this linked list is the KCB_NextLCBL of the

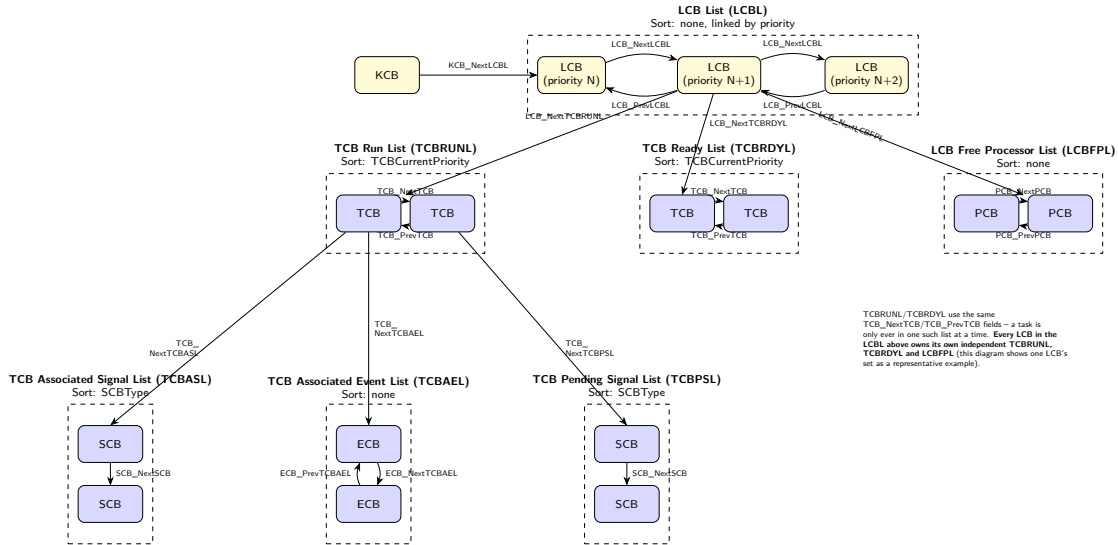


Figure 3.4: Hybrid scheduling list overview.

KCB structure, and the links between the LCB control blocks are defined by the `LCB_NextLCBL` and `LCB_PrevLCBL` fields of each LCB control block.

- **TCB Run List:** Links all TCBs of tasks in the running state assigned to the hybrid scheduling link. The root of the linked list is the `LCB_NextTCBRUNL` field of the LCB control block, and the TCB are dual linked with the fields `TCB_NextTCB` and `TCB_PrevTCB` of each TCB sorted in ascendent order by the field `TCB_CurrentPriority`.
- **TCB Ready List:** Links all TCBs in the ready state assigned to the hybrid scheduling list. The root of the linked list is the `LCB_NextTCBRDYL` field of the LCB control block and the TCB are dual linked with the fields `TCB_NextTCB` and `TCB_PrevTCB` of each TCB sorted in ascendent order by the field `TCB_CurrentPriority` (the same sort key the TCB Run List uses – confirmed against `gemrtos_list.c`'s `TCBRDYL/TCBRUNL` insertion cases, both of which use `offsetof(GS_TCB, TCB_CurrentPriority)`).
- **LCB Free Processor List:** The field of the LCB control blocks points to the LCB Free Processor list that links all the free processor assigned to the hybrid scheduling list.

Each TCB control block of tasks either in ready or running state may have links with other control blocks that may include:

- **TCB Associated Event List:** a list of ECB control blocks that may linked to resource associated lists or the time event list.
- **TCB Associated Signal list:** a list with all the SCB control blocks of the signals created for the task.
- **TCB Pending Signal list:** a list with all the TCB of the signals that are pending to be executed the next time the task goes to running state.

In summary, the hybrid scheduling partition list structure provides a flexible and adaptable framework for managing tasks and processors in GeMRTOS. By allowing dynamic task migration, efficient load balancing, and adaptable processor assignments, this innovative approach enables developers to create high-performance real-time applications that can respond effectively to changing system demands.

3.5.2 Resource Management Lists

GeMRTOS implements a generic structure to manage all the resources. A resource management list is based on the RCB control block. Figure 3.5 shows an overview of the main linked list involved in the resource management

of GeMRTOS.

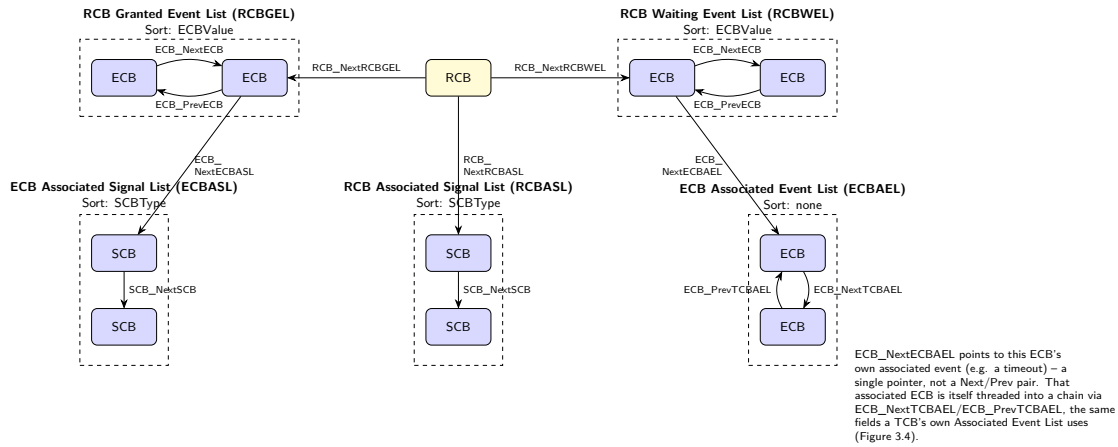


Figure 3.5: Resource management lists.

Key lists involved in the resource management include:

- **RCB Waiting Event List (RCBWEL):** This list links all the events associated with tasks waiting for a specific resource (e.g., a mutex). For a mutex resource, the RCBWEL links all events associated with tasks waiting for the mutex.
- **RCB Granted Event List (RCBGEL):** This list links all the events associated with tasks that have been granted a specific resource. For a mutex resource, the RCBGEL links all the events associated with task granting the mutex.

The linked Events in these lists are also linked with TCB to control a set of the associated tasks. In addition, ECB control blocks may also have signal control blocks linked to them in order to produce a signal in the tasks associated with those events. By using this architecture the GeMRTOS controller implement timeouts associated with events.

In summary, GeMRTOS's resource management lists provide an innovative and adaptable framework for managing access to shared resources. By dynamically linking tasks, events, and resources, GeMRTOS enables efficient resource utilization, prevents contention, and facilitates the development of robust and responsive real-time applications.

3.5.3 ECB Time List (ECBTL)

The ECB Time List (ECBTL) plays a vital role in GeMRTOS by managing timed events efficiently. Its flexible structure allows for precise scheduling and execution of time-critical operations. Figure 3.6 provides an overview of the main components of the ECBTL.

The ECBTL incorporates several essential features:

- **Sorted Structure:** All timed Event Control Blocks (ECBs) are linked in the ECBTL, sorted by their occurrence times (stored in the ECBValue field of each ECB control block).
- **Efficient Access:** The first ECB in the linked list represents the closest timed event. Its time value is consistently copied to the NEXT_OCCURRENCE_TIME register of the GeMRTOS controller, ensuring rapid access and precise timing.
- **Centralized Management:** The KCB_NextECBTL field of the Kernel Control Block (KCB) serves as the root of the list, centralizing time event management within the kernel.

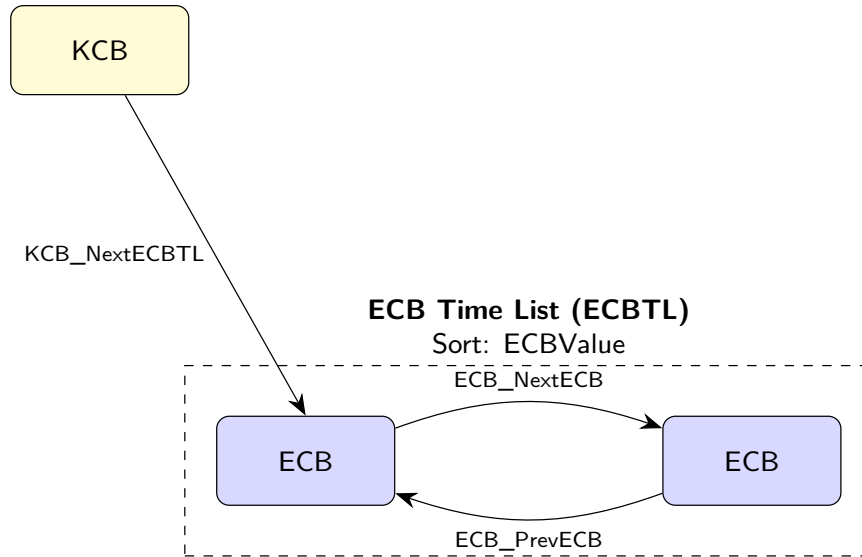


Figure 3.6: ECB Time List (ECBTL) structure.

By organizing ECBs according to their scheduled execution times, GeMRTOS efficiently determines the next event requiring processing. This innovative approach, combined with the `NEXT_OCCURRENCE_TIME` register, enables precise and adaptable scheduling of time-critical operations. As a result, GeMRTOS ensures reliable real-time performance across various applications and scenarios.

3.6 Conclusions

GeMRTOS represents a cutting-edge, highly adaptable solution for real-time multitasking on multiprocessor systems. This chapter has delineated the core structures and fundamental principles that define its architecture. The foundation of GeMRTOS is built upon four key elements:

- **GeMRTOS Controller:** This component efficiently manages external events (IRQs), processor startup, mutual exclusion (Mutex) among processors, time events, frozen and unfrozen modes, and interrupt handling.
- **GeMRTOS Control Blocks:** These highly linked data structures maintain comprehensive information about all entities within GeMRTOS, preserving system state throughout runtime.
- **GeMRTOS Code Structure:** This framework synchronizes task execution across all system processors, ensuring consistency and predictable behavior.
- **GeMRTOS Linked Lists:** These structures facilitate adaptable system management in an event-driven manner, enhancing efficiency and reducing overhead.

These core elements work in concert to enable a generic implementation of diverse mechanisms and resources, empowering GeMRTOS to excel in demanding real-time applications. The innovative design facilitates the natural implementation of sophisticated features, such as frozen and non-frozen modes. As a result, overall system performance and processor efficiency are significantly improved.

In the GeMRTOS architecture, processors are primarily dedicated to executing tasks and updating system status when events occur. The GeMRTOS controller assumes responsibility for event and time management, further enhancing system resource utilization. This strategic division of labor ensures that GeMRTOS can meet the exacting demands of complex real-time systems while maintaining flexibility and scalability.

By leveraging these advanced features and architectural design principles, GeMRTOS positions itself as a powerful and versatile solution for a wide range of real-time multiprocessor applications.

GeMRTOS Controller

4.1 Introduction to the GeMRTOS Controller

The GeMRTOS controller is a pivotal component of the GeMRTOS architecture, providing advanced management of time, external events, and mutual exclusion among system processors. Its design ensures coherent access to the system state, facilitating robust and predictable system operation. The controller incorporates various configuration and status registers, which are mapped into the processors' memory space.

These memory-mapped registers are organized into two distinct sections, as illustrated in Figure 4.1:

The two memory sections are:

- `gemrtos_global_region`: This shared memory region is accessible to all system processors. It begins at the memory address `GEMRTOS_DRIVER_BASE` and contains control and status registers common to all processors.
- `gemrtos_processor_region`: This shared memory region provides each processor with access to an individual shadow memory area. The registers mapped to this region are specific to each processor and may contain different values depending on the accessing processor. For example, the `CPU_ID` register returns a unique identifier assigned to the processor reading the register. This memory region starts at the address `GEMRTOS_DRIVER_PROCESSOR_BASE`.

A comprehensive understanding of these memory regions and their associated registers is essential for effectively utilizing the GeMRTOS controller's capabilities. The subsequent sections will explore each region in detail, elucidating the functions performed by the GeMRTOS Controller during system operation.

This introduction provides a foundation for understanding the GeMRTOS controller's role in managing system resources and facilitating efficient inter-processor communication. The innovative design of the controller, with its distinct memory regions, contributes significantly to the overall performance and flexibility of the GeMRTOS architecture.

4.2 GeMRTOS Global Region

The `gemrtos_global_region` contains all the control and status registers of the GeMRTOS Controller common to all the processors. These registers enable coordinated system-wide management and provide a centralized point for configuring key parameters. Table 4.1 shows the registers according to the address offset referred to the `GEMRTOS_DRIVER_BASE` memory address.

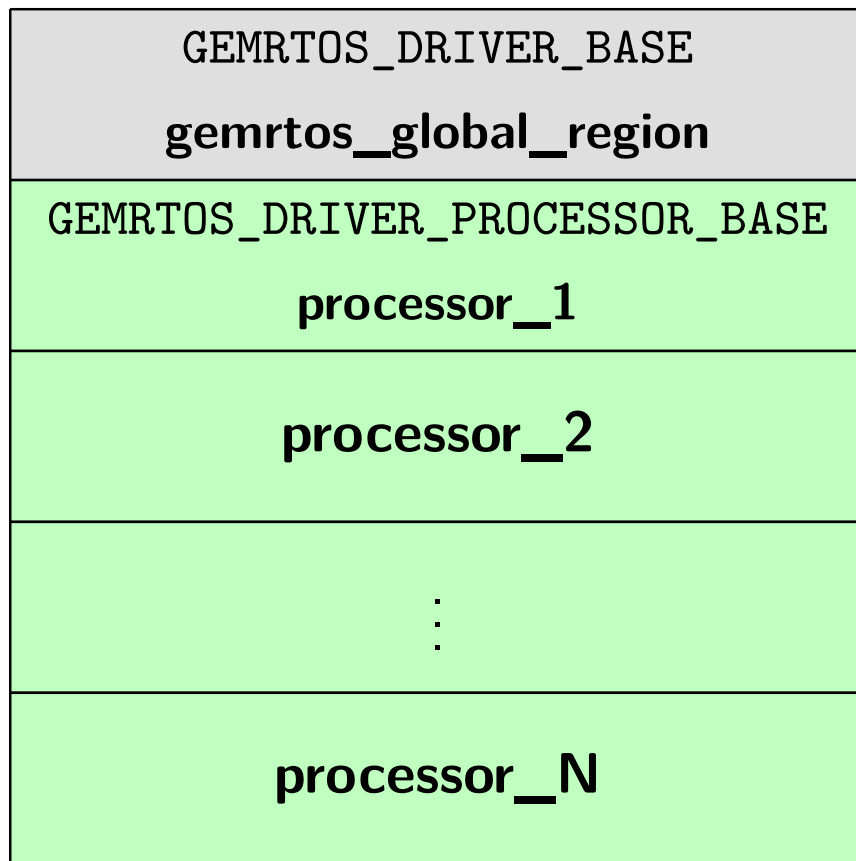


Figure 4.1: Memory-mapped regions of GeMRTOS Controller registers.

Table 4.1: Global region registers of the GeMRTOS controller.

Dword Offset	Name	R/W	Description
0x00-0x07	PRC_CDE_ADDRx	R/W	Reset Code for CPU_ID $\neq$ 1 PRC_CDE_ADDRx registers store the reset code for processors with CPU_ID different from 1.
0x08-0x0F	PRC_CDE_ADDRx	R/W	Interrupt Return Code The PRC_CDE_ADDRx registers also hold the return-from-interrupt instruction code and the sequence to restore the last set of registers.
0x10	STATUS_CTRL	R/W	Status Register (STATUS) The Control Register (CTRL) is accessed when reading. Control Register (CTRL) The Control Register (CTRL) is accessed when writing.
0x11	PRC_IRQ_ACK	R	PRC_IRQ_SECTION Register This register provides a mechanism for synchronizing interrupt handling across multiple processors in the GeMRTOS system. By checking the status of the PRC_IRQ_SECTION register, a processor can determine if the target processor has entered the IRQ section and is ready to process the interrupt.
0x12	IRQ_RQS_PND	R	IRQ_RQS_PND Register The IRQ_RQS_PND register returns the state of the interrupt requests from the system devices. This register provides a centralized view of all the pending interrupt requests in the GeMRTOS system.
0x13	Reserved	-	-
0x14	SYS_TM_HGH	R	SYS_TM_HGH Register The SYS_TM_HGH register holds the most significant word of the System Time register. This register, along with the SMP register, forms the complete 64-bit system time counter maintained by the GeMRTOS controller.
0x15	FRZ_THR	R/W	FRZ_THR Register The FRZ_THR register holds the most significant word of the frozen threshold value. This register, along with the SMP register, forms the complete 64-bit frozen threshold value used by the GeMRTOS controller.
0x16	FRZ_TM	R	FRZ_TM Register The FRZ_TM register holds the most significant word of the frozen time value maintained by the GeMRTOS controller. This register, along with the SMP register, represents the complete 64-bit frozen time counter. content...
0x17	TM_CNT	R	TM_CNT Register The TM_CNT register holds the most significant word of the non-frozen time counter maintained by the GeMRTOS controller. This register, along with the SMP register, represents the complete 64-bit non-frozen time value.

Continued on next page

Table 4.1: Global region registers of the GeMRTOS controller. (Continued)

Dword Offset	Name	R/W	Description
0x17	NXT_OCC_TM	W	NXT_OCC_TM Register The NXT_OCC_TM register is used to set the most significant word of the next occurrence time in the GeMRTOS controller. This register, in conjunction with the SMP register, allows the software to specify the complete 64-bit value for the next timed event.
0x19	INOUTS	R/W	INOUTS Register The INOUTS register serves as a versatile interface for handling both inputs and outputs of the GeMRTOS controller. This dual-purpose register provides a convenient mechanism for the software to interact with external devices and signals, facilitating efficient I/O operations within the GeMRTOS environment.
0x1A	RST_CNT	R	RST_CNT Register The RESET register is a crucial component in the GeMRTOS controller that facilitates the coordinated and sequential start-up of multiple processors in a multi-core system. This register provides essential information for synchronizing the initialization process across different processor cores.
0x1B	SMP	R/W	SMP Register The SMP register is a crucial component in the GeMRTOS controller that facilitates 64-bit data transfers. This register works in conjunction with other registers to enable reading and writing of 64-bit values across the system.
0x1C	MTX_GRN	R	MTX_GRN Register The MTX_GRN (Mutex Grant) register is a crucial component in the GeMRTOS controller that manages mutex (mutual exclusion) allocation across multiple processors in a multi-core system. This register provides a bitmap representation of mutex ownership, allowing for efficient synchronization and resource sharing among processors.

4.2.1 Reset Code for CPU_ID $\neq$ 1

PRC_CDE_ADDRx registers store the reset code for processors with CPU_ID different from 1. This mechanism is implemented to overcome two common limitations:

- Restrictions in reset regions of certain processors
- Constraints in defining multiple reset vectors in compilation tools

By storing reset code for secondary processors in these registers, GeMRTOS ensures proper initialization across all cores in a multi-processor system, regardless of individual processor reset capabilities. This approach enables:

1. Flexible boot sequences for heterogeneous multi-core systems
2. Consistent initialization procedures across diverse hardware configurations
3. Enhanced portability of GeMRTOS to various processor architectures

The PRC_CDE_ADDRx registers thus play a crucial role in the robust and adaptable startup process of GeMRTOS in complex multiprocessor systems.

4.2.2 Interrupt Return Code

The `PRC_CDE_ADDRx` registers also hold the return-from-interrupt instruction code and the sequence to restore the last set of registers. This mechanism serves two important purposes:

1. It enables the processor to execute the interrupt return operation.
2. It ensures the release of the GeMRTOS Controller Mutex at the same time.

When a processor needs to return from an interrupt, it executes the stored return-from-interrupt instruction. This not only completes the interrupt handling, but also automatically releases the GeMRTOS Controller Mutex that was acquired during the interrupt processing.

This approach provides a tightly integrated and efficient way to handle interrupt returns. By coupling the interrupt return logic with the Mutex release, GeMRTOS minimizes the overhead and potential race conditions that could arise if these operations were performed separately.

The ability to atomically return from an interrupt while releasing the Mutex is a key design feature that enhances the responsiveness and determinism of the GeMRTOS system, particularly in real-time and safety-critical applications.

4.2.3 Status Register (STATUS)

The Status Register (STATUS) provides visibility into the internal state of the GeMRTOS controller. It reports the last event that occurred, the current time prescale, and the status of the time counters and frozen mode. This information is crucial for the software to monitor the overall health and behavior of the real-time operating system.

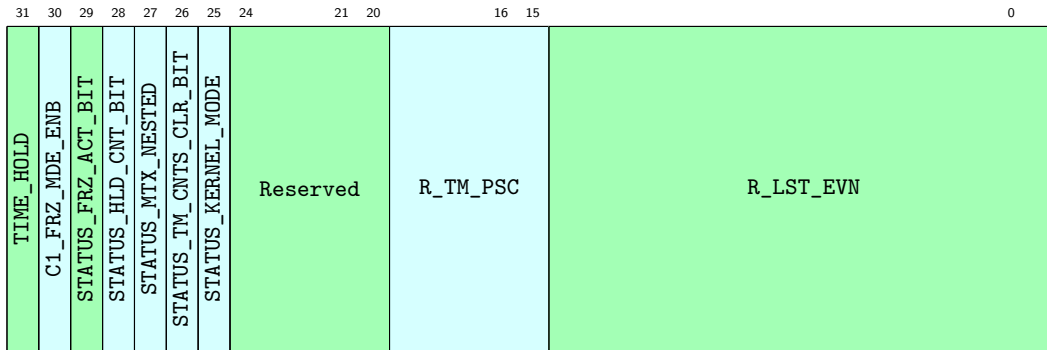


Figure 4.2: Status Register fields (STATUS)

The Status Register (STATUS) returns the following information when read (Fig. 4.2):

- **Bits 15:0 (R_LST_EVN):** Last Event Occurred (LST_EVN_OCC) with the following priorities:
 - `EVN_CODE_FROZEN`: the GeMRTOS controller indicates that it has switched to frozen mode.
 - `EVN_CODE_UNFROZEN`: the GeMRTOS controller indicates that it has switched to non-frozen mode.
 - `EVN_CODE_TIMED`: a timed event happened because the system time counter reached the next time event.
 - `DIRQi_CODE`: the IRQ *i* took place. The trigger release associated is released.
- **Bits 20:16 (R_TM_PSC):** Current time prescale. It is the value defined by setting bit 14 in the control register.
- **Bit 25 (STATUS_KERNEL_MODE):** Determines if the GeMRTOS controller is in kernel mode. In kernel mode, requests for Critical Section are not nested.

- **Bit 26** (STATUS_TM_CNTS_CLR_BIT): Determines if time counter of the GeMRTOS controller are reset and hold.
- **Bit 27** (STATUS_MTX_NESTED): Determines if granted to Critical Section is nested.
- **Bit 28** (STATUS_HLD_CNT_BIT): Status of the reset of the time counters: When 1, the time counters are reset and hold. The time counters are reset and hold reset, when execution `gm_TimeCountersReset()`. The time counters are release to run again when executing `gm_TimeCoutersUnreset()`.
- **Bit 29** (STATUS_FRZ_ACT_BIT): Frozen mode active. This bit is 1 if the GeMRTOS controller is in frozen mode, and 0 if it is in non-frozen mode.
- **Bit 30** (STATUS_FRZ_ENB_BIT): Frozen mode enable. This bit is 1 if frozen mode is enabled, 0 if it is disabled. The execution of `gm_FrozenModeEnable()` enables the frozen mode. The execution of `gm_FrozenModeDisable()` disables the frozen mode.
- **Bit 31** (STATUS_TIME_HOLD_BIT): This bit is status of the TIME_HOLD bit (1 time is held, 0 time is running). The execution of `gm_TimeCountersHold()` holds the time counters in its current state. The execution of `gm_TimeCountersUnhold()` unholds the time counters.

4.2.4 Control Register (CTRL)

The Control Register (CTRL) provides a comprehensive set of configuration and control options for the GeMRTOS controller. It allows the software to precisely manage the time prescale, interrupt handling, task scheduling, mutex management, and various operational modes of the GeMRTOS system. Figure 4.3 shows all bit of the Control Register.

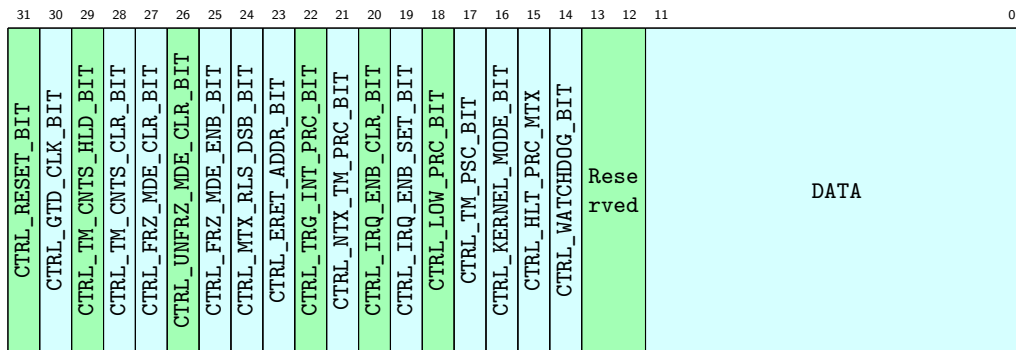


Figure 4.3: Control Register fields (CTRL)

The Control Register (CTRL) is accessed when writing. All the bits are cleared after the command is executed. The command is completed when the bit is set. The meaning of the bits is:

- **Bits 11:0** (DATA): Data for specific operation. This data is used according to the operation in the other command bits.
- **Bit 14** (CTRL_WATCHDOG_BIT): Set the per-processor watchdog timeout in the 12-bit data field (bits 11:0). The watchdog counter uses the 12-bit data field multiplied by $G_WATCHDOG_TIMEOUT_SHIFT = 65536$ (i.e. shifted left 16 bits), counted at the *controller clock* frequency. Unlike the system time counter, the watchdog countdown is not prescaled, so the interval is $data \times 65536 / f_{clk}$ and depends on the clock the controller is generated with: on a 50 MHz system that is approximately 1.31 ms (data = 1) to 5.37 s (data = 0xFFE). The power-up default is 0xFFFF, which is also the reserved value meaning “watchdog disabled”: with 0xFFFF configured no processor’s countdown runs, whatever its individual enable bit says. The largest usable interval is therefore 0xFFE. Enabling/disabling the countdown is a *separate* bit, PRC_WATCHDOG_ENB_BIT (ADDR_REG_1/ADDR_REG_2, §4.3.2) – writing this timeout value does not by itself arm anything; both this bit and the enable bit must be set. `gm_WatchdogTimeoutSet(timeout)` sets this field. See §4.5 for the

full watchdog design.

- **Bit 15** (CTRL_HLT_PRC_MTX): If Bit 0 is equal to 1, the GeMRTOS halts processors when Mutex is granted for debugging. If Bit 0 is equal to 0, the Mutex is released and processors are unhalted. Only for internal use.
- **Bit 16** (CTRL_KERNEL_MODE_BIT): If Bit 0 is equal to 1, the GeMRTOS controller is set to kernel mode, otherwise is set to non-kernel mode. The `gm_KernelModeSet()` function sets the GeMRTOS controller in kernel mode, whilst the `gm_KernelModeReset()` function sets the GeMRTOS controller in non-kernel mode.
- **Bit 17** (CTRL_TM_PSC_BIT): Set the time prescale register (R_TM_PSC) using the data bits 7:0. The time prescale is often configured to obtain a 10MHz time counter from the system clock frequency. The `gm_SystemTimePrescaleSet` function sets the system clock prescale to get the system time unit. The `gm_SystemTimePrescaleGet()` function returns the current time prescale.
- **Bit 18** (CTRL_LOW_PRC_BIT): Set the lowest priority processor detailed in bits 7:0. The kernel `gk_SetLowestProcessor` function sets the lowest priority bit.
- **Bit 19** (CTRL_IRQ_ENB_SET_BIT): Set the IRQ enable bit in the R_IRQ_ENB register of the GeMRTOS controller, enabling the IRQ of the IRQ detailed in bits 7:0. The `gm_IrqEnable(irq)` function is used to enable the irq external interrupt in the GeMRTOS controller. The `gm_IrqDisable(irq)` function disables the irq external interrupt in the GeMRTOS controller.
- **Bit 20** (CTRL_IRQ_ENB_CLR_BIT): Clear the IRQ enable bit in the R_IRQ_ENB register of the GeMRTOS controller, disabling the IRQ of the IRQ detailed in bits 7:0. The `gm_IrqDisable(irq)` function is used to disable the irq interrupt in the GeMRTOS controller. The `gm_IrqEnable(irq)` function enables the irq external interrupt in the GeMRTOS controller.
- **Bit 21** (CTRL_NTX_TM_PRC_BIT): Set the processor executing the next timed event detailed in bits 7:0 (R_NXT_TM_EVN_PRC register of the GeMRTOS controller). `gk_NextOccTimeProcessor()` is used by the kernel to set the processor executing the next time occurrence task (0 if next time occurrence task is not executing).
- **Bit 22** (CTRL_TRG_INT_PRC_BIT): Trigger processor interrupt defined in bits 7:0. The `gm_TriggerPrcIRQ(proc)` produces an interrupt to processor proc.
- **Bit 23** (CTRL_ERET_ADDR_BIT): Set the interrupt return (R_ERET_REG) address specified in bits 7:0. 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. The `gm_ERETAddressSet` function is used by the kernel to configure this register.
- **Bit 24** (CTRL_MTX_RLS_DSB_BIT): If Bit 0 is equal to 0, enable GeMRTOS controller Mutex to be released next time the Mutex is released. It is used to implement nested Mutex requests by GeMRTOS. If Bit 0 is equal to 1, disable Mutex to be released next time the Mutex is released. It is used to implement nested Mutex requests by GeMRTOS. The `gm_MutexNestedValue` function sets the GeMRTOS controller Mutex nested parameter according to the value.
- **Bit 25** (CTRL_FRZ_MDE_ENB_BIT): If Bit 0 is equal to 0, disable frozen mode. When frozen mode is disabled, the GeMRTOS controller remains in non-frozen mode even though the system time overcame the next event by the frozen threshold. The `gm_FrozenModeDisable()` function disables the frozen mode. The `gm_FrozenModeEnable()` function enables the frozen mode event. If Bit 0 is equal to 1, enable frozen mode. The `gm_FrozenModeEnable()` function enables the frozen mode event. The `gm_FrozenModeDisable()` function disables the frozen mode.
- **Bit 26** (CTRL_UNFRZ_MDE_CLR_BIT): UnFrozen mode event flag clear (C1_UNFRZ_EVN). When the GeMRTOS controller indicates an unfrozen mode event, this bit should be set to acknowledge the indication and continue with the next events. The `gm_UnFrozenEventClear()` function clears the C1_UNFRZ_EVN flags. It should be executed when the event is active in order to allow others event to be informed by the

GeMRTOS controller. Is is executed only by the GeMRTOS kernel.

- **Bit 27** (CTRL_FRZ_MDE_CLR_BIT): Frozen mode event flag clear (C1_FRZ_EVN). When the GeMRTOS controller indicates a frozen mode event, this bit should be set to acknowledge the indication and continue with the next events. The `gm_FrozenEventClear()` function clears the frozen flag (C1_FRZ_EVN). It should be executed when the event is active in order to allow others event to be informed by the GeMRTOS controller. Is is executed only by the GeMRTOS kernel.
- **Bit 28** (CTRL_TM_CNTS_CLR_BIT): If Bit 0 is equal to 1, reset and keep reset the frozen and non-frozen time counters. If Bit 0 is equal to 0, set running the frozen and non-frozen time counters. The `gm_TimeCountersReset()` function is executed at system start-up before executing `gm_TimeCountersUnreset()`.
- **Bit 29** (CTRL_TM_CNTS_HLD_BIT): If Bit 0 is equal to 0, unhold the GeMRTOS time counters. The `gm_TimeCountersUnhold()` function is used to unhold the GeMRTOS time counters of the GeMRTOS controller. If Bit 0 is equal to 1, hold the GeMRTOS time counters. All time counts in the GeMRTOS controller are held at the current values. The `gm_TimeCountersHold()` function is used to hold the GeMRTOS time counters of the GeMRTOS controller.
- **Bit 30** (CTRL_GTD_CLK_BIT): If Bit 0 is equal to 0, disable the output clock port of the GeMRTOS controller. The output clock port turns to low when disabled. The `gm_GatedClockDisable()` function is used to disable the gated clock output of the GeMRTOS controller. If Bit 0 is equal to 1, enable the output clock port of the GeMRTOS controller. The output clock port is the GeMRTOS controller input clock when enabled. The `gm_GatedClockEnable()` function is used to enable the gated clock output of the GeMRTOS controller.
- **Bit 31** (CTRL_RESET_BIT): Reset the GeMRTOS controller. By writing a 1 to this bit, all the internal registers of the GeMRTOS controller are reset to their initial state. The `gm_SystemReset()` function is used to set this control bit.

Together, the CTRL and STATUS registers form a powerful interface for the GeMRTOS software to interact with the underlying hardware controller, enabling fine-grained control and observability of the real-time execution environment.

4.2.5 PRC_IRQ_SECTION Register

The PRC_IRQ_SECTION register returns the state of the processors when they are in the IRQ section of GeMRTOS. When a processor interrupts another processor by triggering a hardware interrupt, it must wait until the corresponding bit in this register is set in order to be sure the interrupt was successful.

The use of this register ensures that inter-processor interrupts are handled in a deterministic and reliable manner, preventing race conditions and guaranteeing the successful delivery of interrupts. This is a key feature for maintaining the real-time responsiveness and synchronization of the GeMRTOS system in a multiprocessor environment.

The information provided by the PRC_IRQ_SECTION register is crucial for the GeMRTOS software to coordinate the execution of interrupt service routines across the various processors, enabling efficient and robust interrupt handling throughout the system.

4.2.6 SYS_TM_HGH Register

The SYS_TM_HGH register holds the most significant word of the system time register. This register, along with the SMP register, forms the complete 64-bit system time counter maintained by the GeMRTOS controller.

After reading the SYS_TM_HGH register, the software should also read the SMP register to obtain the lowest significant word of the system time. Together, these two registers provide the full 64-bit system time value.

The System Time register represents the total elapsed time tracked by the GeMRTOS controller. It is the sum of the non-frozen time and the frozen time, providing a comprehensive measure of the total time elapsed in the system.

This System Time value is crucial for the real-time scheduling and event management within the GeMRTOS framework. The software can use the system time to schedule tasks, measure execution durations, and coordinate time-sensitive operations across the system.

By maintaining a 64-bit system time counter, the GeMRTOS controller ensures that the time representation has sufficient resolution and range to accommodate the requirements of even the most demanding real-time applications.

The SYS_TM_HGH and SMP registers provide a reliable and efficient mechanism for the software to access the comprehensive system time information managed by the GeMRTOS controller.

4.2.7 FRZ_THR Register

The FRZ_THR register holds the most significant word of the frozen threshold value. This register, along with the SMP register, forms the complete 64-bit frozen threshold value used by the GeMRTOS controller.

The frozen threshold represents the maximum allowed duration for the system to be in the non-frozen mode. When the system time exceeds this threshold from the last time event missed, the GeMRTOS controller will automatically exit the non-frozen mode.

To set or read the full 64-bit frozen threshold value, the software should use the FRZ_THR register to access the most significant word, and the SMP register to access the least significant word.

This two-part register structure allows the GeMRTOS software to precisely configure and monitor the frozen threshold, which is a critical parameter for maintaining the real-time responsiveness and consistency of the system.

By providing access to the complete 64-bit frozen threshold value, the FRZ_THR and SMP registers enable the GeMRTOS software to adapt the system's behavior to the specific requirements of the target application, ensuring optimal real-time performance and fault tolerance.

4.2.8 FRZ_TM Register

The FRZ_TM register holds the most significant word of the frozen time value maintained by the GeMRTOS controller. This register, along with the SMP register, represents the complete 64-bit frozen time counter.

The frozen time is a measure of the total duration that the GeMRTOS system has spent in the frozen mode. This value is accumulated whenever the controller enters the frozen mode, which is typically used to temporarily suspend the real-time execution in response to certain events or conditions.

To read the full 64-bit frozen time value, the software should first access the FRZ_TM register to obtain the most significant word, and then read the SMP register to get the least significant word.

By providing access to the complete frozen time information, the GeMRTOS software can monitor and analyze the system's behavior in the frozen mode.

The FRZ_TM register is a crucial component of the GeMRTOS time management infrastructure, allowing the software to maintain a comprehensive understanding of the system's temporal behavior and adapt its strategies accordingly.

4.2.9 TM_CNT Register

The TM_CNT register holds the most significant word of the non-frozen time counter maintained by the GeMRTOS controller. This register, along with the TIME_LSW register, represents the complete 64-bit non-frozen time value.

The non-frozen time is a measure of the total elapsed time that the GeMRTOS system has spent in the non-frozen mode. This value is continuously updated by the controller and is the basis for all time-related operations, such as task scheduling, timeouts, and time-stamping of events.

To read the full 64-bit non-frozen time value, the software should first access the TM_CNT register to obtain the most significant word, and then read the SMP register to get the least significant word.

The non-frozen time value is a crucial component of the GeMRTOS time management system. It represents the "wall clock" time that is valid for all timed events and tasks within the real-time operating system. This information is essential for the software to coordinate the execution of real-time applications and ensure that timing constraints are met.

By providing access to the complete non-frozen time information through the TM_CNT and SMP registers, the GeMRTOS software can accurately track the passage of time, schedule tasks, and measure the performance of the system, even in the presence of frozen mode operations.

This time management functionality is a cornerstone of the GeMRTOS real-time capabilities, enabling the system to maintain deterministic and predictable behavior in complex, time-critical applications.

4.2.10 NXT_OCC_TM Register

The NXT_OCC_TM register is used to set the most significant word of the next occurrence time in the GeMRTOS controller. This register, in conjunction with the SMP register, allows the software to specify the complete 64-bit value for the next scheduled event or task occurrence.

The next occurrence time is a critical parameter in the GeMRTOS scheduling system. It allows the software to precisely specify when the next scheduled event should occur, enabling fine-grained control over the timing of real-time operations.

By providing a 64-bit value for the next occurrence time, GeMRTOS ensures that the system can handle a wide range of timing requirements.

This register plays a crucial role in maintaining the deterministic behavior of the real-time system, allowing for accurate and reliable scheduling of tasks and events across the entire GeMRTOS environment.

4.2.11 INOUTS Register

The INOUTS register serves as a versatile interface for handling both inputs and outputs of the GeMRTOS controller. This dual-purpose register provides a convenient mechanism for the software to interact with external devices and signals, facilitating efficient I/O operations within the GeMRTOS environment.

Key features of the IO register:

- Input and Outputs: Functions as both an input and output register.
- Direct access: Allows easy reading of inputs and writing of outputs.
- Simplified I/O handling: Enables straightforward management of I/O operations from within programs.

Functionality:

1. Input operations:
 - Reading the INOUTS register returns the current state of the input pins.
 - Allows the software to quickly sample external signals or sensor data.
2. Output operations:
 - Writing to the INOUTS register sets the state of the output pins.
 - Enables the software to control external devices or signal other components.

The INOUTS register plays a crucial role in the GeMRTOS system by:

- Simplifying I/O programming: Provides a unified interface for both input and output operations.
- Enhancing real-time responsiveness: Allows quick access to I/O states without complex peripheral configurations.
- Facilitating hardware abstraction: Enables software to interact with external hardware through a standardized register.
- Supporting diverse applications: Accommodates a wide range of I/O requirements for various real-time systems.

By incorporating the INOUTS register, GeMRTOS offers developers a powerful and flexible tool for managing input and output operations efficiently. This capability is essential for creating responsive and interactive real-time applications that can effectively interface with the external environment.

4.2.12 RST_CNT Register

The RST_CNT register is a crucial component in the GeMRTOS controller that facilitates the coordinated and sequential start-up of multiple processors in a multi-core system. This register provides essential information for synchronizing the initialization process across different processor cores.

Key features of the RST_CNT register:

- Returns the ID of the last processor unreset
- Enables sequential processor start-up
- Supports synchronization in multi-core systems

The RST_CNT register plays a vital role in the GeMRTOS system by:

- Ensuring orderly system initialization: Helps maintain a predictable and controlled start-up sequence.
- Facilitating inter-processor coordination: Allows each processor to be aware of the system's overall initialization state.
- Supporting scalability: Enables GeMRTOS to efficiently manage systems with varying numbers of processor cores.
- Enhancing system reliability: Reduces the risk of initialization-related issues in multi-processor environments.

By incorporating the RST_CNT register, GeMRTOS provides a robust mechanism for managing the complex process of initializing multi-processor systems. This feature is essential for ensuring that all processor in the system start up in a coordinated and efficient manner, laying the foundation for reliable and deterministic operation of real-time applications in multi-core environments.

4.2.13 SMP Register

The SMP register is a crucial component in the GeMRTOS controller that facilitates 64-bit data transfers. This register works in conjunction with other registers to enable reading and writing of 64-bit values across the system.

Key features of the SMP register:

- Support for 64-bit data transfers
- Acts as a temporary storage for least significant words

- Coordinates with other registers for complete 64-bit operations

Functionality:

1. Reading 64-bit values:
 - First, read the most significant word (MSW) from the appropriate register.
 - Then, read the SMP register to obtain the least significant word (LSW).
2. Writing 64-bit values:
 - First, write the least significant word (LSW) to the SMP register.
 - Then, write the most significant word (MSW) to the appropriate register.

The SMP register plays a vital role in the GeMRTOS system by:

- Enabling precise 64-bit operations: Allows for accurate representation and manipulation of large values or time stamps.
- Maintaining data integrity: Ensures that 64-bit values are read or written atomically.
- Simplifying software implementation: Provides a consistent mechanism for handling 64-bit transfers across different registers.

By incorporating the SMP register, GeMRTOS provides a robust and efficient mechanism for handling 64-bit data transfers. This capability is essential for maintaining high precision in time-critical operations and supporting applications that require extended range values in real-time systems.

4.2.14 MTX_GRN (Mutex Grant) Register

The MTX_GRN (Mutex Grant) register is a crucial component in the GeMRTOS controller that manages mutex (mutual exclusion) allocation across multiple processors in a multi-core system. This register provides a bitmap representation of mutex ownership, allowing for efficient synchronization and resource sharing among processors.

The MTX_GRN register plays a vital role in the GeMRTOS system by:

- Ensuring data integrity: Prevents simultaneous access to shared resources by multiple processors.
- Facilitating inter-processor communication: Provides a hardware-level mechanism for synchronization.
- Enhancing system performance: Allows for quick mutex status checks without software overhead.
- Supporting scalability: Can accommodate systems with varying numbers of processor cores.

By incorporating the MTX_GRN register, GeMRTOS provides a robust and efficient mechanism for managing mutex grants in multi-processor systems. This feature is essential for ensuring proper synchronization and resource sharing among multiple processors, contributing to the overall reliability and performance of real-time applications in complex, multi-processor environments.

This overview of the GeMRTOS global region provides a foundation for understanding the core control mechanisms within the system. The subsequent section will delve into the specifics of the processor region, detailing the registers that enable tailored management of individual processors.

4.3 GeMRTOS Processor Region

The GeMRTOS Processor Region is a crucial component of the GeMRTOS Controller architecture, providing individual control and status registers for each processor in a multi-processor system. This region facilitates efficient management and monitoring of processor-specific operations within the GeMRTOS environment.

Key features of the GeMRTOS Processor Region:

- Individual registers for each processor
- Accessible through individual memory-mapped bus for each processor
- Enables fine-grained control and status monitoring avoiding race conditions

The registers in the GeMRTOS Processor Region are organized based on address offsets from the GEMRTOS_DRIVER_PROCESSOR_BASE memory address. This structure allows for consistent and predictable access to processor-specific registers across different system configurations.

Table 4.2 shows the registers and their corresponding address offsets within the GeMRTOS Processor Region.

Table 4.2: GeMRTOS Processor Region Registers.

Dword Offset	Register Name	R/W	Description
0x00	ADDR_REG_0	R/W	The ADDR_REG_0 register manages processor states, mutex grants, and interrupt requests in the GeMRTOS system. Its behavior differs between read and write operations.
0x01	ADDR_REG_1	R/W	The ADDR_REG_1 register allows for fine-grained management of processor interrupts, mutex operations, and processor states.
0x02	ADDR_REG_2	R/W	The ADDR_REG_2 register allows for fine-grained management of disabling processor states.

The GeMRTOS Processor Region plays a vital role in the overall system by providing several key benefits:

- **CPU_ID Assignment:**
 - Assigns a unique CPU_ID for each processor without requiring special registers in processor architectures.
 - Simplifies processor identification and management across diverse hardware platforms.
- **Scalable Multi-Processor Support:**
 - Facilitates efficient management of systems with parameterizable numbers of processors.
 - Enables seamless scaling from uniprocessor to multi-processor configurations without major software modifications.

These features collectively contribute to the robustness and versatility of the GeMRTOS system:

1. **Hardware Abstraction:** By providing a consistent method for CPU identification and management, GeMRTOS simplifies porting to different processor architectures.
2. **Scalability:** The system can easily adapt to different multi-processor configurations, supporting a wide range of embedded and real-time applications.
3. **Customization:** Developers can fine-tune the behavior of individual processors to meet specific performance or functionality requirements.

By providing a dedicated region for processor-specific registers, GeMRTOS offers a powerful and flexible framework for developing and managing complex real-time systems in multi-processor environments.

4.3.1 ADDR_REG_0 Register

This register manages processor states, mutex grants, and interrupt requests in the GeMRTOS system. Its behavior differs between read and write operations.

Write Operation: When writing to this register:

- If the PRC_HLT_ENB_BIT (Processor Halt Enable) bit is set, the processor enters a wait state when writing the ADDR_REG_0 address in the processor memory region. The processor exits from wait when the GeMRTOS controller Mutex is granted to the processor or an interrupt request is issued to the processor. The `gm_PrcInterruptEnable()` function sets the halt bit.

- The processor is released from this wait state when either:
 - The processor is granted the GeMRTOS Mutex, or
 - The processor receives an interrupt request.

This mechanism allows for efficient processor synchronization and power management by halting the processor until specific conditions are met.

Read Operation: When reading from this register, the following information is available (Figure 4.4):

- **Bit 0: PRC_HLT_ENB_BIT State**
 - Indicates the current state of the Processor Halt Enable for this processor.
 - 0: Processor Halt is disabled
 - 1: Processor Halt is enabled
- **Bit 1 (R_MTX_PRC_GRN): GeMRTOS Mutex Grant State**
 - Indicates whether this processor has been granted the GeMRTOS Mutex.
 - 0: Mutex not granted to this processor
 - 1: Mutex granted to this processor
- **Bit 2 (R_PRC_INT): Processor Interrupt Request**
 - Indicates whether an interrupt request is pending for this processor.
 - 0: No interrupt request pending
 - 1: Interrupt request pending



Figure 4.4: Read ADDDR_REG_0 Register

Functionality: This register serves several crucial functions in the GeMRTOS system:

1. **Processor State Control:** Allows for fine-grained control over processor execution states, enabling power-saving modes when the processor is idle.
2. **Synchronization:** Facilitates inter-processor synchronization through GeMRTOS Mutex management, crucial for maintaining data integrity in multi-processor systems.
3. **Interrupt Management:** Provides a mechanism for monitoring and controlling interrupt requests, essential for real-time responsiveness.
4. **Status Monitoring:** Enables software to quickly assess the current state of the processor, including its halt status, GeMRTOS Mutex ownership, and pending interrupts.

4.3.2 ADDR_REG_1 Register

The ADDR_REG_1 register is a crucial component in the GeMRTOS controller, providing control over various processor-specific functions. This register allows for fine-grained management of processor interrupts, mutex operations, and processor states. Figure 4.5 shows the ADDR_REG_1 bit assignment.

Register Bit Assignments:

- **Bit 0 (PRC_INT_ENB_BIT): Processor Interrupt Enable**

- When set, enables interrupts for the processor.
- Allows the processor to respond to interrupt requests from the GeMRTOS controller.
- **Bit 1 (PRC_MTX_RSV_BIT): Mutex Reservation**
 - When set, requests the GeMRTOS controller to grant the mutex to this processor.
 - Used for synchronization in multi-processor operations.
- **Bit 2 (PRC_IRQ_ACK_BIT): Interrupt Request Acknowledgement**
 - When set (value=1), clears/acknowledges the pending interrupt request. This is the “acknowledge” operation that removes the IRQ from pending state.
 - Should be set when the processor finishes interrupt service by the `gm_PrcIRQSecSet()` function.
- **Bit 3 (PRC_IDL_ENB_BIT): Idle Waiting Enable**
 - When set, enables the processor to enter a wait state upon writing to `ADDR_REG_0`.
 - Facilitates power-saving modes and efficient processor management.
- **Bit 4 (PRC_MTX_RLS_BIT): Mutex Release**
 - When set, releases the GeMRTOS controller mutex previously granted to the processor.
 - Allows other processors to acquire the mutex.
- **Bit 5 (PRC_WATCHDOG_ENB_BIT): Watchdog Enable**
 - OR-set: when written as 1, arms this processor’s watchdog countdown (does not itself set the timeout value – see `CTRL_WATCHDOG_BIT`, §4.5).
 - Cleared via `ADDR_REG_2` bit 5 (`gm_PrcWatchdogDisable()`), not by writing 0 here – matches the same OR-set/AND-clear shape as `PRC_IDL_ENB_BIT`/`ADDR_REG_2` bit 0.
 - `gm_PrcWatchdogEnable()` sets this bit.



Figure 4.5: ADDR_REG_1 Register

Functionality: The `ADDR_REG_1` register provides essential control mechanisms for:

1. **Interrupt Management:** Enabling/disabling interrupts and acknowledging interrupt requests.
2. **Mutex Control:** Requesting and releasing the GeMRTOS controller mutex for inter-processor synchronization.
3. **Power Management:** Controlling processor idle states for energy efficiency.

The `ADDR_REG_1` register plays a vital role in the GeMRTOS system by providing a centralized mechanism for controlling key aspects of processor behavior.

4.3.3 ADDR_REG_2 Register

The `ADDR_REG_2` register provides access to disable the processor interrupt, and to disable the per-processor watchdog, in the GeMRTOS controller. Figure 4.6 shows the `ADDR_REG_2` bit assignment.

Register Bit Assignments:

- **Bit 0 R_PRC_INT_ENB: Processor Interrupt Enable (PRC_INT_ENB)**
 - When set, disables interrupts for the processor.

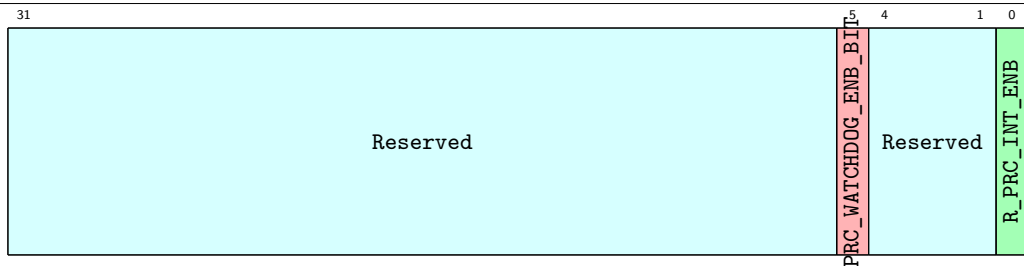


Figure 4.6: ADDR_REG_2 Register

- **Bit 5 (PRC_WATCHDOG_ENB_BIT): Watchdog Disable**

- AND-clear counterpart to ADDR_REG_1 bit 5 (§4.3.2): writing this bit as 0 clears this processor's own watchdog-enabled state, stopping the countdown without changing the configured timeout value. Writing 1 is a no-op.
- `gm_PrcWatchdogDisable()` writes this bit as 0 (all other bits written as 1, i.e. no-ops against whatever they currently hold).

In summary, the GeMRTOS Processor Region is a cornerstone of the system's architecture, providing essential functionality for multi-processor management, system customization, and debugging. This design choice enhances the overall flexibility, scalability, and ease of use of GeMRTOS in a wide variety of real-time and embedded applications.

4.4 GeMRTOS Controller Mutex Architecture

The GeMRTOS Controller implements a shared resource access control mechanism through a mutex system. This system allows the different processors in the system to coordinate access to critical resources, avoiding race conditions and ensuring data integrity.

4.4.1 Mutex Structure

The GeMRTOS Controller mutex is composed of the following key elements:

Mutex Reservation Register: This is the bit 1 of the ADDR_REG_1 register in processor memory region of the GeMRTOS controller. Setting this bit allows the processor to request ownership of the mutex.

Mutex Grant Indicator: This is the bit 1 in the processor memory region of the GeMRTOS controller, which indicates whether the current processor has the mutex granted.

Mutex Release Mechanism: When a processor has completed its critical operations, it can release the mutex by setting the bit 4 in the ADDR_REG_1 register in the processor memory region of the GeMRTOS controller.

4.4.2 Mutex Acquisition

For a processor to acquire the GeMRTOS Mutex, it must follow these steps:

1. Set the mutex reservation bit (bit 1) in the ADDR_REG_1 register in the processor memory region of the GeMRTOS controller.
2. Verify the mutex grant indicator (bit 1) in the ADDR_REG_1 register in the processor memory region of the GeMRTOS controller.
 - If the mutex is granted, the processor can proceed with its critical operations.

- If the mutex is not granted, the processor may enter a wait state (HALT or infinite loop) from which the GeMRTOS controller will wake it up when the GeMRTOS Mutex is granted or when the processor is interrupted.

The wait state is triggered by setting the bit 0 in the ADDR_REG_0 register of the processor memory region of the GeMRTOS controller. This bit activates the waitrequest signal to put the processor in a wait state, which is then cleared when the GeMRTOS Mutex is granted or when the processor's interrupt is activated.

4.4.3 Mutex Release

When a processor has completed its critical operations, it must release the GeMRTOS Mutex to allow other processors to access the shared resource:

The processor can release the mutex in two ways:

- Set the bit 4 (PRC_MTX_RLS_BIT) in the ADDR_REG_1 register of the processor memory region of the GeMRTOS controller to clear the GeMRTOS Mutex register.
- Perform a return instruction from PRC_CDE_ADDRx register address in the global memory region of the GeMRTOS controller, then to execute a new task.

In both cases, the release takes effect only while the nested flag is 0. The nested flag (R1_MTX_NESTED) is written through the STATUS_CTRL register of the GeMRTOS controller's global region: a write with bit 24 (CTRL_MTX_RLS_DSB_BIT) set stores the value of bit 0 as the flag, as `gm_MutexNestedValue()` does, and the flag reads back as bit 27 (STATUS_MTX_NESTED) of the same register. While the flag is 1 the GeMRTOS Mutex is not released; this is what allows the nesting of GeMRTOS Mutex requests (see Section 4.4.5).

This GeMRTOS Controller mutex mechanism allows processors to coordinate access to shared resources efficiently and reliably, avoiding race conditions and ensuring data integrity in multi-processor environments, even with nested mutex requests.

4.4.4 Mutex Release Decision Tree

The GeMRTOS controller implements a sophisticated 5-way multiplexer that determines which processor holds the mutex on each clock cycle. This decision tree ensures that the mutex is released only when appropriate, considering system state, processor mode, and task nesting level.

Release Decision Precedence

The hardware evaluates the following conditions in order (highest to lowest priority):

1. **Frozen Mode Release:** If system enters frozen mode AND no current holder exists, release all processors.
2. **Grant to Waiter:** If no processor currently holds the mutex, grant it to a waiting processor. The waiter is chosen by fixed priority, not by rotation: among the processors that have reserved the mutex, those already in their interrupt section are served first, otherwise those without a pending interrupt are eligible, and the lowest processor index wins.
3. **Nested Flag Gate (Critical):** If `R1_MTX_NESTED = 1`, HOLD the mutex with the current holder. This gate completely blocks release regardless of other signal states. This is the core mechanism that enforces nested critical section semantics.
4. **Kernel Mode Protection:** If processor is in kernel mode, HOLD the mutex to protect kernel operations.

5. **Return From Exception:** If ERET address is detected (return from interrupt), RELEASE the mutex unconditionally.
6. **Default (Conditional Release):** If no higher-priority condition applies, apply release bits: only processors with release-bit set will release.

Critical: The Nested Flag Gate

Condition 3 above is the most important for nested critical sections. When `R1_MTX_NESTED = 1`, the entire release logic is blocked. This means:

- Release bit has NO effect when nested flag is set
- Processor holds mutex regardless of any other signals
- Only when nested flag becomes 0 can release occur

This implements the nested lock semantics: a processor can only release the mutex when the task has exited ALL nested critical sections (nested counter reaches 0).

Register Reference

Table 4.3: Mutex Register Reference.

Register	Bits	Name	Purpose
R1_MTX_NESTED	[0]	Nested Flag	Controls release gate (0=can release, 1=hold)
R_MTX_PRC_GRN	[N-1:0]	Grant	One bit per processor; 1=holds mutex
R_MTX_RLS_BIT	[N-1:0]	Release Requests	One bit per processor; set to request release

4.4.5 R1_MTX_NESTED Synchronization Protocol

The nested flag (`R1_MTX_NESTED`) must be kept in synchronization with the current task's nesting counter (`TCB_MTX_NESTED`). This section describes the synchronization mechanism and protocol.

Two-Level Nesting Model

GeMRTOS uses two separate nesting counters at different levels:

Software Level (`TCB_MTX_NESTED`): Tracks the actual nesting depth of the current task.

- Location: Task Control Block (`GS_TCB` structure)
- Type: Unsigned integer (32-bit)
- Range: 0 to $2^{32}-1$
- Semantics: Incremented on entry to each nested critical section, decremented on exit
- Example: 0 = task not in CS, 1 = in first level, 2 = in second level, etc.

Hardware Level (R1_MTX_NESTED): Binary flag that gates mutex release.

- Location: GeMRTOS controller register
- Type: 1-bit flag (not a counter)
- Range: 0 or 1 only
- Semantics: 0 = processor can release mutex, 1 = processor must hold mutex
- Relationship: $R1_MTX_NESTED = (TCB_MTX_NESTED == 0)?0 : 1$

Synchronization Points

The nested flag must be synchronized at two specific points:

1. Point A - Mutex Acquisition (Automatic in Hardware):

- When processor acquires mutex via ADDR_REG_0 write
- Hardware automatically reads current task's TCB_MTX_NESTED
- Hardware automatically loads: $R1_MTX_NESTED \leftarrow (TCB_MTX_NESTED == 0)?0 : 1$
- Timing: Same clock cycle as grant decision
- Windows emulation must replicate this in software (see Windows Emulation chapter)

2. Point B - Nesting Level Change (Software via IOWR):

- When user task enters/exits critical section
- Software calls function: `gm_MutexNestedValue(new_nesting_value)`
- This expands to IOWR to ADDR_CTRL_STATUS with bit 24 set
- Hardware handler updates: $R1_MTX_NESTED \leftarrow value \& 0x1$
- Timing: Same clock cycle as IOWR completion

Protocol Invariant

At all times during mutex ownership, the following invariant **MUST** hold:

$$R1_MTX_NESTED == ((TCB_MTX_NESTED == 0)?0 : 1) \quad (4.1)$$

If this invariant is violated, mutex behavior becomes unpredictable and deadlocks can occur.

Protocol Sequence Example

1. Task calls `gm_GeMRTOSCriticalSectionEnter()` (level 1)
2. Writes to ADDR_REG_0 to request mutex
3. Hardware grants mutex and auto-syncs: $R1_MTX_NESTED \leftarrow 0$ (since TCB_MTX_NESTED still 0)

4. Task increments TCB_MTX_NESTED: $0 \rightarrow 1$
5. Task calls `gm_MutexNestedValue(1)` → IOWR updates R1_MTX_NESTED: $0 \rightarrow 1$
6. Task calls `gm_GeMRTOSCriticalSectionEnter()` (level 2 - nested)
7. No ADDR_REG_0 write (already inside CS), but TCB_MTX_NESTED incremented: $1 \rightarrow 2$
8. Task calls `gm_MutexNestedValue(2)` → IOWR updates R1_MTX_NESTED: $1 \rightarrow 1$ (stays 1)
9. Task calls `gm_GeMRTOSCriticalSectionExit()` (exit level 2)
10. Task decrements TCB_MTX_NESTED: $2 \rightarrow 1$
11. Task calls `gm_MutexNestedValue(1)` → IOWR: R1_MTX_NESTED stays 1
12. Task calls `GRTOS_MTX_RLS` (release request) → Hardware checks R1_MTX_NESTED=1 → HOLDS
13. Task calls `gm_GeMRTOSCriticalSectionExit()` (exit level 1)
14. Task decrements TCB_MTX_NESTED: $1 \rightarrow 0$
15. Task calls `gm_MutexNestedValue(0)` → IOWR: R1_MTX_NESTED: $1 \rightarrow 0$
16. Task calls `GRTOS_MTX_RLS` (release request) → Hardware checks R1_MTX_NESTED=0 → RELEASES

4.4.6 Nested Mutual Exclusion Terminology

To avoid confusion, this manual carefully distinguishes between two related but different concepts:

TCB_MTX_NESTED: The **software nesting counter** maintained by the kernel.

- Stored in each Task Control Block (TCB)
- Tracks how many nested critical sections the task has entered
- Type: Unsigned integer (can be 0, 1, 2, 3, ..., N)
- Updated by: Kernel when task enters/exits critical sections
- API: Incremented by `gm_GeMRTOSCriticalSectionEnter()`
- API: Decrement by `gm_GeMRTOSCriticalSectionExit()`
- Example values:
 - 0 = task not in any critical section
 - 1 = task in first-level critical section
 - 2 = task in second-level (nested within first-level) critical section
 - N = task in Nth-level nested critical section

R1_MTX_NESTED: The **hardware nested flag** in the GeMRTOS controller.

- Stored in GeMRTOS controller hardware (VHDL register)

- Binary gate that controls whether mutex can be released
- Type: Single bit (0 or 1 only - NOT a counter)
- Updated by: Hardware (automatic on acquire) or software (via IOWR)
- Read by: Release decision logic (the 5-way mux)
- Semantics:
 - 0 = processor CAN release the mutex
 - 1 = processor MUST HOLD the mutex
- Example values:
 - 0 = gate open (release permitted)
 - 1 = gate closed (release blocked)

Relationship Between the Two

The nested flag (R1_MTX_NESTED) is derived from the nesting counter (TCB_MTX_NESTED) via a simple conversion:

$$R1_MTX_NESTED = \begin{cases} 0 & \text{if } TCB_MTX_NESTED == 0 \\ 1 & \text{if } TCB_MTX_NESTED > 0 \end{cases} \quad (4.2)$$

This conversion compresses the full nesting depth (0, 1, 2, ..., N) into a binary gate signal (0 or 1). The hardware only needs to know “is there a nesting?” not “how deeply nested?”

Why Two Levels?

- **Software level** provides fine-grained control for complex task logic
- **Hardware level** provides simple, fast decision-making for release logic
- **Synchronization** between them ensures correctness without coupling hardware and software too tightly
- **Separation** allows hardware to be simple (one bit gate) while software is flexible (can track arbitrary nesting depth)

4.5 GeMRTOS Watchdog Architecture

The GeMRTOS Controller implements a per-processor hardware watchdog: a processor that stops halting – a hung task, by design – is reset by the controller without perturbing any other processor still running correctly. This is a narrower guarantee than it may first appear: the watchdog targets a hung task specifically, not a kernel or mutex problem. A legitimate firing structurally cannot happen while the processor holds the GeMRTOS Mutex (§4.4.4) – the watchdog counter does not decrement in that state. If that assumption is ever violated, that is itself a kernel defect, not the condition this watchdog exists to catch; the kernel’s recovery code halts loudly with diagnostics rather than attempting a cleanup whose safety guarantee no longer holds. A separate, harder mutex-hold-duration watchdog for that different failure class is deliberately out of scope here, left as future work.

4.5.1 Timeout and Arming

Arming a processor's watchdog requires two independent register writes – neither alone is sufficient:

1. **Set the timeout:** `gm_WatchdogTimeoutSet(timeout)` writes `CTRL_WATCHDOG_BIT` in the Control Register (§4.2.4) with the timeout value. Setting the timeout alone does not start the countdown.
2. **Enable the countdown:** `gm_PrcWatchdogEnable()` sets `PRC_WATCHDOG_ENB_BIT` in `ADDR_REG_1` (§4.3.2) for the calling processor. This is what actually arms the watchdog, using whatever timeout was last configured.

`gm_PrcWatchdogDisable()` stops the countdown (via `ADDR_REG_2` bit 5) without clearing the configured timeout value – re-enabling later reuses the same timeout unless `gm_WatchdogTimeoutSet()` is called again.

4.5.2 The Halt/Run Pet Mechanism

There is no explicit software “pet” call. The watchdog counter for an enabled processor reloads to the full configured timeout continuously while that processor is halted (`waitrequest` asserted, §4.3.2's Bit 3 wait mechanism) or while its watchdog is disabled, and only counts down while the processor is both enabled *and* actively running. In practice, this means: a processor correctly cycling between halt (nothing to do) and run (dispatching tasks) never accumulates count toward reset, since every return to halt reloads the timer. A task that spins forever without ever yielding – never letting the scheduler return this processor to halt – is exactly the condition that accumulates count toward the reset, matching the watchdog's intended purpose.

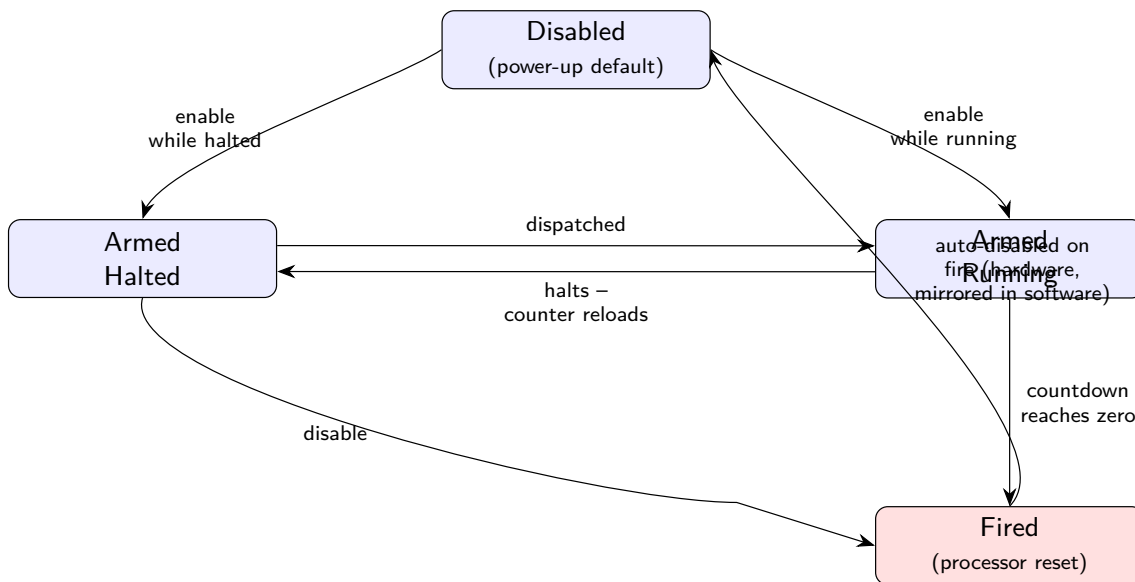


Figure 4.7: Per-processor watchdog counter states. Halting or disabling reloads the countdown; only continuous enabled, running time counts down toward a reset.

4.5.3 Recovery on Reset

This mechanism is supported on every processor, including processor 1. The watchdog's reset affects only the target processor's core (program counter and registers); shared memory – including the Kernel Control Block used by every processor – is untouched, so the rest of the system continues running undisturbed. Three properties of the design make this true, and all three are required:

- **The reset is a handshake, not a pulse.** The GeMRTOS Controller raises the target processor's reset-request interface and holds the request asserted until that processor acknowledges it, rather than driving

its global reset input. This is the mechanism both Nios II and Nios V provide for resetting one processor without disturbing the rest of the system: the processor completes any outstanding bus transaction, flushes its pipeline and then resets, and the vendor documentation is explicit that releasing the request before the acknowledge leaves the core in an undefined state. The controller applies a bounded timeout to the handshake, so a processor held under debugger control – which by design does not answer a reset request – cannot stall the request indefinitely.

- **Processor 1's reset vector is redirected.** Processor 1 normally boots at the standard reset vector, whose start-up code unconditionally clears shared memory before any GeMRTOS code runs. On a watchdog reset that would destroy the Kernel Control Block every other processor depends on. The `avalon_monitor` therefore redirects processor 1's reset-vector fetch to the same boot entry point processors 2 through N already use, which does not perform that clearing. Processors 2 through N need no redirection; they use that entry point natively.
- **A recovering processor answers pending inter-processor interrupts first.** A processor returning from a watchdog reset re-enters the kernel through its start-up path rather than through the interrupt handler. If another processor was waiting for it to acknowledge an inter-processor interrupt – a wait held under the GeMRTOS Mutex – the recovering processor acknowledges that request before it contends for the mutex, so the waiting processor is released instead of the two blocking on each other.

On the next boot entry, the kernel distinguishes a genuine first join from a watchdog recovery and, if it is a recovery, restores its own invariants before resuming normal operation:

- If the abandoned task was the processor's own IDLE task, nothing needs cleanup – its linkage was never disturbed by the reset.
- If the abandoned task was a real, user-created task, it is unlinked and returned to the task pool – the same cleanup an ordinary task kill already performs.
- An application may register a handler (`gu_WatchdogHandlerSet()`) that is invoked *after* this cleanup completes, with a read-only snapshot describing what was abandoned (which processor, which task, whether it was IDLE). The handler chooses to resume normally or treat the event as fatal; if no handler is registered, the system resumes normally by default – unchanged from before this feature existed.

This recovery logic is identical, unmodified source for every processor, for both the embedded target and the Windows development/testing simulation – only the mechanism that actually *triggers* a reset differs, since only the real hardware has an independent counter to provide it. The Windows simulation models the trigger with a dedicated monitor thread instead, useful for fast development iteration before verifying on real hardware; see the GeMRTOS developer reference (`gemrtos.md` §14) for the full platform comparison, and Chapter 14 for the Windows-specific implementation.

Important: a system generated before the reset-request handshake was introduced drives the processors' global reset inputs instead, and its processor 1 reset vector is not redirected. Regenerate the system in Platform Designer and rebuild the FPGA image to use the watchdog on processor 1.

4.6 Time Management in the GeMRTOS Architecture

Time management within the GeMRTOS architecture is crucial for enabling real-time operation and ensuring timely execution of tasks. The system relies on a combination of hardware and software components to track time, and handle timed events with precision. This involves a timer-tickless approach, reducing overhead and improving responsiveness to critical system events. The foundation of time management is the precise control and configuration of hardware timers, allowing for granular control over time intervals.

Central to the time management strategy is the utilization of dedicated hardware registers and counters within the

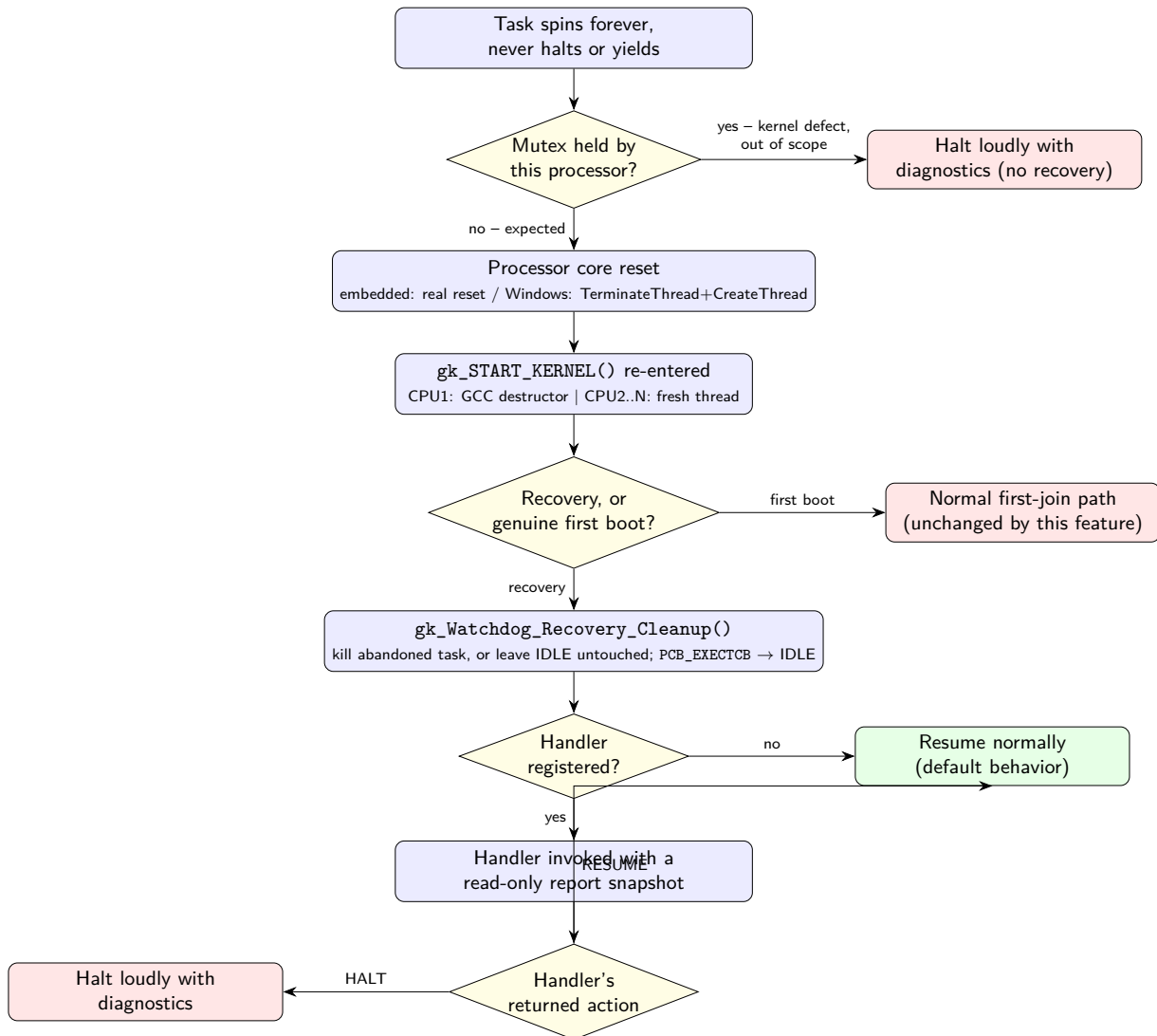


Figure 4.8: Watchdog recovery sequence, from a hung task through the optional application handler. Every step from the processor reset onward is identical, unmodified source for every processor on every platform; only the trigger (top box) differs.

controller. A time prescaler allows for scaling the system clock to provide flexible time units. Up-counters track elapsed time, triggering time events upon reaching predefined values stored in specific registers. The architecture also supports "frozen mode," which provides a mechanism for handling situations where event processing lags behind real-time, allowing the system to prioritize critical operations without losing track of overall timing.

The GeMRTOS architecture utilizes specific registers to manage time events:

- **R_TM_CNT** register: Increments when the system is in non-frozen mode, providing a continuously updating system time.
- **R_NXT_EVN_CNT** register: Holds the time of the next timed event, and the hardware compares these registers to determine event occurrences.
- **R_FRZ_TM_THR** register: Defines a threshold for maximum time lag between an event occurrence and its processing; if this threshold is exceeded, the system enters in the frozen mode to maintain timing integrity.

Events within the GeMRTOS controller are prioritized using both timed events, configured to occur at specific times, and device interrupt requests (DIRQs), triggered by external hardware. Timed events occur when the system time counter reaches the value stored in the **R_NXT_EVN_CNT** register, signaling the need for time-sensitive task execution.

The GeMRTOS controller manages two primary timekeeping modes:

- **Non-frozen Mode:** Represents the cumulative time the system operates under normal conditions, with events being processed in a timely manner. GeMRTOS typically resides in this mode.
- **Frozen Mode:** The GeMRTOS controller transitions to frozen mode when timed events cannot be processed promptly, and the next timed event's occurrence time falls behind a predefined "frozen threshold," indicating a backlog.

The system's total elapsed time is the sum of time spent in both non-frozen and frozen modes. A timed event triggers when the next scheduled occurrence time is greater than or equal to the current system time. At that point, if a specific processor is designated for handling the event, that processor is interrupted; otherwise, the interrupt is issued to the lowest-priority processor to ensure event processing.

By combining hardware and software elements, and two primary timekeeping modes, the controller is an optimized and powerful system.

The following section will discuss the GeMRTOS controller device interrupt management, so you will have even better performance.

4.7 GeMRTOS Controller Device Interrupt Management

Device interrupt management is a fundamental aspect of the GeMRTOS architecture, enabling the system to respond efficiently to external events triggered by peripheral devices. This mechanism enables the system to handle asynchronous events in a timely manner without continuously polling devices, optimizing processor utilization and improving overall system responsiveness. The device interrupt management strategy integrates both hardware and software components to ensure reliable and prioritized handling of interrupt requests.

The GeMRTOS controller incorporates an interrupt input module to receive interrupt requests from various peripheral devices. These requests, denoted as Device Interrupt Requests (DIRQs), signal the need for immediate attention from the system processors. Each DIRQ can be individually enabled or disabled, allowing the system to selectively respond to specific device events based on system configuration and operational requirements.

The next actions taken on an interrupt request are:

- The controller prioritizes and manages the request, ultimately generating a Processor Interrupt Request (PIRQ) to one or more of the system's processors.
- The controller employs specific registers, such as the R_IRQ_ENB register, to enable or disable individual DIRQs and the R_IRQ_PND register to track pending interrupt requests.
- The interrupt handler section, executed by the designated processor, processes the interrupt event and takes appropriate action by releasing the associated trigger resource.

To prevent race conditions and ensure data consistency, interrupt handlers operate under critical sections, managed by the GeMRTOS Mutex, to protect shared resources.

This combination of hardware and software ensures predictable and reliable handling of device interrupts within the GeMRTOS environment. The GeMRTOS controller manages the peripheral interrupt requests by selecting the processor most adequate to serve the request from a peripheral. When a peripheral issues an interrupt request, the lowest priority processor interrupt request is asserted.

4.8 Conclusions

The GeMRTOS controller is the central element in the GeMRTOS framework, providing a foundation for real-time, multiprocessor operations. Through its efficient management of time, external events, and processor synchronization, the controller enables consistent and predictable system behavior, maximizing system capabilities for complex embedded applications.

This chapter detailed the core architectural components of the controller. These components, including specialized registers and memory regions, work in a collaborative approach. By providing a centralized point of control for these critical functions, GeMRTOS ensures robust and reliable system operation in demanding environments.

These elements also collaborate for enabling a flexible, low latency device interrupt and management tasks. The adaptable management architecture facilitates precise control over interrupt handling and processor resources to deliver well controlled operation.

GeMRTOS: An Event-Driven RTOS

5.1 Introduction

GeMRTOS is an innovative RTOS built on an event-driven architecture. This design approach enhances adaptability in decision-making and optimizes real-time performance.

Key Features:

1. Event-driven architecture
2. Real-time performance with low latency
3. Efficient and precise event response
4. Customizable and scalable project structure

System Status Management: GeMRTOS maintains system status through interconnected control blocks. The status is updated when:

- Data within control blocks changes
- Links between control blocks are modified
- System events occur

Event Types:

- Timed events
- External events
- Internal events

Efficiency Mechanism: The event-driven design ensures that system activity is triggered only when necessary, resulting in:

- Reduced overhead
- Improved responsiveness

System Status Updates: When an event occurs, GeMRTOS updates only the relevant parts of the system status. This approach significantly reduces operating system overhead, limiting it to the actions required for maintaining an up-to-date system status.

In the following sections, we will explore:

- Time management mechanisms
- Event handling procedures
- Detailed features of the GeMRTOS architecture

These explanations will provide a comprehensive overview of GeMRTOS's capabilities in managing both time and events within a real-time operating environment.

5.2 Timed Events

Timed events are an important component of GeMRTOS's real-time operation. They are generated by the GeMRTOS controller when the system time register reaches or exceeds the value stored in the next occurrence time register.

5.2.1 Interrupt Handling

When a timed event occurs:

- The GeMRTOS controller issues an interrupt request to:
 - The processor executing the associated task (if active)
 - The lowest priority processor (if no associated task is running)
- The interrupted processor executes the GeMRTOS interrupt routine

5.2.2 Event Processing

1. The GeMRTOS controller returns EVN_CODE_TIMED as the event code
2. The `gk_TimeCallback` function is called with the associated Event Control Block (ECB) as a reference argument

5.2.3 System Behavior

The action specified in `gk_TimeCallback` for each timed event type defines the system's behavior. For example, when a task is suspended using `gu_TaskDelay`, a timed event of type `G_ECBType_OSTimeDly` is created.

5.2.4 Example: Delay Event Processing

When a `G_ECBType_OSTimeDly` timed event occurs, the following code in `gk_TimeCallback` is executed:

```
1 GS_TCB ptcb = pevent->ECB_AssocTCB;
2 switch (pevent->ECBType) {
3     case G_ECBType_OSTimeDly:  / It is a Delay Event */
4         gk_TCBWL_Unlink(ptcb);
5         gk_TCBRDYL_Link(ptcb);
6         gk_ECBFL_Link((GS_ECB *) pevent);
7         break;
8     ...
```

This code performs the following actions:

1. Unlinks the associated task from the waiting task list
2. Links the task to the ready list
3. Links the ECB control block to the free ECB list

5.2.5 Flexibility and Extensibility

The timed event mechanism in GeMRTOS is highly flexible:

- New timed event types can be defined

- Actions for new event types can be added to `gk_TimeCallback`
- `gk_TimeCallback` includes behavior for timeout events associated with various system resources (semaphores, mutexes, queues, etc.)

This design allows for easy expansion of the system's capabilities and adaptation to diverse real-time requirements.

5.2.6 Frozen Mode

Timed-event processing normally keeps pace with the system's real-time clock. If a backlog builds up – the system time advances past a missed event's occurrence time by more than a configurable threshold – the GeMRTOS controller can switch from **Non-Frozen Mode** into **Frozen Mode** to let the backlog be absorbed without losing timing consistency. Figure 5.1 shows the two modes and the events that move the system between them.

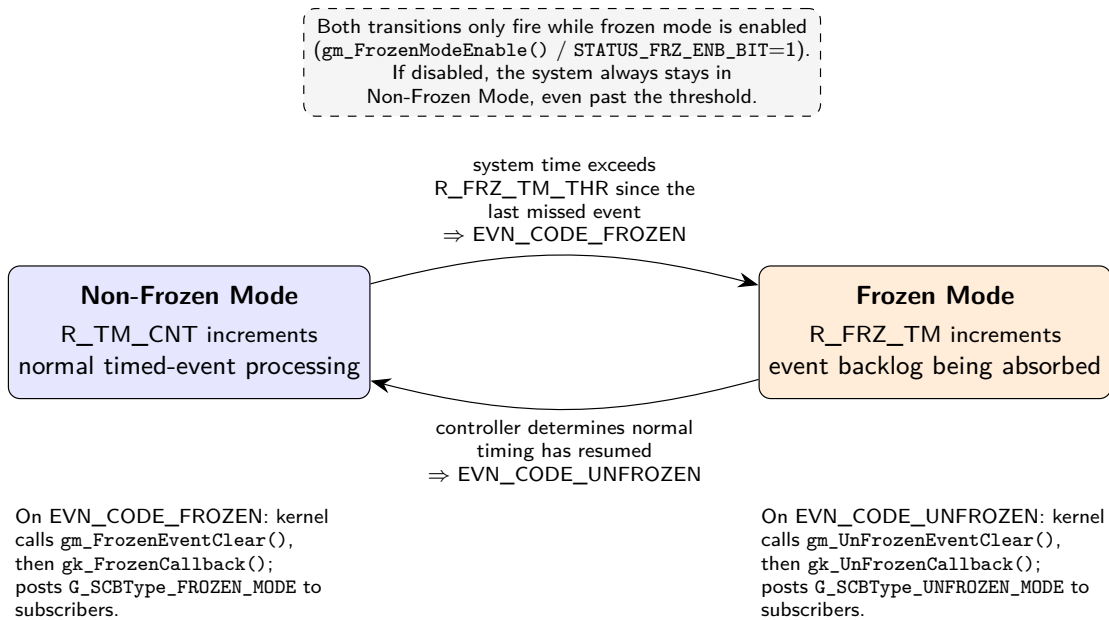


Figure 5.1: Frozen mode state diagram.

Both transitions are hardware-driven: the GeMRTOS controller raises an interrupt (`EVN_CODE_FROZEN` or `EVN_CODE_UNFROZEN`) that the kernel acknowledges before invoking the corresponding user-configurable callback, `gk_FrozenCallback` or `gk_UnFrozenCallback` (see later in this chapter for what a task can do inside them). A task that needs to react to the mode change directly, rather than through the kernel callback, may subscribe to the `G_SCBType_FROZEN_MODE`/`G_SCBType_UNFROZEN_MODE` signals that the kernel posts on each transition.

Frozen mode is opt-in: while it is disabled (`gm_FrozenModeDisable()`, the default unless the application calls `gm_FrozenModeEnable()`), the system remains in Non-Frozen Mode indefinitely, even if the backlog threshold is exceeded. The threshold itself is configured with `gu_FrozenModeThresholdSet()` and read back with `gu_FrozenModeThresholdGet()`; accumulated frozen-mode time is available via `gu_FrozenModeTimeGet()`.

5.3 External Events

External events are asynchronous signals triggered by activity in the hardware attached to the system, which requires processing by the RTOS to enable appropriate behavior in a flexible manner. The GeMRTOS controller interrupts the lowest priority processor when an enabled external interrupt is asserted. The interrupted processor

executes the interrupt routine of the GeMRTOS kernel where the associated ID of the interrupt is returned and then the trigger resource associated with the external event is released, to signal the application.

Each external interrupt request is associated with a trigger resource that is released when the external interrupt is asserted. Any task that needs to be executed when the external interrupt is asserted should be registered to the corresponding trigger resource by executing `gu_TriggerRegisterTask` function with the following arguments:

- `ptcb`: pointer to the TCB of the task to associate with the external event
- `ptrigger`: handle of the trigger resource associated with the external event (returned by `gu_TriggerCreate`)

When the task is associated with the external event, the code of the task should contain the `gu_TriggerWait` function to wait for the external event to take place. The `gu_TriggerWait` system primitive suspends the execution of the task until the external event releases the trigger associated.

5.4 Internal Events in GeMRTOS

Internal events, unlike external events, are triggered by the execution of code within the system. These events play a crucial role in managing system resources and task execution.

Characteristics of Internal Events

- Launched by code execution rather than external hardware signals
- Used to update system status and manage task execution
- Often associated with system primitives and linked list operations

Examples of Internal Events

1. **Task Execution Event:** When a task is unlinked from the running linked list, an internal event may be created to allow the execution of a ready task.
2. **Resource Management Event:** Unlinking an event control block from a resource-granting queue may trigger the activation of another event control block waiting for that resource.
3. **System Primitive Execution:** Many system primitives are associated with internal events to ensure proper updating of the system status.

Internal events are crucial for maintaining the continuous workflow of the system. When an internal event is raised, it signals the system to proceed with the next appropriate action, ensuring efficient resource utilization and task management.

5.5 GeMRTOS Linked List Management

GeMRTOS maintains system state through a set of linked lists that connect control blocks. Linking and unlinking operations on these lists simultaneously update system state, generating the internal events that drive the event-driven architecture described in Sections 5.4 and 5.3.

The complete structure and purpose of each list — including the ECB Time List (ECBTL), the RCB Granted and Waiting Event Lists (RCBGEL, RCBWEL), the TCB Running and Ready Lists (TCBRUNL, TCBRDYL), and the Signal Association Lists (ECBASL, KCBASL, RCBASL, TCBASL, TCBPSL) — is defined in Section 3.5 of Chapter 3.

GeMRTOS implements both single-linked and double-linked list variants; the choice depends on whether reverse traversal is needed. All list manipulation functions are executed exclusively while the GeMRTOS Mutex is held, guaranteeing atomic state transitions.

5.6 Callback Functions

In GeMRTOS, operations on linked list data structures often necessitate updates to the overall system status. To maintain system coherence and ensure real-time responsiveness, GeMRTOS employs a sophisticated callback mechanism. These callback functions are strategically designed to orchestrate various event-driven actions, allowing the system to dynamically adapt to changes while preserving a consistent and accurate system state during runtime.

The callback system serves multiple crucial purposes:

- It enables efficient propagation of state changes across relevant system components.
- It ensures that all interdependent parts of the system remain synchronized.
- It facilitates the execution of necessary follow-up actions in response to list modifications.
- It helps maintain the integrity of resource allocation and task scheduling information.

By organizing these callbacks in a structured manner, GeMRTOS can swiftly respond to events, manage resources effectively, and maintain optimal performance, all while guaranteeing the coherence and reliability essential for real-time operations. This approach significantly contributes to the robustness and predictability of the operating system, making it well-suited for complex, time-critical applications.

5.6.1 `gk_ResourceECBKilCallback` function

The `gk_ResourceECBKilCallback` function is executed each time an ECB must be removed from a resource's granted or waiting list.

Consider this scenario: When a task is aborted, any associated event control blocks (ECBs) linked to granting or waiting lists must be unlinked. This action must be consistent to maintain system state.

When the `gk_ResourceECBKilCallback` function is called, performs these actions to preserve consistency. For example, consider when the event concerns with a Mutex (`G_ECBType_SEM_GRANTED` type) and that is killed, then the following procedure is performed: The Mutex available count is incremented, the Mutex recurrence is reset (because a granting event is going to be removed), the ECB is unlinked from the RCBGEL, and the highest-priority task waiting for the Mutex is resumed.

5.6.2 `gk_TimeCallback` function

The `gk_TimeCallback` function is associated with timed events. When a timed event occurs, this function is automatically invoked from the GeMRTOS interrupt handler. Within `gk_TimeCallback`, the system status is updated based on the specific timed event type.

When a timed event type is defined, then the action should be implemented in the `gk_TimeCallback` function in order to support it. The `gk_TimeCallback` function is executed in kernel mode with the executing processor granting the GeMRTOS Mutex.

For instance, if a task is suspended for a specified duration using the `gu_TaskDelay` function, a timed event of type `G_ECBType_OSTimeDly` is created to signal when the interval expires. When the `G_ECBType_OSTimeDly` timed event occurs, the associated code within the `gk_TimeCallback` is executed.

5.6.3 `gk_FrozenCallback`

The `gk_FrozenCallback` function is executed when the system enters frozen mode, provided frozen mode is enabled. This routine is executed in kernel mode by the executing processor, which also holds the GeMRTOS

Mutex.

The `gk_FrozenCallback` function must be modified to define the runtime strategy applied during frozen mode.

The `gk_FrozenCallback` function may be configured to execute code during system transitions to frozen mode. Because it is a kernel signal, rather than a task signal, it is executed by the processor holding the GeMRTOS Mutex.

This mechanism provides a powerful tool for developers to implement tailored responses to system overload conditions, ensuring a graceful degradation of performance rather than catastrophic failure. It could also include a function that executes with priority and stops tasks that are not required.

5.6.4 `gk_UnFrozenCallback`

The `gk_UnFrozenCallback` function is executed when the system enters non-frozen mode, assuming that the frozen mode is enabled. This routine is executed in kernel mode by the executing processor, which also holds the GeMRTOS Mutex.

This function must be modified for defining the runtime strategy in non-frozen mode. The `gk_UnFrozenCallback` signal may also be configured to execute code when the system switches to frozen mode. Because this is a kernel signal and not a task signal, it is executed by the processor granting the GeMRTOS Mutex.

5.6.5 `gk_TaskPrioritySetCallback`

The `gk_TaskPrioritySetCallback` function is invoked each time a task is linked to either the running list or the ready list. This provides a flexible mechanism for dynamically adjusting task priorities based on system conditions or application requirements.

The priority used to link the task may be defined dynamically in order to implement different priority disciplines. For instance, the EDF (Earliest Deadline First) discipline is implemented by setting the task priority equal to the next absolute deadline.

5.6.6 `gk_StartTaskCallback`

The `gk_StartTaskCallback` function is called each time a task is started with the `gu_TaskStartWithOffset` system primitive.

It may define runtime parameters to define the behaviour of the task during runtime. This approach enables adaptable task configuration based on specific application needs. This allow also tasks in external code to define its behaviour.

In this way, the user can program the tasks and make his special routines on the system.

5.7 Conclusions

This chapter highlighted how GeMRTOS leverages an event-driven architecture. With the use of dedicated controller hardware, all functions are implemented in a straightforward and efficient manner.

GeMRTOS uses different kinds of Control Blocks, and thanks to this, and the linked lists the scheduling can be optimized. As long as this is done, the results are really accurate and easy to get with a high efficiency.

From its core design principles to specific implementations, GeMRTOS leverages events to enable adaptable, robust, and efficient real-time performance. By decoupling task execution from event occurrence and providing flexible mechanisms for managing event responses, GeMRTOS empowers developers to build complex and demanding embedded systems that can meet the challenges of today's applications.

GeMRTOS IP for Platform Designer

6.1 Introduction

The GeMRTOS Multiprocessor IP component simplifies the design of multiprocessor systems, offering significant advantages over traditional development methods. By reducing the complexity of processor booting, synchronization, and mutual exclusion, the IP component makes designing a multiprocessor system nearly as straightforward as creating a uniprocessor system. This streamlined approach lowers the barrier to entry for developers seeking to harness the power of parallel processing in their embedded designs.

6.2 GeMRTOS IP Architecture

The GeMRTOS IP architecture includes all the components required to implement multiprocessor systems along with other components for ease of system design and debugging. As depicted in Figure 6.1, the GeMRTOS Multiprocessor IP incorporates the following components, each designed for a specific purpose:

- **GeMRTOS Controller:** This serves as the core of the GeMRTOS Multiprocessor IP component, responsible for time management and external events. It enables the Avalon-conduit port, providing easy access to external signals.
- **System Processors:** This component supports both Nios II and Nios V soft processors, offering flexibility in processor selection.
- **Nios Monitor Component:** This serves as a debugging tool, sniffing each processor bus to detect special transaction patterns. In configurations without a debugging architecture, this component operates transparently.
- **MM-Avalon Bridge for External Devices:** This bridge facilitates the definition of different clock sources for processors and external devices, enabling efficient clock domain adaptation.
- **JTAG-UART Devices:** These devices, configurable for STDIO and STDERR consoles, streamline system design and debugging.
- **MM-Avalon Bridge for HPS Interface:** This bridge enables access to and from the HPS domain memory map. Knowledge required for use.

Overall, the GeMRTOS Multiprocessor IP component significantly reduces the challenges associated with designing complex multiprocessor systems, thereby streamlining the development process and improving efficiency.

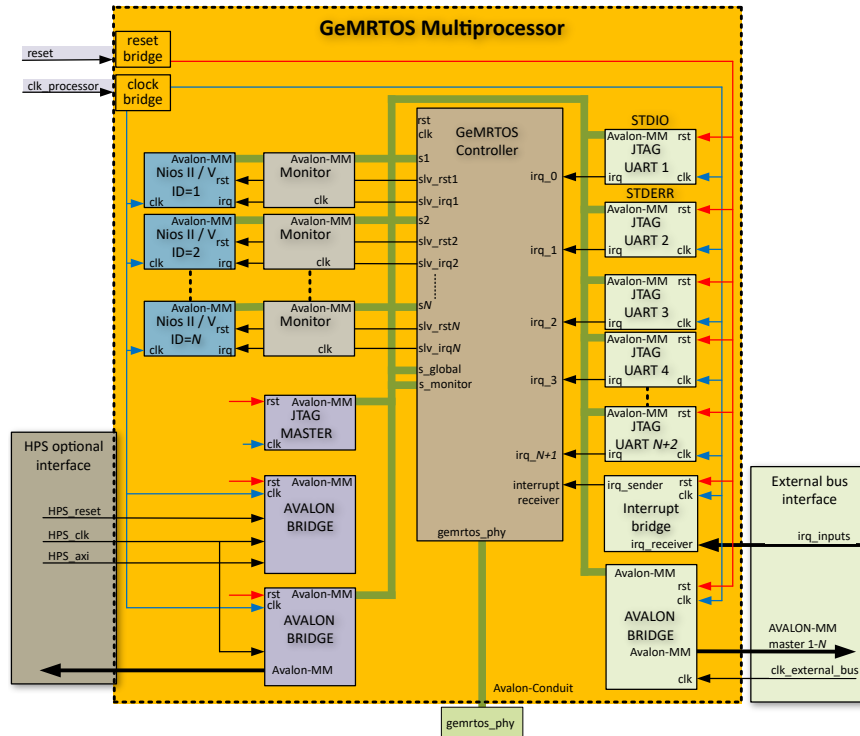


Figure 6.1: GeMRTOS Multiprocessor IP internal architecture.

6.3 GeMRTOS IP Configuration Parameters

The GeMRTOS IP component seamlessly integrates with Altera's Platform Designer development tool, appearing as the GeMRTOS_Multiprocessor component in the catalog. This integration provides a streamlined and adaptable approach to system configuration. Developers can tailor the GeMRTOS IP component by adjusting the parameters that affect the GeMRTOS Multiprocessor component's operation.

Table 6.1 summarizes the GeMRTOS IP configuration parameters, detailing their ranges and descriptions.

Table 6.1: GeMRTOS IP Configuration Parameters

Parameter	Range	Description
Number of processors	1-32	Determines the number of processors to be included in the GeMRTOS Multiprocessor component.
Processor type	Nios II/e, Nios II/f, Nios V/m	Determines the type of processor to be utilized.
Reset vector memory	System memories available	Determines the memory component in which the reset vector of the processor is configured. The parameter may be configured from a list of available options according to the systems memory the component is connected to.
Reset vector offset		Determines the offset in the reset vector memory for the reset vector of the processor.

Continued on next page

Table 6.1: GeMRTOS IP Configuration Parameters (Continued)

Parameter	Range	Description
Exception vector memory (only for Nios II)	System memories available	Determines the memory component in which the exception vector of the processor is configured. The parameter may be configured from a list of available options according to the systems memory the component is connected to.
Exception vector offset (only for Nios II)		Determines the offset in the exception vector memory for the exception vector of the processor.
Instruction cache size	None, 512Bytes, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB.	Set the instruction cache size for each processor of the system when it is supported.
Enable pipeline in CPU (Nios V/m only)	True / False	Enable the pipeline feature of the CPU when supported.
Independent external processor buses	True / False	When True, the GeMRTOS IP component produces one bus for each processor, useful to share the bandwidth load over several buses. When False, all the processors share the same bus, useful to simplify the design of the system. It is always advisable to use independent external processor buses to improve the performance of the system.
Include JTAG UART for STDIO	True / False	When True, a JTAG UART component is included in the GeMRTOS IP configuration for STDIO usage.
Include JTAG UART for STDERR	True / False	When True, a JTAG UART component is included in the GeMRTOS IP configuration for STDERR usage.
Include JTAG UART for each processor	True / False	When True, a JTAG UART component is included in the GeMRTOS IP configuration for each processor. This feature may be useful for debugging purposes or for demonstrations.
Enable HPS internal access	True / False	When True, a bridge is enable for accessing the GeMRTOS IP internal memory map from a HPS.
Quartus Prime project name	Selection from project folder	Determines the Quartus Prime project name in which the Platform Designer system is instantiated. This configuration is useful to automate the <code>gemrtos_build</code> script.
Platform Designer project name	Selection from project folder	Determines the name of the current Platform Designer project name, useful to automate the <code>gemrtos_build</code> script.

The GeMRTOS Multiprocessor IP parameters (illustrated in Figure 6.2) can be configured through the standard parameter windows of the Platform Designer. These configuration options become accessible when the GeMRTOS component is selected within the System Contents window of the Platform Designer interface.

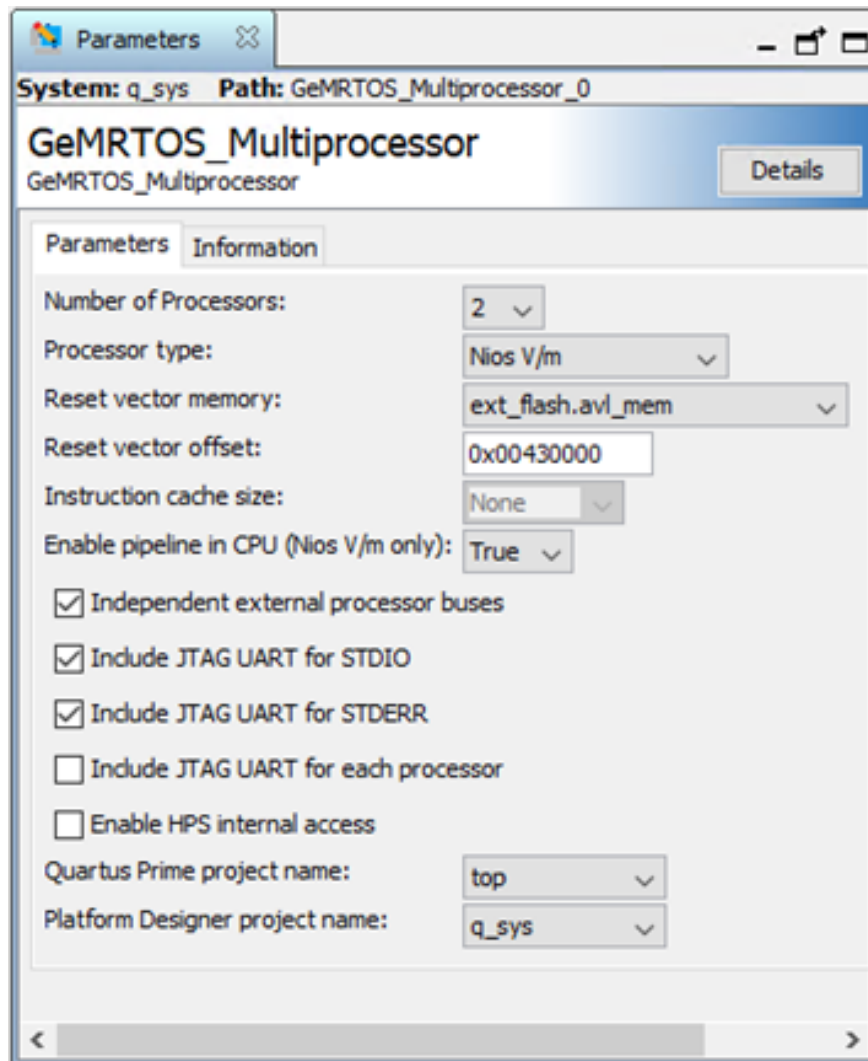


Figure 6.2: GeMRTOS IP parameter configuration tab window.

This setup allows system designers to customize the GeMRTOS Multiprocessor IP to suit their specific project requirements, integrating it effectively within the larger system architecture defined in the Platform Designer tool.

The available parameters and their possible values dynamically adapt based on two key factors:

- Connected Memory Devices: The type, number, and characteristics of memory devices connected to the GeMRTOS component influence which parameters are presented and their available options. For example:
- Different memory types (e.g., SRAM, DRAM, Flash) may expose different configuration options. The size and organization of connected memories could affect addressable ranges or caching parameters.
- Selected Processor Type: The specific processor architecture chosen for the system impacts the available features and, consequently, the relevant configuration parameters.

This adaptive parameter system ensures that:

- Users are presented with only relevant configuration options.
- The component can be optimally configured for the specific hardware environment.
- Incompatible or unavailable options are automatically hidden or disabled.

By dynamically adjusting the parameter set, the GeMRTOS configuration interface streamlines the setup process and helps prevent potential misconfigurations that could arise from presenting irrelevant options.

6.4 GeMRTOS IP Information

The GeMRTOS IP Information tab window (Figure 6.3) displays configuration information that is valuable for design and debugging purposes. It provides a quick overview of key system parameters without the need to navigate through multiple configuration windows.

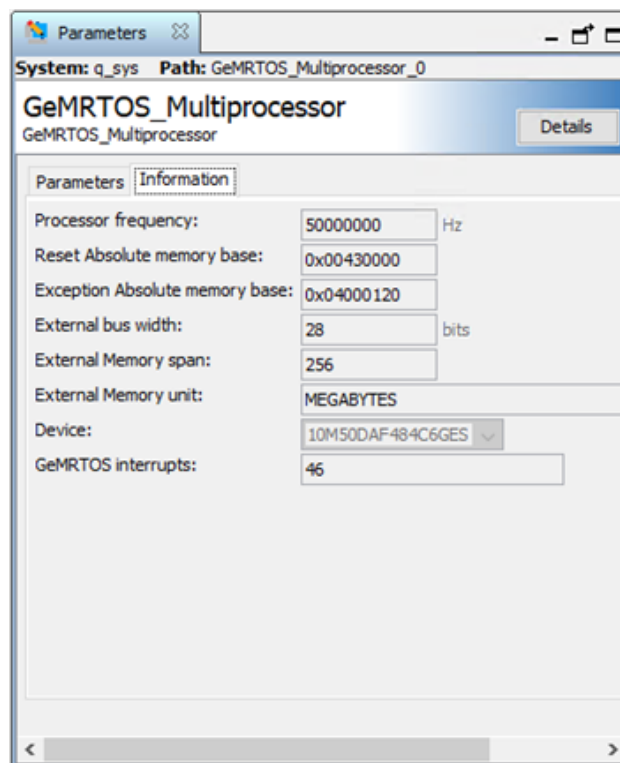


Figure 6.3: GeMRTOS IP Information tab window.

Table 6.2: GeMRTOS IP information parameters

Parameter	Description
Processor frequency	Shows the clock frequency connected to the processors. Important for timing calculations and performance estimations.
Reset absolute memory base	Displays the absolute address where processors reset. Useful when defining addresses in flash memory regions shared with FPGA configuration.
Exception absolute memory base	Shows the absolute address for processor exceptions. Also helpful for address allocation in shared flash memory regions.
External bus width	Indicates the width of the bus connecting external components. Automatically updated when components are connected to processor buses.
External Memory span	Displays the length of the external bus. Measured in units specified in the "External Memory Unit" field.
External Memory unit	Specifies the unit of measurement for the External Memory span.
Device	Shows the selected FPGA device for implementation. Crucial for understanding hardware constraints and capabilities.
GeMRTOS Interrupts	Indicates the number of interrupts configured in the GeMRTOS IP. Important for understanding the system's interrupt handling capacity.

This information tab serves as a quick reference for system designers and debuggers. It provides a snapshot of critical system parameters, helping to ensure that the configuration aligns with the intended design and assisting in troubleshooting efforts. The data presented here can be particularly useful when defining memory maps, calculating timing requirements, or verifying interrupt configurations.

By presenting this information in an easily accessible format, the GeMRTOS IP information tab streamlines the design process and helps prevent configuration errors that could arise from misunderstanding or overlooking key system parameters.

6.5 GeMRTOS IP Memory Map

The GeMRTOS IP handles memory mapping for all its internal components which are connected to the system bus. This section details the memory allocation process and the structure of the internal memory map.

Memory Map Assignment

- The GeMRTOS IP automatically assigns memory map addresses to all its internal components connected to the system bus.
- The processor with CPU_ID 1 determines the system's memory map (useful for BSP project generation).
- The memory addresses assigned by CPU_ID 1 are used by the compiler when generating the application.

Internal Memory Map Components

The internal memory map includes the following components:

Table 6.3: GeMRTOS IP internal memory map.

Component	Description
processor debug memory	Memory region dedicated to debug capabilities of the processor, if available.
gemrtos_global_region	Memory region of the global register of the GeMRTOS controller.
gemrtos_processor_region	Processor shadow memory region in the GeMRTOS controller.
jtag_uart_0	JTAG UART component for STDIO when enabled in GeMRTOS parameter window.
jtag_uart_1	JTAG UART component for STDERR when enabled in GeMRTOS parameter window.
jtag_uart_2..Processors+1	JTAG UART component for each processor when enabled in GeMRTOS parameter window.

6.5.1 Memory Map Organization

All memory-mapped components listed in Table 6.3 are placed in memory addresses after (higher than) the external components connected to the GeMRTOS IP component in the System Contents window.

This memory mapping scheme ensures:

- Each component has a dedicated address space.
- A consistent memory layout across the system.
- Proper integration of debug and communication features (e.g., JTAG UARTs).
- Correct memory location referencing by the compiler when generating code.

Understanding this memory map is crucial for developers working with GeMRTOS, as it impacts how they access various system components and debug their applications. It also highlights the system's flexibility in accommodating different configurations of processors and communication interfaces.

6.5.2 Conclusion

The GeMRTOS IP's memory mapping capabilities are a crucial component of its overall system architecture. By automatically assigning memory addresses to connected components, the GeMRTOS IP ensures a consistent and organized memory layout across the entire system.

The central role of the CPU_ID 1 processor in defining the memory map is particularly noteworthy. This design choice allows for a unified memory representation, which is essential for the compiler to correctly generate application code that can seamlessly interact with the various system resources. The memory addresses determined by CPU_ID 1 serve as the reference point for the entire software ecosystem, streamlining development and integration efforts.

The internal memory map components, such as the processor debug memory, GeMRTOS global and processor regions, and the JTAG UART interfaces, provide valuable system-level features and capabilities. By strategically placing these components in the memory hierarchy, the GeMRTOS IP ensures that they can be efficiently accessed and utilized by both hardware and software components, enhancing the overall system's functionality.

and debugging capabilities.

Ultimately, the GeMRTOS IP's memory mapping approach demonstrates a well-designed and comprehensive strategy for managing the complex interplay of hardware and software within a real-time embedded system. This level of attention to memory organization and accessibility is a testament to the GeMRTOS IP's commitment to delivering a robust and versatile platform for developers working on demanding, time-critical applications.

GeMRTOS Hardware and Software Design Flow

Note: This chapter provides a conceptual overview of the hardware and software design flow. For the complete step-by-step walkthrough — including Platform Designer connection steps, Quartus Prime compilation, BSP generation, and application build — see Chapter 8.

7.1 Introduction

GeMRTOS makes the process of designing an efficient multiprocessor system as easy as a uniprocessor one. The hardware and software design flow involves the same tools that are used when designing a uniprocessor system based on Nios II or Nios V soft-processors. This consistency in the design approach simplifies the transition from uniprocessor to multiprocessor architectures, allowing developers to leverage their existing knowledge and expertise.

The GeMRTOS IP provides a comprehensive solution that seamlessly integrates hardware and software components, enabling developers to focus on their application-specific requirements rather than the underlying complexity of a multiprocessor system. By abstracting away the low-level details, GeMRTOS empowers engineers to rapidly prototype, configure, and deploy their designs, ultimately accelerating the development lifecycle.

One of the key advantages of the GeMRTOS design flow is its ability to adapt to the specific needs of each project. Whether the system requires a single processor or multiple processors, the design process remains largely unchanged, ensuring a familiar and efficient workflow. This flexibility allows developers to scale their designs as the project requirements evolve, without the need for a complete architectural overhaul.

Furthermore, the GeMRTOS design flow leverages the well-established Nios II and Nios V development ecosystems. Developers can continue to utilize the same Quartus Prime and Altera SoC FPGA Embedded Development Suite tools they are already familiar with, minimizing the learning curve and enabling a seamless integration with existing development practices. This approach helps to preserve investments in existing toolchains and ensures a smooth transition to multiprocessor architectures.

7.2 Quartus Prime / Platform Designer Hardware Design Flow

Quartus Prime is a sophisticated software suite developed by Altera for designing, synthesizing, and verifying digital circuits targeting Altera FPGAs, SoCs, and complex programmable logic devices (CPLDs). It provides a comprehensive environment that enables designers to take a project from initial design entry to final programming of the target device.

Platform Designer, formerly known as Qsys, is a system integration tool within the Quartus Prime software. It automates the process of connecting different intellectual property (IP) components and subsystems within an FPGA or SoC design. Essentially, it handles the complex task of generating the interconnect logic needed to link these components, saving significant time and effort in the design process.

The GeMRTOS IP integrates seamlessly into Platform Designer, as it is available as any other IP component in Platform Designer. Once installed, the GeMRTOS Multiprocessor IP is easily accessible from the IP Catalog in Platform Designer, as shown in 7.1.

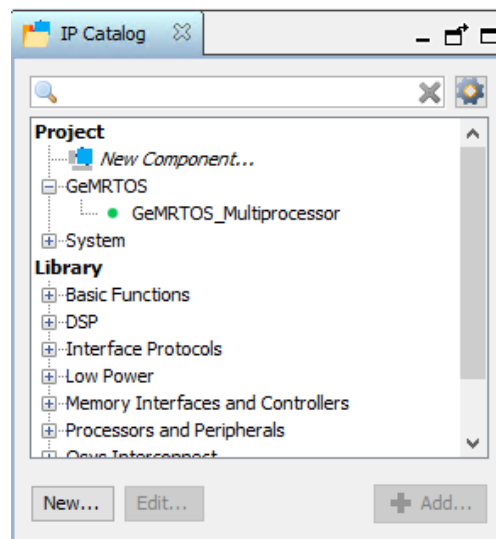


Figure 7.1: GeMRTOS IP integration in Platform Designer IP Catalog.

The GeMRTOS IP may be integrated into any project as if it were a Nios II or Nios V soft-processor by adding and connecting it in the System Contents window of Platform Designer.

Figure 7.2 shows the design flow of a project using Platform Designer and Quartus Prime.

The design flow starts by including the GeMRTOS Multiprocessor component into a Platform Designer project. A Platform Designer project is saved in a file with the extension `.qsys`. This Platform Designer project contains all the components and connections of the SOPC design.

When the SOPC is designed, it should be generated using the "Generate HDL" command in Platform Designer. The Platform Designer generation process produces the following files:

- `<sopc_system>.cmp`: a plain-text file including the declaration of the component that contains all the SOPC hardware project. This allows the SOPC to be instantiated into a Quartus Prime HDL project as a hardware component for further compilation to a targeted FPGA device. The `.cmp` file shows the input and output conduit ports defined in the Platform Designer SOPC for easy integration into the HDL project in Quartus Prime.

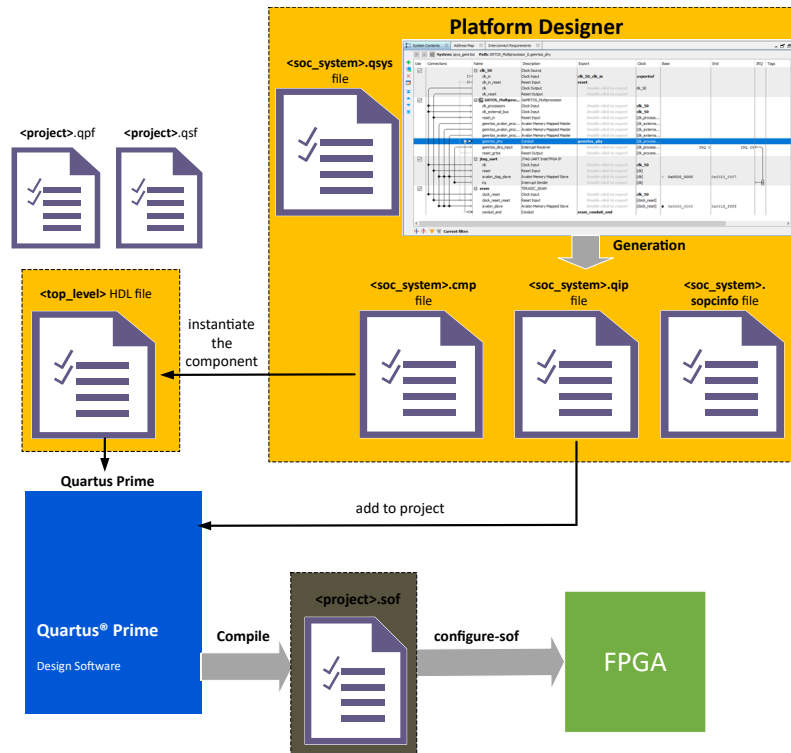


Figure 7.2: Platform Designer and Quartus Prime design flow.

- `<sopc_system>.qip`: a Quartus Prime IP project file that includes all the information about the SOPC project created in Platform Designer to be included in Quartus Prime for compilation. The `.qip` file should be included into the Quartus Prime project in order to make all the components included in the SOPC project available.
- `<sopc_system>.sopcinfo`: a file that contains all the SOPC information required to generate the software support for the SOPC system created with Platform Designer. System defines, memory map addresses, and system components are detailed in the `.sopcinfo` file for software interface generation.

As it can be seen, Platform Designer allows creating an SOPC project and also generates all the files and information required to produce both the hardware and the software components of the system.

Quartus Prime is then used to generate the FPGA configuration file from the hardware description, which should include the Platform Designer component that integrates the GeMRTOS IP for implementing the multiprocessor system. Quartus Prime is also utilized for programming the FPGA device with the configuration file produced.

7.3 The Software Design Flow

The software design flow involves several steps to create an application from the hardware specification, given in the `.sopcinfo` file, to the full compilation of the application code.

For Nios II and Nios V processors, the software creation involves two software projects:

- **BSP (Board Support Package) Project:** This project contains all the drivers and system definitions required to compile, initialize, and manage all the hardware components included in the SOPC project. For instance, when a JTAG UART component is included in the SOPC project, the initialization routines for that component are included in the system initialization, and the read and write functions are made available for the user to access the functionality of the JTAG UART device. All the information required to create the BSP project is included in the `.sopcinfo` file. Each time the Platform Designer project is modified, the BSP project should be regenerated to update its software components.
- **Application Project:** This is the software project that implements the functionality required by the application. It uses the BSP project interfaces to access the hardware devices of the system. For example, a `printf()` instruction may invoke the writing routines for the STDOUT JTAG UART of the system.

Figure 7.3 shows the different layers covered by the BSP and Application projects.

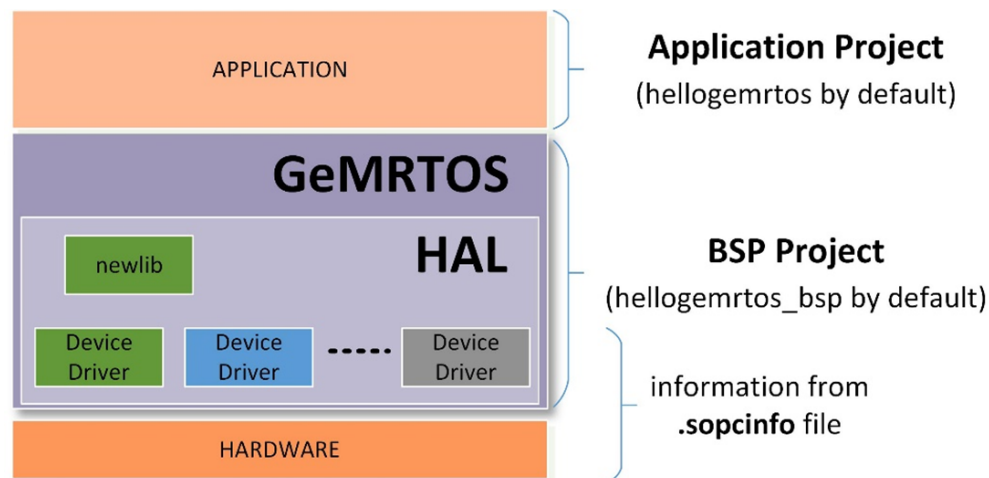


Figure 7.3: GeMRTOS software layer containing the BSP and Application projects.

The software design flow involves several steps that must be taken in the correct order.

Note: The Nios II Command Shell requires a Linux-like environment. Earlier versions of Quartus Prime utilized the Cygwin environment to execute these commands. However, Quartus Prime versions later than 22.0 now use the Windows Subsystem for Linux (WSL) command-line environment. The procedures for installing Quartus Prime version 22.0 and configuring WSL for the GeMRTOS design flow on Windows 10-64bit systems are detailed in gemrtos.com.

By following the established software design flow and utilizing the BSP and Application projects, developers can effectively leverage the GeMRTOS IP and create robust, multi-threaded applications that take full advantage of the underlying hardware architecture.

7.3.1 The `settings.bsp` file

The `settings.bsp` file is the file that contains all the configuration of the BSP project. Memory regions, device drivers, parameter settings, and the main processor are included in the `settings.bsp` file for use in compilation.

The `niosv-bsp` utility is used to create and regenerate the `settings.bsp` file. The `niosv-bsp-editor` utility

(Figure 7.4) may be used for graphical editing of the settings in the `settings.bsp` file.

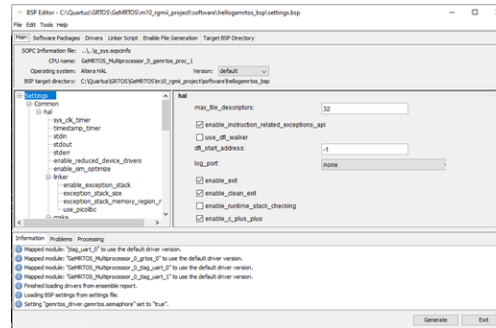


Figure 7.4: The `niosv-bsp-editor` utility for editing the `settings.bsp` file.

GeMRTOS requires some HAL (Hardware Abstraction Layer) settings for better performance that are set by default when implementing the software design flow through the `gemrtos_build` script:

- `hal.sys_clk_timer` (none): Usually, most SOPC designs define a timer component for time support in HAL. However, this feature is not required for GeMRTOS, and consequently, it should be set to none.
- `hal.timestamp_timer` (none): The timestamp timer may be set when debugging features are required. In GeMRTOS, this feature is not required, and consequently, it should be set to none.
- `hal.enable_reduced_device_drivers` (true): By setting `enable_reduced_device_drivers` to true, the device drivers avoid interrupt alternatives. In GeMRTOS, it should be convenient to avoid using HAL interrupts instead of GeMRTOS triggers.
- `enable_instruction_related_exceptions` (true): This setting allows catching the processor exceptions to avoid runtime errors.
- `hal.stdin`, `hal.stdout`, and `hal.stderr`: These settings define the JTAG UART devices for STDIN, STDOUT, and STDERR consoles, respectively. In GeMRTOS, by default, these are set to `jtag_uart_0`, `jtag_uart_0`, and `jtag_uart_1` for STDIN, STDOUT, and STDERR, respectively.

GeMRTOS generates the `gemrtos_build` script for automated software development, which is discussed in the next section.

7.4 `gemrtos_build` Script for Software Design Flow

The software design flow involves several steps for configuring the different projects involved. Figure 7.5 shows the software design flow detailing the main files involved.

The `gemrtos_build` script executes several commands to process the BSP and application projects to produce the `<application>.elf` file that contains the configuration file ready to download to the GeMRTOS multiprocessor system.

The main steps performed by the `gemrtos_build` script are:

- **Generate the BSP Project:** The script generates the BSP project based on the configuration specified in the `.sopcinfo` file and the settings in the `settings.bsp` file. This creates the necessary device drivers, system definitions, and other components required for the hardware platform.
- **Compile the BSP Project:** The script compiles the BSP project to produce the BSP library that can be used by the application project.
- **Compile the Application Project:** The script compiles the application project, which uses the BSP library to access the hardware devices and implement the desired functionality.

- **Link the Application Project:** The script links the compiled application code with the BSP library to produce the final <application>.elf file.

By running the `gemrtos_build` script, developers can streamline the software design flow and ensure that the necessary steps are performed correctly to produce the final executable file that can be downloaded to the GeMRTOS multiprocessor system.

The `gemrtos_build` script abstracts away the complex details of the software design flow, allowing developers to focus on the application-level development while ensuring that the underlying hardware-software integration is properly handled.

7.5 Conclusions

This chapter has provided a comprehensive overview of the hardware and software design flow for GeMRTOS-based multiprocessor systems. It highlighted the seamless integration of GeMRTOS IP into Altera's Platform Designer, allowing developers to leverage familiar tools and methodologies for efficient system creation. The streamlined design flow, encompassing both hardware configuration in Quartus Prime and software development using the Nios II or Nios V toolchain, ensures a cohesive and predictable development experience.

A key aspect of this flow is the automated generation of the Board Support Package (BSP), which contains the drivers and system definitions necessary for software development. The `gemrtos_build` script simplifies the software build process, automating tasks such as BSP configuration, application compilation, and FPGA programming. This reduces development time and minimizes the potential for manual configuration errors.

Finally, by adhering to a well-defined and repeatable design flow, developers can harness the full potential of GeMRTOS to create innovative and adaptable multiprocessor systems. The automation of key steps, combined with the availability of comprehensive software libraries and reference designs, empowers developers to confidently tackle complex embedded projects and deliver high-quality solutions.

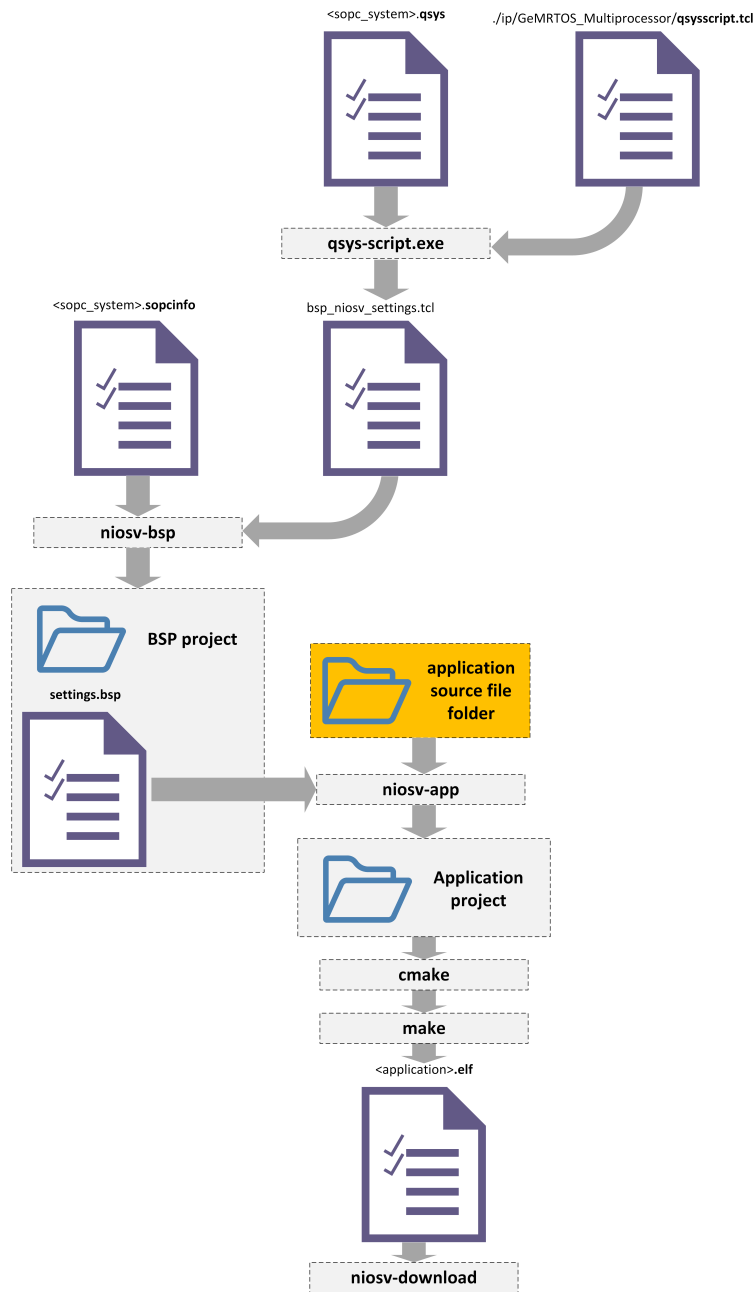


Figure 7.5: The `gemrtos_build` script software flow.

Development Guide

Note: This chapter provides the step-by-step procedure for building a GeMRTOS-based system. For a conceptual overview of the tools and files involved in the design flow, see Chapter 7.

8.1 Introduction

This chapter provides a step-by-step guide to developing systems based on GeMRTOS. The development process closely mirrors that of uniprocessor systems based on Nios II or Nios V, with necessary adaptations for the multiprocessor environment. Users familiar with the Nios II or Nios V development flow will find the GeMRTOS development process easy to learn.

Note: This chapter assumes a basic understanding of embedded systems development and FPGA concepts. Refer to the cited resources for background information if needed.

8.2 Licensing GeMRTOS

GeMRTOS consists of two separately licensed components:

- **GeMRTOS Software** — the kernel, drivers, hardware abstraction layer, and API — is distributed under the **PolyForm Noncommercial License 1.0.0**. This license permits free use, study, and modification for any noncommercial purpose, including academic research, educational projects, and personal experimentation. Commercial use or commercial distribution requires a separate written license from the GeMRTOS project. The full license text is available in the `LICENSE` file at the root of the source repository and at polyformproject.org.
- **GeMRTOS Hardware IP** — the Platform Designer IP component for Quartus® Prime — is a separately licensed proprietary component. A hardware IP license is required to use the GeMRTOS IP core in Platform Designer. This license is available at no cost for academic and research purposes and can be requested at gemrtos.com/license. Commercial use of the hardware IP requires a separate commercial agreement.

8.2.1 Hardware IP License Configuration

Once you have obtained the GeMRTOS hardware IP license file (.dat extension), configure it in Quartus Prime as follows. Quartus Prime supports both Fixed and Floating license types; select the type that matches your Quartus Prime license configuration.

Fixed License

To configure a Fixed license:

1. Navigate to *Tools* → *License Setup...* in Quartus Prime.
2. Browse to the GeMRTOS license file (.dat extension). See Figure 8.1 for an example configuration.

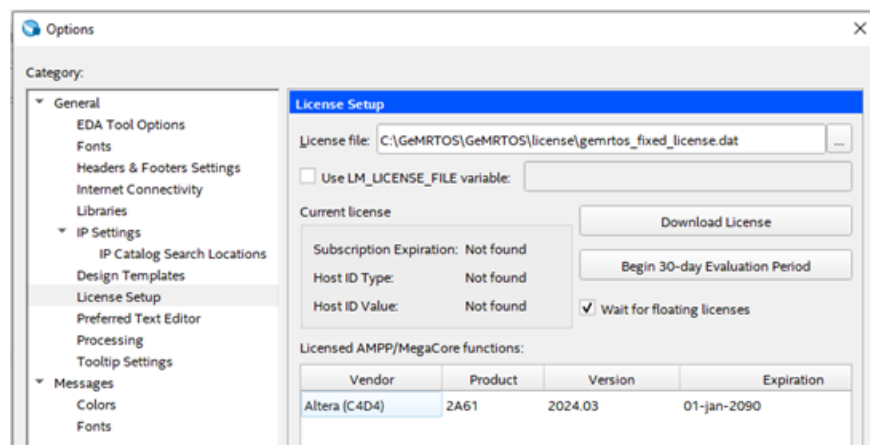


Figure 8.1: GeMRTOS Hardware IP Fixed License Setting in Quartus Prime

Important: Ensure the hardware IP license is correctly configured in Quartus Prime before synthesizing a design that includes the GeMRTOS IP component. An unconfigured or invalid license will cause compilation errors in Platform Designer.

8.3 Hardware Design Guide

8.3.1 Integrating and Designing with the GeMRTOS Multiprocessor IP Component in Platform Design Projects

Integrating the GeMRTOS IP component into Platform Designer projects is similar to creating SOPC projects for uniprocessor systems. A recommended approach involves adapting existing example projects provided with development kits. This strategy simplifies the process, as these examples typically include necessary components and pre-configured pin assignments, particularly for complex elements like DDR memory controllers. Reference designs for specific boards (e.g., DE2-115, MAX10, Bemicro SDK) can be requested from gemrtos.com.

Step 1: Create a Folder for your GeMRTOS Project

This step establishes the foundation for your project by creating a dedicated folder and copying the necessary Quartus Prime project files into it.

1. Create a new folder for your GeMRTOS project. This folder will be referred to as `<Quartus_Prime_project_folder>`.
2. Copy the base Quartus Prime project files into this folder.

The `<Quartus_Prime_project_folder>` must contain the following files for a valid Altera Platform Designer project:

- `<quartus_project_name>.qpf`: Contains project information and is opened when launching the project in Quartus Prime.
- `<quartus_project_name>.qsf`: Contains project settings, including pin assignments provided by the board manufacturer.
- `<top_level_entity_file>.v` or `<top_level_entity_file>.vhd`: Contains the top-level hardware description (Verilog or VHDL) that instantiates the embedded system created in Platform Designer, including the GeMRTOS IP component.

Note: If starting with a kit or board manufacturer's example, add the GeMRTOS IP component to the project for instantiation in Platform Designer. If using a project from gemrtos.com, proceed to Step 8.3.2.

Step 2: Installing GeMRTOS into the Project

This step integrates the GeMRTOS IP component into your project by copying the necessary files.

1. The GeMRTOS IP component is fully compatible with Altera's Platform Designer.
2. Copy the `/ip` subfolder, obtained by downloading the GeMRTOS IP package from this link, into the `<Quartus_Prime_project_folder>` folder.

Figure 8.2 illustrates the basic directory structure.

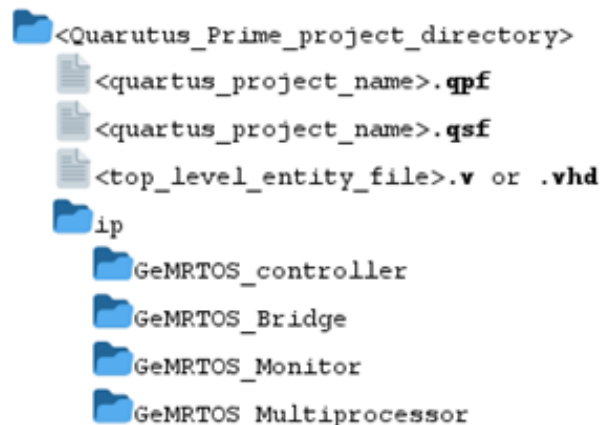


Figure 8.2: GeMRTOS Basic Project Directory Structure

Important: Verify that all necessary files are correctly located within the `<Quartus_Prime_project_folder>`. This ensures seamless integration of the GeMRTOS IP component and efficient project development.

8.3.2 Step 3: Open Quartus Prime Project

This step opens the main Quartus Prime project file to load your design's overall configuration.

1. Launch Quartus Prime.

2. Open the project file by selecting *File* → *Open Project...* as shown in Figure 8.3.

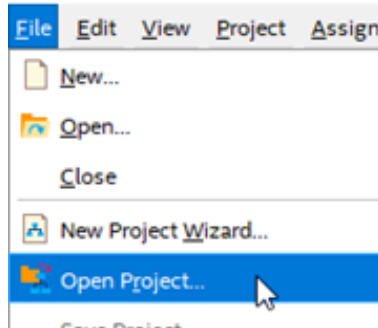


Figure 8.3: Opening Quartus Prime Project

Important: After opening the project file, verify that all design files and settings have been correctly loaded. This ensures the Quartus Prime environment is properly configured for hardware development.

8.3.3 Step 4: Open the Altera's Platform Designer Project

This step opens the Platform Designer project file, where the embedded system and GeMRTOS IP component are configured.

1. In Quartus Prime, select *Tools* → *Platform Designer* as shown in Figure 8.4.
2. In the Platform Designer GUI, open the <platform_designer_file>.qsys file located in the <Quartus_Prime_project_folder>.

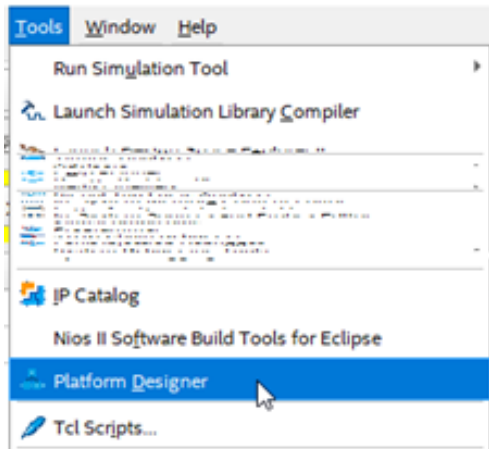


Figure 8.4: Opening Altera's Platform Designer

Note: The GeMRTOS Multiprocessor component is accessible in the *IP Catalog* tab, and the *System Contents* tab displays all elements of the GeMRTOS multiprocessor system.

Step 5: Add the GeMRTOS Multiprocessor Component

This step details how to add the GeMRTOS Multiprocessor component to your Platform Designer project and configure its essential parameters.

1. In the Platform Designer window, navigate to *Project* → *GeMRTOS* → *GeMRTOS Multiprocessor* within the *IP Catalog* tab.
2. Click the *Add* button. This will add the GeMRTOS Multiprocessor component to your system.
3. The GeMRTOS parameter configuration panel will appear, either upon adding the component or by selecting it in the *System Contents* window (see Figure 8.5). This panel allows you to configure various features of the multiprocessor architecture.

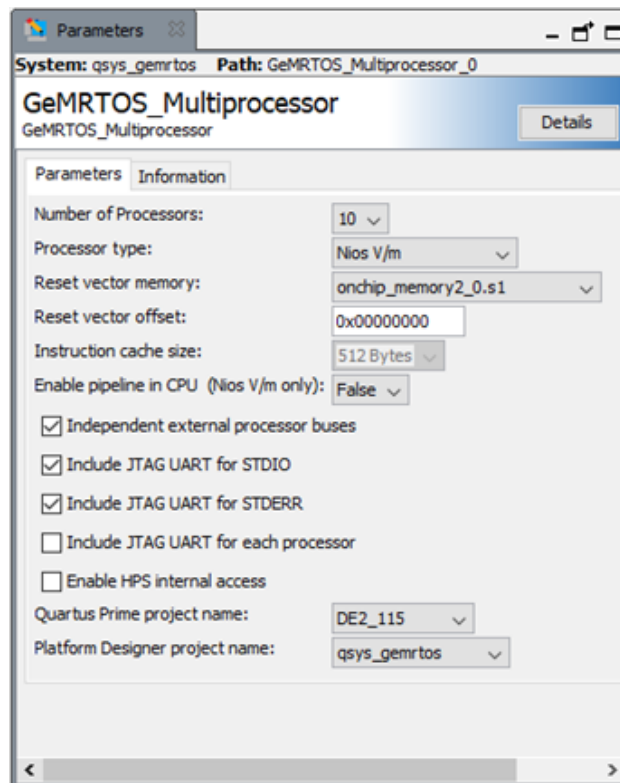


Figure 8.5: GeMRTOS Multiprocessor Parameter Configuration

Properly configuring the GeMRTOS Multiprocessor component is critical for a functional multiprocessor system. This step ensures proper integration and allows for subsequent configuration and refinement.

Configuring the GeMRTOS Hardware Parameters When the GeMRTOS Multiprocessor component is added or double-clicked within Platform Designer, the *Parameters* tab becomes accessible. This tab provides options for configuring key hardware parameters of the GeMRTOS Multiprocessor component.

The following parameters are available for configuration:

- **Number of Processors:** Specifies the number of processors (1-32) to include in the system. The GeMRTOS component manages internal configurations to accommodate the selected number.
- **Processor Type:** Determines the processor type (Nios II/e, Nios II/f, or Nios V/m). Select from the available options.
- **Reset Vector Memory:** Identifies the memory device containing the reset vector. Options are populated based on devices connected to the GeMRTOS Multiprocessor buses.
- **Reset Vector Offset:** Specifies the offset of the reset vector within the selected memory device.

- **Exception Vector Memory (Nios II only):** Identifies the memory device containing the exception vector. Applicable only to Nios II processors.
- **Exception Vector Offset (Nios II only):** Specifies the offset of the exception vector within the selected memory device. Applicable only to Nios II processors.
- **Instruction Cache Size:** Defines the instruction cache size for processors that support it.
- **Enable Pipeline in CPU (Nios V/m only):** Enables the pipeline feature for Nios V/m processors.
- **Independent External Processor Buses:** Determines whether each processor has its own external bus or shares a single external bus. Independent buses can be useful for address shadowing or performance improvement.
- **Include JTAG UART for STDIO:** Includes a JTAG UART for standard input/output (STDIO).
- **Include JTAG UART for STDERR:** Includes a JTAG UART for standard error (STDERR).
- **Include JTAG UART for Each Processor:** Includes a JTAG UART for each processor for debugging purposes.
- **Enable HPS Internal Access:** Includes an Avalon bridge to allow the HPS system to access the GeMRTOS bus and devices.
- **Quartus Prime Project Name:** Specifies the Quartus Prime project name, configuring the `gemrtos_build` script.
- **Platform Designer Project Name:** Specifies the Platform Designer project name, configuring the `gemrtos_build` script.

Properly configuring these parameters is essential for ensuring the GeMRTOS Multiprocessor component operates as intended.

8.3.4 Step 6: Design the Altera's Platform Designer System

This step describes connecting the GeMRTOS Multiprocessor component to the rest of your system within Platform Designer.

As with uniprocessor systems using Nios II or Nios V, you must add and interconnect the remaining components in your system.

Important: If the "*Independent External Processor Buses*" option is enabled, ensure that each processor bus is connected to the appropriate devices.

- Memory devices should be connected before configuring the *Reset and Exception Vector Memory*, as valid options are derived from the memory devices attached to the bus of the processor with an ID equal to 1.
- Connect interrupt signals to the `gemrtos_dirq_input` port of the GeMRTOS Multiprocessor component.
- Connect the GeMRTOS Multiprocessor buses to the clock sources, and any other necessary components in your system.

Figure 8.6 provides an example of connecting the 10 processor buses of a GeMRTOS Multiprocessor-based system to an on-chip memory.

Like Nios II or Nios V processors, interrupt signals must be connected to the `gemrtos_dirq_input` port of the GeMRTOS Multiprocessor component.

The GeMRTOS Multiprocessor component significantly simplifies the design of multiprocessor systems in much the same way as uniprocessor designs. It helps manage complexities such as exclusion components, boot synchronization, interrupt management, system time management, and processor identification.

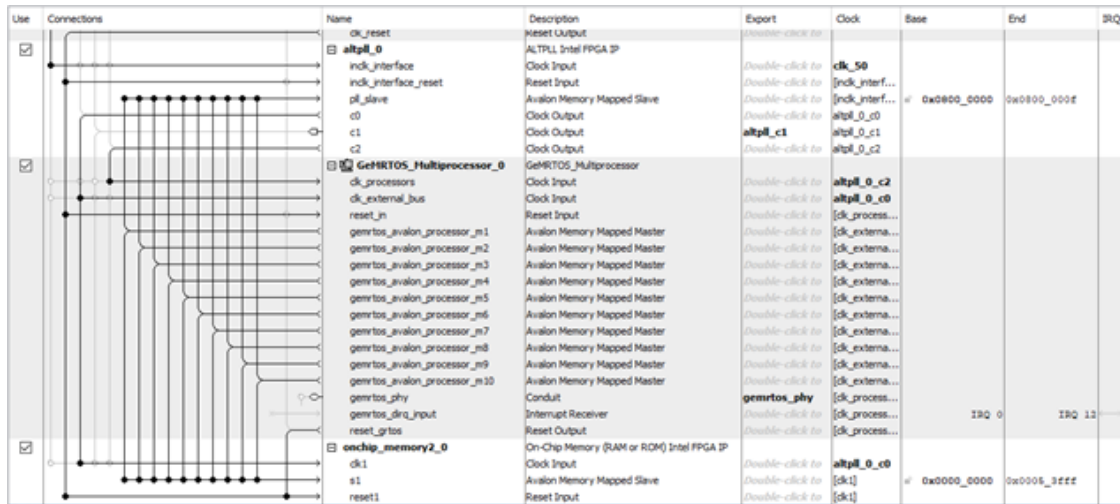


Figure 8.6: GeMRTOS Multiprocessor Connections

8.3.5 Step 7: Generate the Altera's Platform Designer System and Compile the Quartus Prime Project

This step involves generating the hardware design in Platform Designer and compiling it within Quartus Prime to create the FPGA programming file.

1. In Platform Designer, generate the system. This creates the hardware description files for your design.
2. In Quartus Prime, compile the project. The compilation process synthesizes the design, performs place and route, and generates the programming file (.sof or .pof) for your FPGA device.

Ensure that the GeMRTOS Multiprocessor-based system component is instantiated correctly in the Quartus Prime project. You may need to make modifications to include conduit ports added by the GeMRTOS component.

Important: After generating the system in Platform Designer, a .sopcinfo file is created. This file contains all the information needed to generate the Board Support Package (BSP), which provides the software layer for your application.

Figure 8.7 illustrates the complete hardware creation flow. The software creation flow, which builds upon the hardware design, is detailed in the following sections. This flow closely resembles the process used for uniprocessor systems based on Nios II or Nios V.

8.4 Designing the GeMRTOS Multiprocessor Software Application

Building the software application for a GeMRTOS multiprocessor system closely resembles the process for Nios II and Nios V processors.

This process leverages the Board Support Package (BSP) project, which includes the device drivers for all components in the Altera Platform Designer project. The configuration of these devices is determined by the information in the .sopcinfo file generated during the Platform Designer process. This file consolidates device details, ensuring correct alignment between the software and hardware.

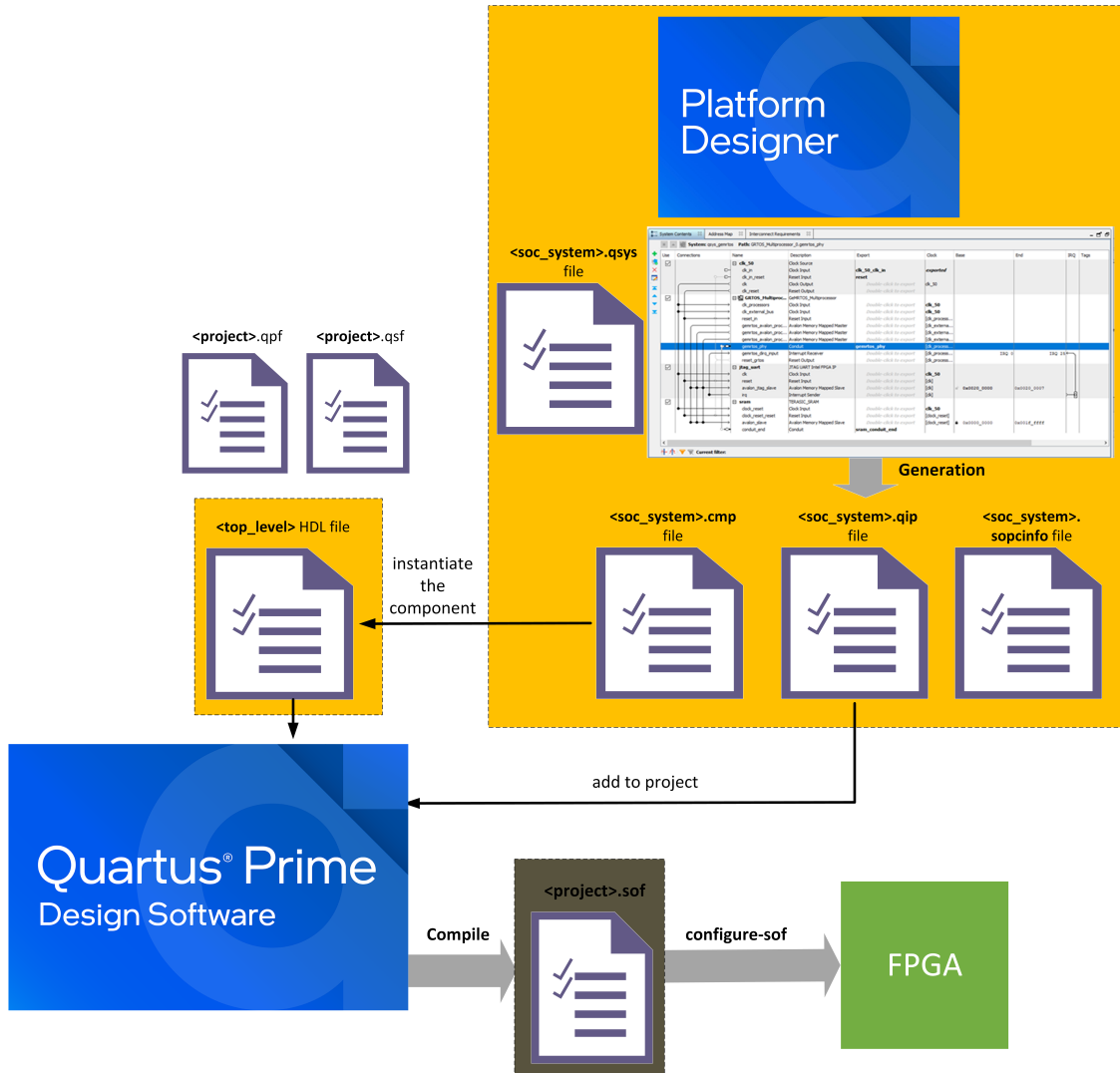


Figure 8.7: Quartus® Prime / Platform Designer hardware design flow

Figure 8.8 illustrates the software layer, containing both the BSP and Application projects.

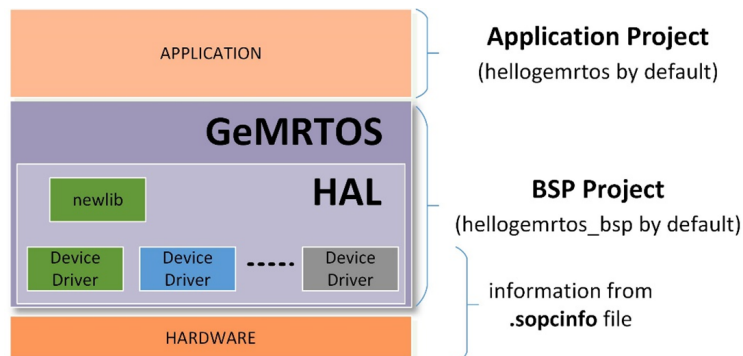


Figure 8.8: GeMRTOS Software Layer: BSP and Application Projects

The software building process comprises several ordered steps involving command execution. The environment for

executing these commands depends on the selected processor: Nios II commands are run in the Nios II Command Shell, while Nios V commands use the Nios V Command Shell (see Figure 8.9).

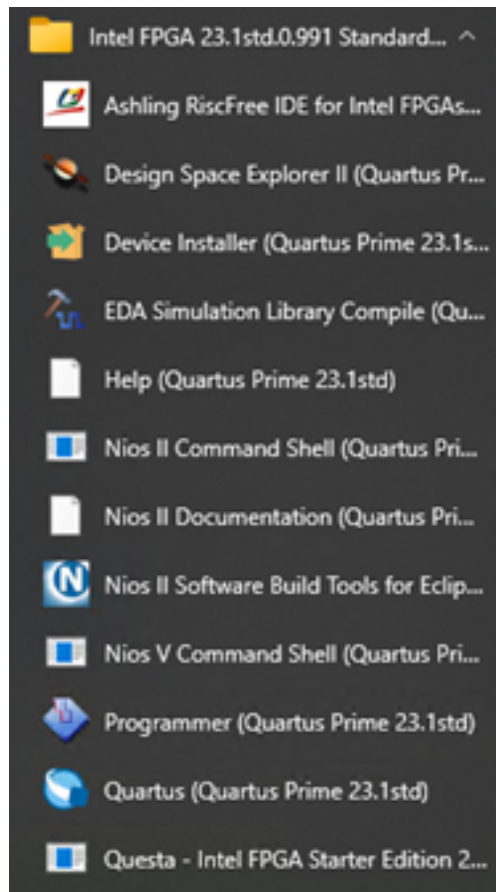


Figure 8.9: Nios® II and Nios® V Command Shells in the Quartus® Prime program group

Note: The Nios II Command Shell requires a Linux-like environment. Quartus Prime versions 22.0 and later use the Windows Subsystem for Linux (WSL) command-line environment. Procedures for installing Quartus Prime version 22.0 and configuring WSL on Windows 10-64bit systems are documented here: [link_to_WSL_configuration](#).

8.4.1 Alternative 1: Building the Application with the `gemrtos_build` Script

This section explains how to use the `gemrtos_build` script for streamlined software application creation.

To simplify the software build process, the GeMRTOS Multiprocessor component generates the `gemrtos_build` script (`.sh` for Nios II, `.bat` for Nios V). This script automates the essential steps for building a GeMRTOS application, reducing manual configuration and improving efficiency, particularly for users new to GeMRTOS.

Execute the script from the command line of the Nios II or Nios V Command Shell within the `<Quartus_Prim`

e_project_folder> directory:

- **Nios II:**

```
1 <Quartus_Prime_project_folder>/bash gemrtos_build.sh
```

- **Nios V:**

```
1 <Quartus_Prime_project_folder>/gemrtos_build.bat
```

The `gemrtos_build` script performs the following key operations, configured with parameters from the GeMRTOS Multiprocessor component's parameter panel:

- **Platform Designer Project Name:** Specifies the Platform Designer project file (`.qsys`) in the `<Quartus_Prime_project_folder>`.
- **Quartus Prime Project Name:** Specifies the Quartus Prime project containing the GeMRTOS IP component.
- **Application Project Name:** Determines the name of the application (default: `hellogemrtos`) created in the `<Quartus_Prime_project_folder>/software` subdirectory.
- **BSP Project Name:** Specifies the name of the Board Support Package (BSP) (default: `hellogemrtos_bsp`) in the `<Quartus_Prime_project_folder>/software` subdirectory.

The `gemrtos_build` script performs the following steps:

1. On the first run only, copies the Windows-simulation files from `ip/GeMRTOS_Multiprocessor/gemrtos_windows` to `software/gemrtos_windows` (see the Windows Emulation Layer chapter, Chapter 14).
2. Determines whether a full hardware compilation is required by comparing the file timestamps of the `.qsys` project and the compiled `.sof` bitstream: a full compilation runs when the `.qsys` is newer than the `.sof`, when the `.sof` is missing, or when the `-f` flag is given. If `processor_set.bat` is missing, the Quartus Prime Standard edition takes a fast path that regenerates it with `qsys-script` alone (no FPGA recompilation), while the Pro edition falls back to a full regeneration.
3. When a full compilation is required, generates the Altera Platform Designer system.
4. When a full compilation is required, compiles the Quartus Prime project, creating the `<quartus_project_name>.sof` file.
5. Generates the BSP settings file (`.bsp`) and BSP project directory.
6. Sets up the software directory and includes the `hellogemrtos.c` template file if needed.
7. Compiles both BSP and application projects, creating the `hellogemrtos.elf` file.
8. Transfers the `.sof` file to the FPGA using the Quartus Prime Programmer tool.
9. Opens terminals to connect to the system's JTAG UARTs.
10. Downloads and initiates execution of the `hellogemrtos.elf` file on the target hardware.

Figure 8.10 illustrates the sequence of operations.

Upon completion, the multiprocessor application will begin executing, with system and task messages displayed in connected terminals.

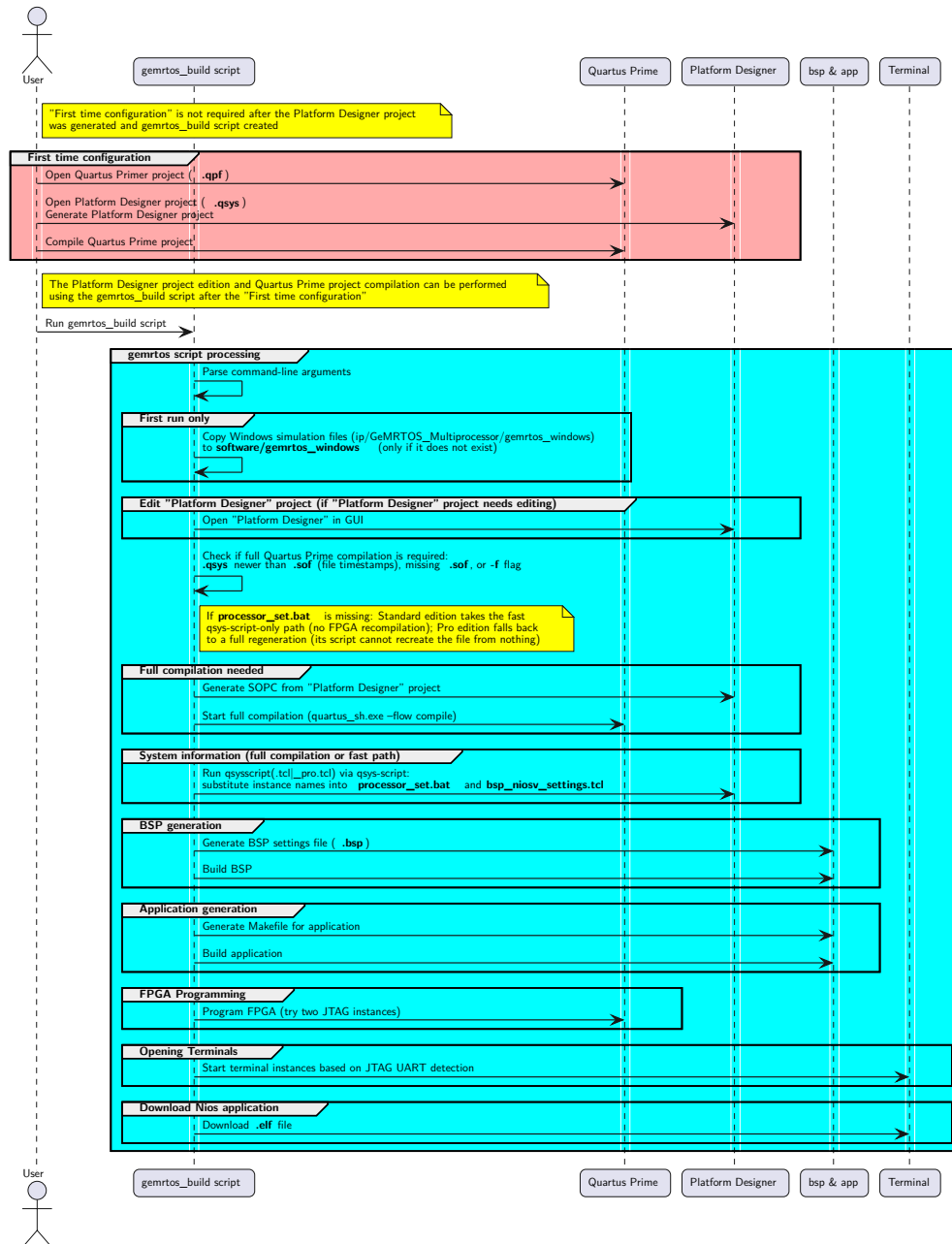


Figure 8.10: gemrtos_build Script Sequence Diagram

8.4.2 Beyond the Default Application: Building a Catalogue Example on Your Board

Running `gemrtos_build` directly, as shown above, always builds the default `hellogemrtos` application. Once your own board project integrates the GeMRTOS IP and `gemrtos_build` runs successfully against it, the same underlying build machinery is also reachable through `new_script.bat`, at the root of the GeMRTOS repository — a wrapper that adds one capability: selecting which application to build via the `-ex` option, from GeMRTOS's eleven-example catalogue (Chapter 2, Section 2.3) instead of only the default demo:

```
new_script.bat -std -ex 01_hello_task --board <your_board_directory> ^
```

```
-qpf <your_qpf_name> -qsys <your_qsys_name> -dir software -sd e -fat f -debug
```

<your_board_directory> is the directory your own board project lives in at the repository root — the same one this chapter has you create and populate. -qpf/-qsys name your Quartus Prime/Platform Designer project files, exactly as configured earlier in this chapter; they do not change per example, only per board. Running `new_script.bat -ex-list --board <your_board_directory>` lists every example available, with a one-line description of each, before building anything.

Note: Not every example needs the same hardware. `04_dining_philosophers` additionally needs LEDs, `07_multiprocessor_scheduling` needs a second processor configured in your board project, and `09_network_httpd/10_full_demo` need Ethernet — Table 2.1 lists this per example. `new_script.bat` does not check this for you: selecting an example your board's hardware does not support builds and runs, but the unsupported functionality (an LED write, a second processor, a network interface) will not do anything observable.

8.4.3 Alternative 2: Executing All the Steps to Create the Software Application

This section outlines the manual software build process, providing greater control over individual steps for advanced users.

While the `gemrtos_build` script offers a simplified approach, users desiring finer-grained control can execute the entire build process manually using a sequence of commands within the Nios II or Nios V Command Shell.

Note: The commands to generate the Altera Platform Designer system and compile the Quartus Prime hardware project (steps typically performed using graphical interfaces) are included here as command-line functions for completeness. These steps are also integrated into the `gemrtos_build` script for automated builds.

Step 1: Update the IP cores of the Altera's Platform Designer project

Purpose: This step updates the IP cores used in the Platform Designer project to their latest versions.

- Run the appropriate command:

- **Nios II:**

```
\path{qsys-generate.exe <Platform_Designer_project_name>.qsys --upgrade-ip-cores}
```

- **Nios V:**

```
\path{%QSYS_ROOTDIR%/qsys-generate <Platform_Designer_project_name>.qsys --upgrade-ip-cores}
```

Note: %QSYS_ROOTDIR% is an environment variable specifying the Altera Platform Designer installation directory.

This step is often performed automatically when opening a project in Platform Designer, especially if the project was created with an older tool version. If the components are already up-to-date, no action is necessary.

The `gemrtos_build` script executes this command automatically using the project name from the GeMRTOS Multiprocessor component's configuration. To override this, use the `-qsys` option:

- **Nios II:**

```
bash gemrtos_built.sh -qsys <Platform_Designer_project_name>
```

- **Nios V:**

```
gemrtos_built.bat -qsys <Platform_Designer_project_name>
```

Step 2: Generate Platform Designer System

Purpose: This step generates the Platform Designer system, producing files for synthesis and the crucial `.sop` `cin` file.

- Run the appropriate command:

- **Nios II:**

```
qsys-generate.exe <Platform_Designer_project_name>.qsys --synthesis=VERILOG
```

- **Nios V:**

```
%QSYS_ROOTDIR%/qsys-generate <Platform_Designer_project_name>.qsys --synthesis=VERILOG
```

Note: This command is equivalent to clicking the "Generate HDL" button in the Platform Designer GUI. The `gemrtos_build` script also performs this step automatically, with the `-qsys` option available to override the project name.

Step 3: Compile the Quartus Prime Project

Purpose: If the Platform Designer or Quartus Prime project files have been modified, a full compilation of the generated hardware is required to implement the changes.

- Run the following command (for both Nios II and Nios V):

```
quartus_cmd.exe <Quartus_Prime_Project_Name>.qpf -c <Quartus_Prime_Project_Name>.qsf
```

Note: The `gemrtos_build` script uses the project name specified in the "Quartus Prime Project Name" parameter of the GeMRTOS Multiprocessor component's configuration panel. To override this, use the `-qpf` option:

- **Nios II:**

```
bash gemrtos_build.sh -qpf <Quartus_Prime_Project_Name>
```

- **Nios V:**

```
\path{gemrto_build.bat -qpf <Quartus_Prime_Project_Name>}
```

This step compiles the entire hardware project and may take considerable time. It is recommended to execute this step, along with Steps 1 and 2, only when the Platform Designer or Quartus Prime project files have been modified.

Step 4: Generate the BSP Settings File

Purpose: This step generates the Board Support Package (BSP), which provides the software support layer for the hardware, including device drivers and system configuration.

- Run the appropriate command:

- **Nios II:**

```
nios2-bsp hal ./<BSP_sub_folder> <Platform_Designer_project_name>.sopcinfo --cpu-name <
```

- **Nios V:**

```
niosv-bsp -c --sopcinfo=<Platform_Designer_project_name>.sopcinfo --type=hal -i=<GeMRTOS_M
```

Where:

- `<BSP_sub_folder>` is the destination folder for the BSP project (default: `./software`).
- `<Platform_Designer_project_name>` is the name of the Platform Designer project.
- `<GeMRTOS_Multiprocessor_component_name>` is the name of the GeMRTOS component in Platform Designer (default: `"GeMRTOS_Multiprocessor_0"`).

This command creates or updates the `settings.bsp` file within the BSP project folder, configuring the BSP project for the processor with ID 1 (the reference processor).

Note: The BSP project is created similarly to a uniprocessor BSP project, but requires specifying the reference processor in a multiprocessor system (processor with ID 1 by default).

To obtain the correct command arguments, you can execute:

- **Nios II:**

```
\path{qsys-script.exe --system-file <Platform_Designer_project_name>.qsys --script=./ip/GeMRTOS
```

- **Nios V:**

```
%QSYS_ROOTDIR%/qsys-script.exe --system-file <Platform_Designer_project_name>.qsys --script=.
```

This will produce the command line to execute according to the system's configuration.

Step 5: Generate the Application Project

Purpose: This step generates the application project, which references the BSP project and contains the application's source files (including the main function).

- Run the appropriate command:

- **Nios II:**

```
nios2-app-generate-makefile.exe --bsp-dir ./<BSP_sub_folder> --app-dir ./<Application_folder>
```

- **Nios V:**

```
niosv-app --app-dir=<Application_folder> --srcs=<Application_folder> --bsp-dir=<BSP_sub_folder>
```

Note: GeMRTOS provides the `hellogemrtos.c` file (located in the `./ip/GeMRTOS_Multiprocessor` folder) as a starting point for your application.

Step 6: Compile Projects and Generate .ELF File

Purpose: This step compiles the BSP and application projects to create the executable `.elf` file, which will be loaded onto the FPGA.

- **Nios II:** Compile the BSP project first and then compile the application project.

```
cd <BSP_sub_folder>
make all
cd <Quartus_Prime_project_folder>
cd <Application_folder>
make all
```

- **Nios V:**

```
cmake -G "Unix Makefiles" -B <Application_folder> -S <Application_folder>
make -C <Application_folder>
```

This compilation process will produce the `.elf` file, used to program the application onto the FPGA.

Step 7: Program the FPGA

Purpose: This step programs the generated hardware configuration onto the FPGA device.

- Use the Quartus Prime Programmer tool, or execute the following command:

```
quartus_pgm.exe -m JTAG -o "p;output_files/<Quartus_Prime_Project_Name>.sof"
```

For devices configured second in the programming chain, use:

```
quartus_pgm.exe -m JTAG -o "p;output_files/<Quartus_Prime_Project_Name>.sof@2"
```

Note: After successful execution, the FPGA will be configured. Verify the system's processors and JTAG UARTs by running:

```
jtagconfig.exe -n
```

This command displays the accessible processors and JTAG UARTs. For example:

```
1) USB-Blaster [USB-0]
   020F30DD   10CL025(Y|Z)/EP3C25/EP4CE22
   Design hash 0CBAE831A5A1AE5AA13C
+ Node 19104600  Nios II #0
+ Node 19104601  Nios II #1
+ Node 19104602  Nios II #2
+ Node 0C006E00  JTAG UART #0
+ Node 0C006E01  JTAG UART #1
+ Node 0C206E00  JTAG PHY #0
+ Node 0C006E02  JTAG UART #2
```

Step 8: Download and Run the Application

Purpose: This step downloads the compiled application to the target hardware and initiates its execution.

- Run the appropriate command:

– **Nios II:**

```
nios2-download --go -r ./<Application_folder>/<Application_name>.elf --instance=0
```

– **Nios V:**

```
niosv-download -g -i 0 <Application_folder>/<Application_name>.elf
```

Upon execution, the application is downloaded and begins running. If the application sends messages to the JTAG UART, it will pause, awaiting a terminal connection.

Note: It is necessary to connect as many terminals as there are JTAG UART devices. Use the following commands:

- **Nios II:**

```
cmd.exe /c start nios2-terminal.exe -v --flush --no-quit-on-ctrl-d --instance=<instance>
```

- **Nios V:**

```
cmd.exe /c start juart-terminal.exe -v --flush --no-quit-on-ctrl-d --instance=<instance>
```

Here, <instance> ranges from 0 to the total number of JTAG UART devices in the system minus 1.

Following these steps gives users full control over the software build process, and allows for customization of each phase of the GeMRTOS application development.

8.4.4 Alternative 3: Creating Your First GeMRTOS Project with the `gemrtos_build` Script

This section provides a simplified, step-by-step guide for setting up a multiprocessor system on Altera FPGAs, particularly beneficial for new GeMRTOS users.

GeMRTOS simplifies setting up multiprocessor systems on Altera FPGAs through its automated design flow, making it convenient for new users. This straightforward approach ensures a smooth initial experience by automating much of the setup, allowing you to quickly familiarize yourself with GeMRTOS and its capabilities.

Follow these steps to get started:

1. Obtain a Quartus Prime project with a pre-existing SOPC project tailored for your board, using either the Nios II or Nios V soft-processor.
2. Download, unzip, and copy the GeMRTOS component into the /ip subfolder of the Quartus Prime project.
3. Verify that the GeMRTOS license is correctly configured.
4. Open the Quartus Prime project in Quartus Prime.
5. Launch the SOPC project using Altera's Platform Designer tool.
6. Replace the Nios II or Nios V processor with the GeMRTOS Multiprocessor component.
7. Connect buses and interrupts as they were connected to the original processor of the project
8. Configure the parameters of the GeMRTOS Multiprocessor component.
9. Check and adjust the top-level design if necessary due to SOPC interface modifications.
10. Save and exit the Altera Platform Designer tool.
11. Save and exit Quartus Prime.
12. Run the `gemrtos_build` script in the Command Shell.
13. Enjoy your Multiprocessor System!

Note: By default, the `gemrtos_build` script generates two projects: `hellogemrtos` and `hellogemrtos_bsp`, located in the `/software` subdirectory of the `<Quartus_Prime_Project_Name>` directory. The `hellogemrtos` project includes a template C code file named `hellogemrtos.c`, providing a basic starting point for your GeMRTOS application. You can modify this file to fit your project's specific requirements.

8.5 Conclusion

This Development Guide has outlined the essential steps for creating and deploying applications using the GeMRTOS Multiprocessor RTOS within the Altera FPGA design environment. By following the procedures detailed in this chapter, developers can effectively navigate the hardware design and software development flows, leverage the automated build script, and customize their projects to meet specific application requirements. The chapter emphasizes clarity and practicality, aiming to provide a solid foundation for both novice and experienced embedded systems engineers.

The power of GeMRTOS lies in its ability to simplify complex multiprocessor designs. Through its intuitive interface and automated features, developers can focus on application-specific logic rather than struggling with the intricacies of multiprocessor management. The examples and instructions in this guide demonstrate how GeMRTOS seamlessly integrates into the established Altera design flow, making it a natural extension for those familiar with Nios II or Nios V development.

The manual build process, while more involved, provides invaluable insight into the underlying mechanisms of GeMRTOS and allows for fine-grained control over each step of the development cycle. This understanding is crucial for troubleshooting issues and optimizing system performance. Whether you choose the automated script or the manual approach, this guide equips you with the knowledge and tools needed to succeed.

Ultimately, the goal is to empower developers to harness the full potential of multiprocessor systems on Altera FPGAs. With its flexible architecture, comprehensive feature set, and a development process designed for ease of use, GeMRTOS offers a robust platform for building innovative and high-performance embedded applications.

GeMRTOS Programming Guide

9.1 Introduction

GeMRTOS provides an efficient and intuitive mechanism for implementing embedded multiprocessor systems on Altera FPGAs. Building upon familiar concepts from traditional 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.

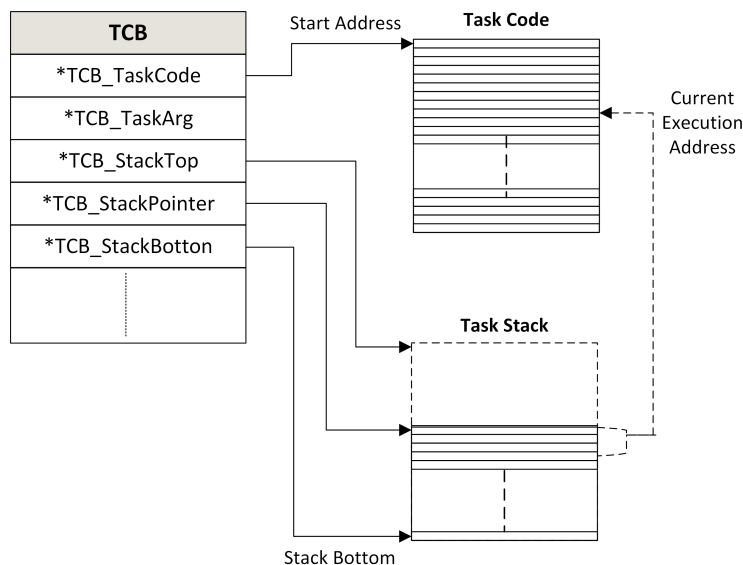


Figure 9.1: GeMRTOS Task Main Components

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:

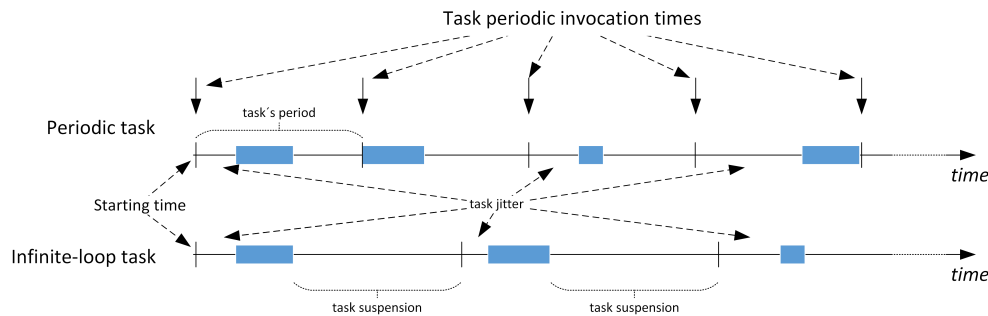


Figure 9.2: Periodic vs. Infinite-Loop Task Runtime Behavior

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 `\glstask\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 */
4     /* Set a 600ms waiting timeout for LEDs Mutex */
```

```

5  /* Define Mutex timeout (600ms) */
6  G_UINT64 timeout_leds = gu_ConvertTime(0, 0, 0, 600);
7
8  /* Request LEDs Mutex with timeout */
9  if (gu_SemaphoreTake(mutex_leds, timeout_leds) == G_TRUE) {
10     leds = leds ^ 1;          /* Invert LED 1          */
11     gm_WriteOutputs(leds);    /* Write LEDs to GeMRTOS output */
12     gu_SemaphoreGive(mutex_leds); /* Release Mutex          */
13 } else {
14     /* mutex_leds timeout */
15     gu_printf("Semaphore not taken in \gls{task} periodic\n");
16 }
17 }

```

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.

```

1  t_semaphore_resource *mutex_leds;
2
3  int main(void)
4  {
5      // System configuration
6      mutex_leds = gu_SemaphoreCreateRecursiveMutex();
7      ...
8      GS_TCB *ptcb = gu_TaskCreate((void *) task_periodic, /* Task code */
9      (void *) 1,          /* Call argument */
10     "periodic \gls{task}"); /* Task description */
11
12     if (ptcb != (GS_TCB *) 0) { /* Task was created successfully */
13         gu_TaskTypeSet(ptcb, G_TCBType_Periodic); /* Set the \gls{task} as periodic */
14
15         gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 1); /* Set READY priority */
16         gu_TaskRunPrioritySet(ptcb, (G_UINT64) 1); /* set RUN priority */
17
18         gu_TaskPeriodSet(ptcb, 0, 0, 1, 0); /* Set \gls{task} period (1 second) */
19
20         gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0); /* Set starting time */
21
22     } else {
23         gu_printf("Error when creating \gls{task}\n");
24         while(1);

```

```

25 }
26 ...
27 }

```

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:

```

1  void task_infinite_loop(void* pdata)
2  {
3      while(1) {
4          /* Set a 600ms waiting timeout for LEDs Mutex */
5          /* Define Mutex timeout (600ms) */
6          G_UINT64 timeout_leds = gu_ConvertTime(0, 0, 0, 600);
7
8          /* Request LEDs Mutex */
9          if (gu_SemaphoreTake(mutex_leds, timeout_leds) == G_TRUE) {
10             leds = leds ^ 2;          /* Invert LED 1 */
11             gm_WriteOutputs(leds);    /* Write LEDs to GeMRTOS output */
12             gu_SemaphoreGive(mutex_leds); /* Release Mutex */
13         } else {
14             /* mutex_leds timeout */
15             gu_printf("Semaphore not taken in \gls{task} periodic\n");
16         }
17         gu_TaskDelay( 0, 0, 1, 0);    /* Suspend \gls{task} 1 second */
18     }
19 }

```

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).

```

1  ...
2  ptc_b = gu_TaskCreate((void *) task_infinite_loop, /* Task code */
3                      (void *) 2,                  /* Call argument */
4                      "infinite-loop \gls{task}"); /* \gls{task} description */
5
6  if (ptcb != (GS_TCB *) 0) { /* Task was created successfully */
7      /* Set the \gls{task} as one-shot */
8      gu_TaskTypeSet(ptcb, G_TCBType_OneShot);
9
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 Recovering from a Hung Task: the Per-Processor Watchdog

A per-processor hardware watchdog (§4.5) resets a processor that stops halting – a task that spins forever without ever yielding control back to the scheduler. This section covers the application-level pattern for using it; the register-level and kernel-recovery detail is in Chapter 4.

9.3.1 Arming a Task's Own Watchdog

Arming is per-task, not automatic or system-wide: a task arms its own processor's watchdog explicitly, choosing a timeout appropriate to how long it can legitimately take before yielding again.

```

1  void my_task(void *pdata)
2  {
3      /* Arm once, before entering the part of the loop that must not
4       * hang for longer than the configured timeout. */
5      gm_WatchdogTimeoutSet(150U << G_WATCHDOG_TIMEOUT_SHIFT); /* ~983ms */
6      gm_PrcWatchdogEnable();
7
8      while (1) {
9          /* ... normal task work, including gu_TaskDelay() or other
10         * calls that let this processor return to halt -- each
11         * halt reloads the watchdog countdown, so a task that
12         * yields normally never trips it. */
13     }
14 }

```

Code 9.5: Arming the Current Processor's Watchdog

9.3.2 Registering a Recovery Handler

By default, if a watchdog fires and no handler is registered, the system simply resumes normal operation once the kernel's own recovery completes – silently, with no application-visible effect beyond the abandoned task no

longer running. An application that wants visibility into this, or wants to decide whether the event is acceptable, registers a handler once at start-up, before any processor's watchdog is armed:

```

1  enum g_watchdog_action my_watchdog_handler(
2      const struct g_watchdog_report *report)
3  {
4      gu_printf("watchdog recovered cpu=%u task=\"%s\"\n",
5              (unsigned int) report->CPUID,
6              report->TaskDescription);
7
8      /* G_WATCHDOG_ACTION_RESUME: accept the recovery, keep running.
9       * G_WATCHDOG_ACTION_HALT: treat this as unacceptable; halt with
10      * diagnostics instead of continuing. */
11     return G_WATCHDOG_ACTION_RESUME;
12 }
13
14 int main(void)
15 {
16     gu_WatchdogHandlerSet(my_watchdog_handler);
17     /* ... create and start tasks as usual ... */
18 }

```

Code 9.6: Registering a Watchdog Recovery Handler

The handler receives a read-only snapshot (`struct g_watchdog_report`) describing the abandoned task – *after* the kernel has already restored its own internal invariants, never a live pointer into kernel structures. One handler is registered system-wide, not per-processor; `report->CPUID` distinguishes which processor recovered if the application needs to react differently per processor.

Scope note: this watchdog is for a hung task, not a substitute for fixing a kernel bug. A legitimate firing structurally cannot happen while the processor holds the GeMRTOS Mutex – if that assumption is ever violated, recovery halts loudly rather than silently papering over what would be a genuine kernel defect.

9.4 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.

9.5 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.

9.5.1 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.

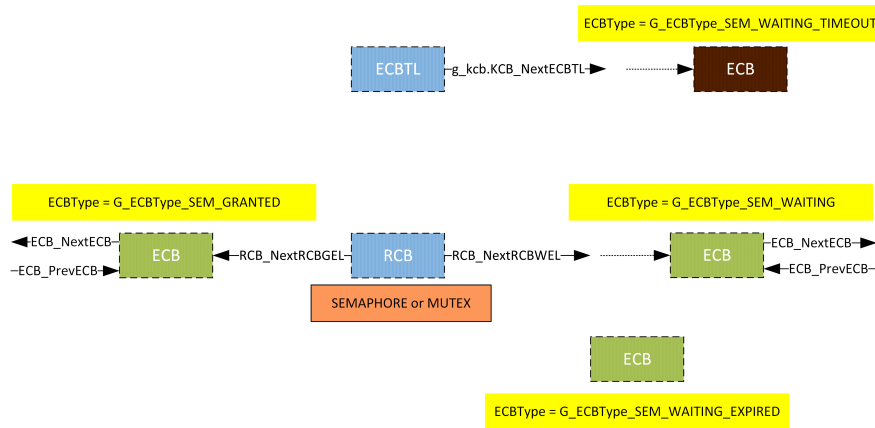


Figure 9.3: ECB States for Semaphore/Mutex Resources

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.

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

9.5.2 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.

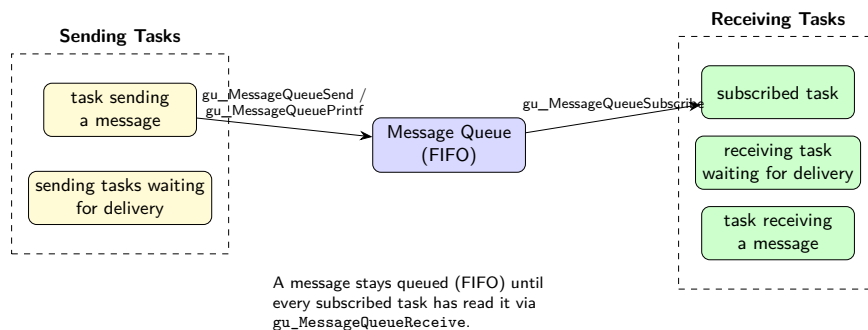


Figure 9.4: Message Queue flow

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 message 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_MESSAGE_RECEIVER_SUBSCRIBED` to the granting event list (RCBEG) of the RCB control block of the message queue. It returns a pointer to the ECB control block associated with the granting ECB list (RCBEG) 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_SENDER_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_RECEIVER_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
<code>gu_MessageQueueCreate</code>	<code>GS_RCB *</code>	Non-NULL pointer to the new queue; NULL if the RCB pool is exhausted.
<code>gu_MessageQueueSubscribe</code>	<code>GS_ECB *</code>	Non-NULL receiver ECB; NULL if the ECB or RRDS pool is exhausted.
<code>gu_MessageQueueSend</code>	<code>int</code>	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.
<code>gu_MessageQueueReceive</code>	<code>int</code>	Number of bytes copied into the subscriber buffer on this call. Returns 0 if the queue was destroyed while the task was waiting.
<code>gu_MessageQueuePrintf</code>	<code>int</code>	Same semantics as <code>gu_MessageQueueSend</code> applied to the formatted string.
<code>gu_MessageQueueDestroy</code>	<code>G_UINT32</code>	<code>G_TRUE</code> on success; <code>G_FALSE</code> 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.

9.5.3 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.

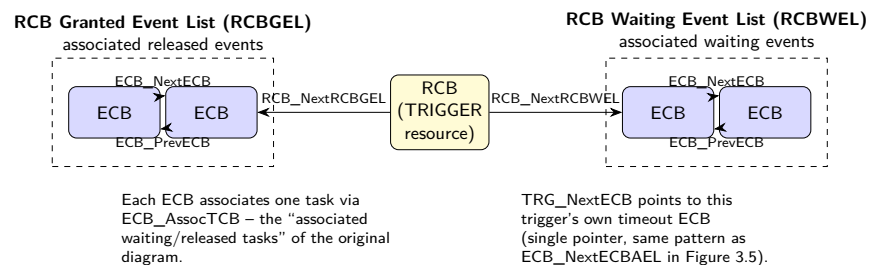


Figure 9.5: The Trigger Structure

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 reenab 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 which 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 9.5.3.

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);
```

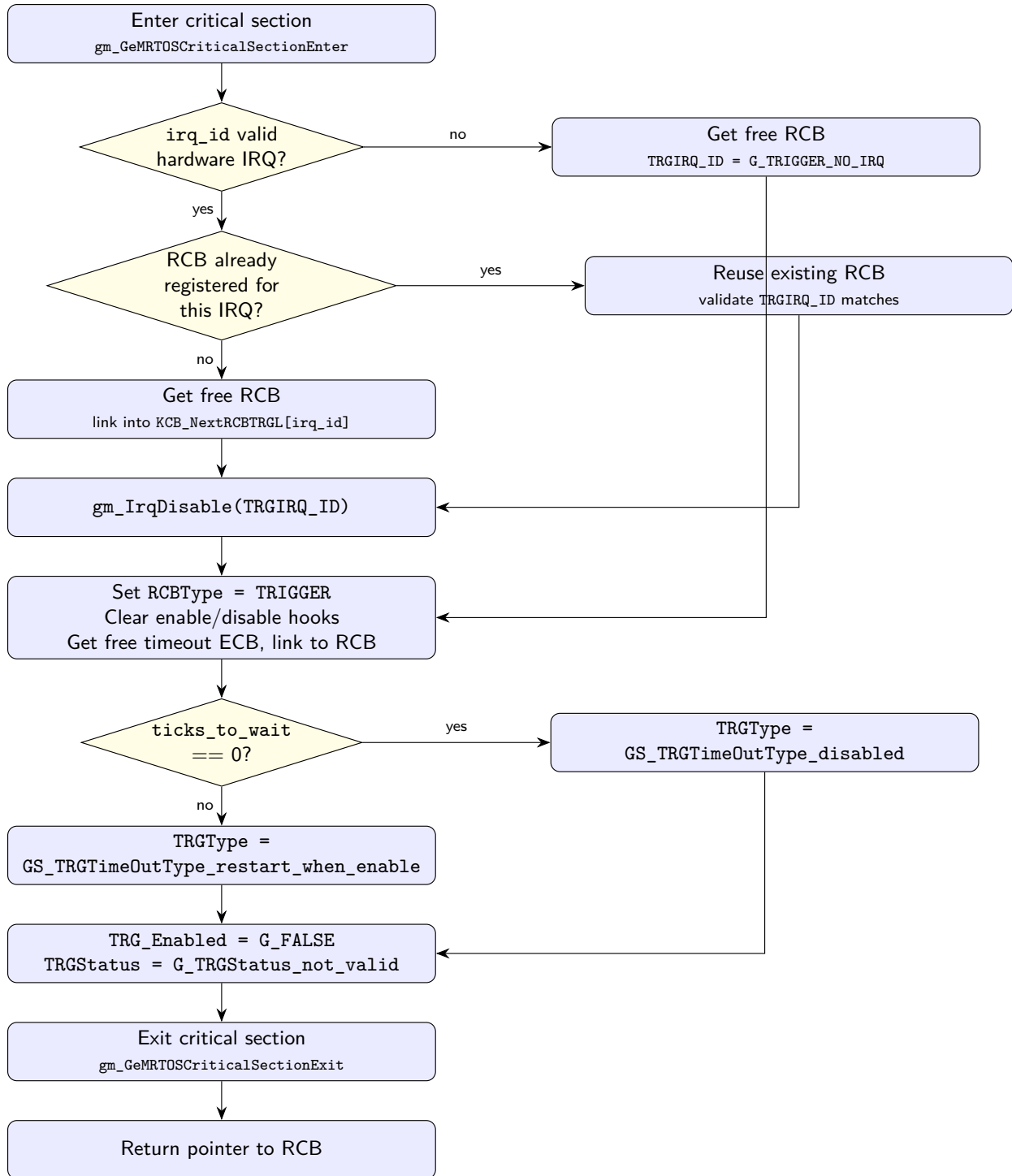


Figure 9.6: Trigger creation process

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

to.

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.

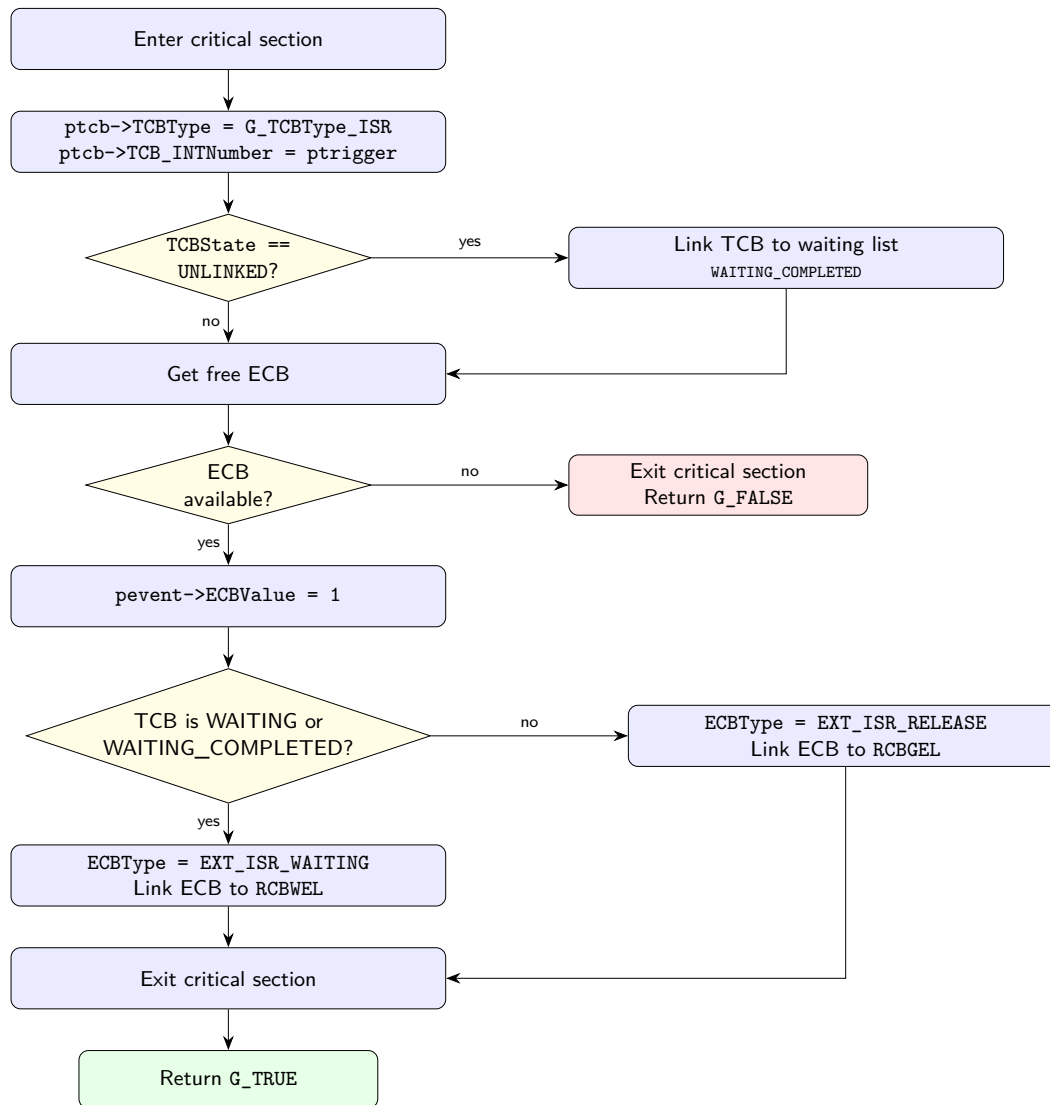


Figure 9.7: Task trigger registration process

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:

- `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 9.7:

```
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 9.7: 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 9.7 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).

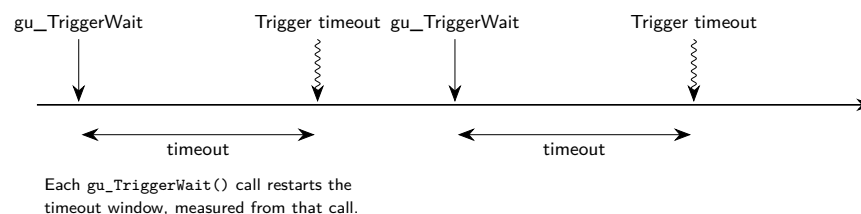


Figure 9.8: Trigger restart when enable timeout

- `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).

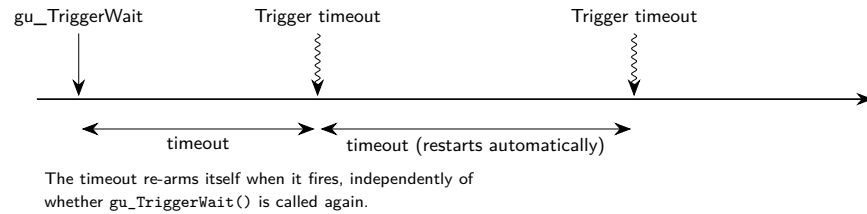


Figure 9.9: Trigger restarts when timeout.

- `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).

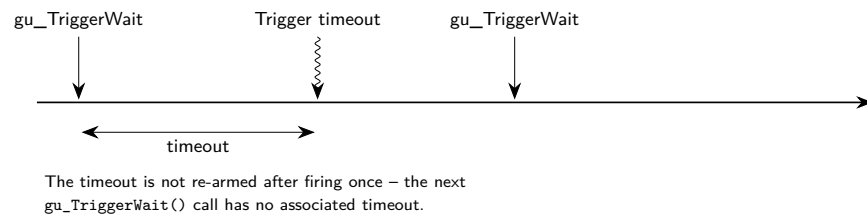


Figure 9.10: Trigger no-restart timeout.

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.

Code Examples

10.1 Introduction

In real-time applications, effective and reliable code patterns are essential to ensure that system responses meet strict timing and synchronization requirements. These common patterns help developers implement predictable behavior, manage task priorities, handle interrupts, and coordinate resources efficiently within the constraints of real-time systems. Understanding these fundamental code structures is crucial for designing robust applications that can handle complex, time-critical operations seamlessly.

This section will describe the most frequently used code patterns in GeMRTOS-based real-time applications, providing practical examples to illustrate their implementation. By examining these typical patterns, developers can better grasp best practices for synchronization, task management, and event handling, enabling them to create efficient and dependable real-time systems tailored to specific application needs.

10.2 Example 1: Creating Infinite-Loop Tasks

In real-time applications, infinite-loop tasks are used to execute continuous, background operations that must run persistently during the system's operation. These tasks handle ongoing monitoring, data acquisition, or system health checks, and are implemented with the system's task management features to ensure they operate predictably and efficiently. Each task possesses assigned priorities for both the ready and running states, facilitating effective scheduling and behavior during runtime. This priority structure allows the scheduler to manage task execution efficiently, ensuring that the most critical functions receive appropriate CPU time and system responsiveness is maintained.

When implementing infinite-loop tasks in GeMRTOS, it is essential to incorporate mechanisms such as periodic delays (`gu_TaskDelay` or `gu_TaskDelayTime`), event waiting (using Triggers, Semaphores, Message Queues), which temporarily relinquish processor control to allow other tasks to execute.

GeMRTOS enables defining separate priorities for the ready and running states of tasks, providing significant flexibility in determining task interactions and scheduling behavior. This priority management allows developers to specify which tasks can be preempted or resumed based on system needs, optimizing responsiveness and ensuring critical functions are always prioritized.

By carefully designing these loops and leveraging GeMRTOS's scheduling policies along with its synchronization

mechanisms, developers can create systems that exhibit reliable, deterministic, and highly responsive behavior suitable for real-time embedded applications.

Code 10.1 demonstrates a basic structure of the implementation of an infinite-loop task. In GeMRTOS, a task is defined as a C function that returns void and with a void argument. In this example, function argument is defined as a task index that may be used to identify different tasks created with the same code. This function argument may also be defined as a pointer to a structure that holds different tasks parameters such as identification and task period.

```
void task_infinite_loop(void* data)
{
    int task_index = (int) data; /* function argument pass the \gls{task} index */
    /* defined when \gls{task} is created in main */
    while(1) {
        /* print a message to show \gls{task} is running */
        gu_printf("Infinite-loop \gls{task} %d running\n", task_index);

        /* Suspend \gls{task} 1 second */
        gu_TaskDelay( 0, 0, 1, 0);
    }
}
```

Code 10.1: Infinite-loop task example.

The main() function creates the task. Code 10.2 shows an example of how an infinite-loop may be created. The argument, defined as 2, specifies that the task index is 2 for the task. The last argument of the create function, the task description string, is a variadic one, like the traditional C printf function. In this case, the task description is defined as “infinite-loop task 2”. Of course, when tasks are created in a for loop, the task index may be resolved using the iteration variable.

```
void main(void)
{
    ... user code ...

    /* Create \gls{task} using the task_infinite_loop function */
    /* called with argument 2 */
    /* define \gls{task} description as "infinite-loop \gls{task} "2 */
    ptcb = gu_TaskCreate((void *) task_infinite_loop, /* Task code */
        (void *) 2, /* Call argument */
        "infinite-loop \gls{task} %d", 2); /* \gls{task} description */

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

        gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 2); /* Ready priority */
        gu_TaskRunPrioritySet(ptcb, (G_UINT64) 1); /* Run priority */
        gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0); /* Starting time */
    } else {
        gu_printf("Error when creating \gls{task}\n");
        while(1);
    }
}
```

```

}

... user code ...

}

```

Code 10.2: Infinite-loop task configuration example.

If task is successfully created (return value different from (GS_TCB *) 0), the task parameters are defined by executing:

- `gu_TaskTypeSet(ptcb, G_TCBType_OneShot);`
sets the task as OneShot type. GeMRTOS executes this type of task just once. It is up to the task to keep executing if it is an infinite-loop task.
- `gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 2);`
sets the ready priority of task, in this case with a value of 2. As smaller the value, the higher the priority. Defining priority equal to 2, it must wait for the executing of tasks with priority equal to 1 and equal to 0 to complete before to be granted with a processor to execute.
- `gu_TaskRunPrioritySet(ptcb, (G_UINT64) 0);`
set the executing priority of the task, in this case with a value of 1. Only tasks with ready priority equal to 0 will preempt the executing of this task. Managing the ready and running priority tasks allows flexibility in determining the runtime behavior of the real-time system.
- `gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0);`
sets the releasing time of the task equal to 5 seconds. This function defines when the system tasks are released for the first time.

This example shows the basic but flexible creation of an infinite-loop task. As described in section 9.2.1, infinite-loop tasks has specific features for real-time systems.

10.3 Example 2: Creating Periodic Tasks

In real-time applications, periodic tasks are employed to perform regular, time-sensitive operations that need to occur at fixed intervals throughout the system's operation. These tasks are designed to handle functions such as sensor sampling, control loop updates, or routine status checks, and are implemented using the system's task management features to ensure they execute predictably and efficiently. Each periodic task is assigned specific priorities for both the ready and running states, allowing effective management of execution order and timing during runtime. This priority differentiation provides the scheduler with flexibility to control which tasks are preempted or resumed, maintaining system responsiveness and meeting strict timing deadlines.

Code 10.3 demonstrates a basic structure of the implementation of a periodic task. In GeMRTOS, a task is defined as a C function that returns void and with a void argument. In this example, function argument is defined as a task index that may be used to identify different tasks created with the same code. This function argument may also be defined as a pointer to a structure that holds different tasks parameters such as identification and task period. In this example, task prints a message indicating it is running and the index assigned when created.

```

void task_periodic (void* data)
{
    int task_index = (int) data; /* function argument pass the \gls{task} index */
    /* defined when \gls{task} is created in main */
    /* print a message to show \gls{task} is running */
    gu_printf("Periodic \gls{task} %d running\n", task_index);
}

```

}

Code 10.3: Periodic task example

The `main()` function creates the task. Code 10.4 shows an example of how a periodic task may be created. The argument, defined as 1, specifies that the task index is 1 for the task. The last argument of the create function, the task description string, is a variadic one, like the traditional C `printf` function. In this case, the task description is defined as "periodic task 1". Of course, when tasks are created in a for loop, the task index may be resolved using the iteration variable.

```
void main(void)
{
    ... user code ...

    /* Create \gls{task} using the task_periodic function          */
    /* called with argument 1                                     */
    /* define \gls{task} description as "periodic \gls{task} "2    */
    ptcb = gu_TaskCreate((void *) task_periodic,                /* Task code */
        (void *) 1,                                           /* Call argument */
        "periodic \gls{task} %d", 1);                          /* \gls{task} description */

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

        gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 1);           /* Ready priority */
        gu_TaskRunPrioritySet(ptcb, (G_UINT64) 0);              /* Run priority */

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

        gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0);              /* Starting time */
    } else {
        gu_printf("Error when creating \gls{task}\n");
        while(1);
    }

    ... user code ...
}
```

Code 10.4: Periodic task configuration example

If task is successfully created, return value different from `(GS_TCB *) 0`, then the task parameters are defined by executing:

- `gu_TaskTypeSet(ptcb, G_TCBType_Periodic)`
sets the task as Periodic type. GeMRTOS executes this type of task periodically according to the period defined. When the task is completed, its state becomes waiting and it is released when the next periodic release time takes place. When the task is completed, all the events, but the periodic one, are freed as well as the granted resources are freed.
- `gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 1);`

sets the ready priority of task, in this case with a value of 1. As smaller the value, the higher the priority. Defining priority equal to 2, it must wait for the executing of tasks with priority equal to 0 to complete before to be granted with a processor to execute.

- `gu_TaskRunPrioritySet(ptcb, (G_UINT64) 0);`
set the executing priority of the task, in this case with a value of 0. In this case, no task will preempt the executing of this task. Managing the ready and running priority tasks allows flexibility in determining the runtime behavior of the real-time system.
- `gu_TaskPeriodSet(ptcb, 0, 0, 1, 0);`
set the period of the task equal to 1 second. Task will be released every 1 second since its starting time.
- `gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0);`
sets the releasing time of the task equal to 5 seconds. This function defines when the system tasks are released for the first time.

This example shows the basic but flexible creation of a periodic task. As described in section 9.2.1, periodic tasks has specific features for real-time systems.

10.4 Example 3: Trigger-Driven JTAG-UART Management Tasks

JTAG UART devices is a very valuable peripheral in embedded systems. It allows designers to communicate the embedded application with the host computer for debugging and messaging. JTAG UART device is included in the Platform Designer project and its Avalon Memory Mapped slave port should be connected to all the processor master ports of the GeMRTOS IP in order to be able to access the internal JTAG UART register to send and receive messages. By default, Platform Designer assigns the `JTAG_UART_0` name for the first JTAG UART device included in the design and that is the name considered in this example. In the case that a JTAG UART device with different name is considered, then all the references should be modified accordingly. Nios® II processor users can access the JTAG UART via the `nios2-terminal` command-line utility whilst Nios® V processor users can access the JTAG UART using the `juart-terminal` command-line utility.

10.4.1 The JTAG-UART Device

The Altera JTAG UART is a hardware peripheral that offers a serial communication interface via the JTAG port, enabling data exchange between the FPGA or embedded system and a host computer. It functions as a virtual serial port, facilitating debugging, logging, and data transfer without the need for extra UART hardware. This capability is especially valuable for embedded development and testing, providing a straightforward and efficient means to communicate with the system through standard serial tools over the JTAG connection.

Figure 10.1 shows the block diagram of the JTAG UART core. Through the Avalon-MM interface, internal data, control, send and receive register are access for configuring and data management

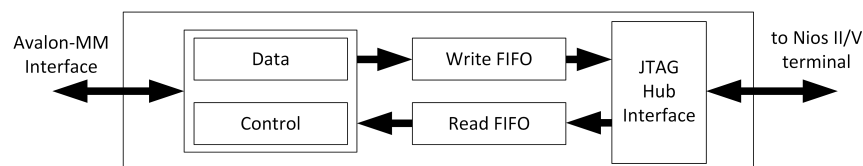


Figure 10.1: JTAG UART Core block diagram

Configuring the JTAG UART in Platform Designer involves adding the JTAG UART component to your system

design and establishing the necessary connections such as clock, reset, interrupt and Avalon-MM interface to the GeMRTOS processor interfaces. Figure 10.2 shows a common connection among JTAG UART and GeMRTOS processors in a typical embedded application.

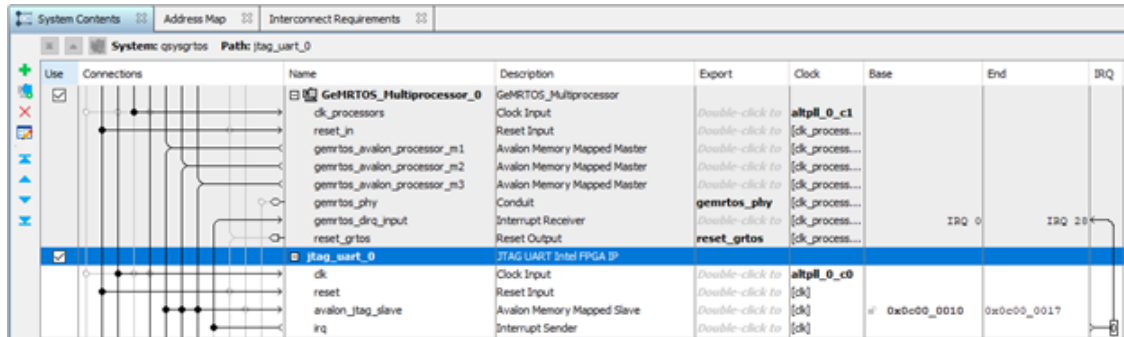


Figure 10.2: JTAG-UART interface in a GeMRTOS-based system

The JTAG UART interface maps all the control, status, and transmit and receive buffers into the memory map of the system. The processors access the JTAG UART functionality through these registers.

JTAG-UART Software Programming model :

When the JTAG-UART core is included in a Platform Design project, software macros are included in the software application for easy access to the device configuration and data management. GeMRTOS `gu_printf` function enable access the JTAG UART for output message. HAL system library allows accessing JTAG UART using ANSI C standard library functions, such as `printf()` and `getchar()`, but it should take into account that must be executed in mutex section to make it thread safe.

For an efficient management of the JTAG UART resource, the JTAG UART core provides an interrupt signal to request data processing when it is necessary, and it requires knowledge about the device registers. Table 10.1 shows the register map for JTAG UART core used to manage the device behavior.

Table 10.1: JTAG UART Core Register Map

Offset	Register Name	R/W	Bit Description								
			31 - 16	15	14 - 11	10	9	8	7 - 2	1	0
0	data	R/W	RAVAIL	RVALID	Reserved			DATA			
1	control	R/W	WSPACE	Reserved		AC	WI	RI	Reserved	WE	RE

The WE and RE bits enable interrupts for the write and read FIFOs. The RI and WI bits reflect the status of these interrupt sources, controlled by the enable bits (RE and WE). Software can check RI and WI to identify the cause of the IRQ. An ISR should check whether the IRQ was produced by either WI, RI or both, and process the data accordingly.

Figure 10.3 show a template layout for an ISR for a JTAG UART device. The ISR checks if the IRQ was produced by the writing FIFO, the reading FIFO, or both. If there was a writing interrupt, the writing data processing is executed to write (send) new data. If there was a reading interrupt, the reading data processing is executed to read (receive) new data.

When writing and reading is wanted to be shared among several tasks that want to use the JTAG UART device, this simple template becomes more complex to include enabling and disabling both interrupt sources (read and write) and determining the instant that the device should be enabled for all the tasks that share it use.

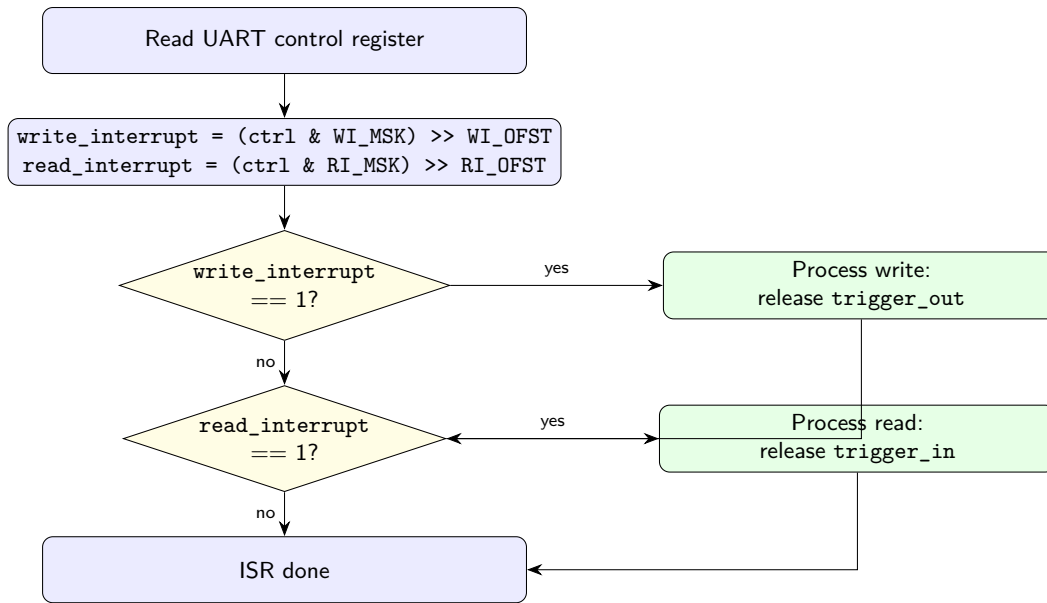


Figure 10.3: JTAG UART ISR template

Note: When writing and reading is wanted to be shared among several tasks that want to use the JTAG UART device, this simple template becomes more complex to include enabling and disabling both interrupt sources (read and write) and determining the instant that the device should be enabled for all the tasks that share it use.

GeMRTOS trigger resources is an efficient way to manage complexity of event management in embedded systems.

10.4.2 Using Trigger for Easy and Efficient JTAG UART Management

A task triggered by an event should be similar in structure to one that runs periodically at fixed intervals (their nature—time-based or external—may differ, but the core layout of the task remains consistent). This facilitates flexible event handling using triggers.

Figure 10.4 show a layout of a event-based tasks: the task stay suspended until either the associated event is activated or the timeout is expired. Then, the task is executed to process the event or the timeout, and then goes to suspended again waiting for the next time the event is activated.

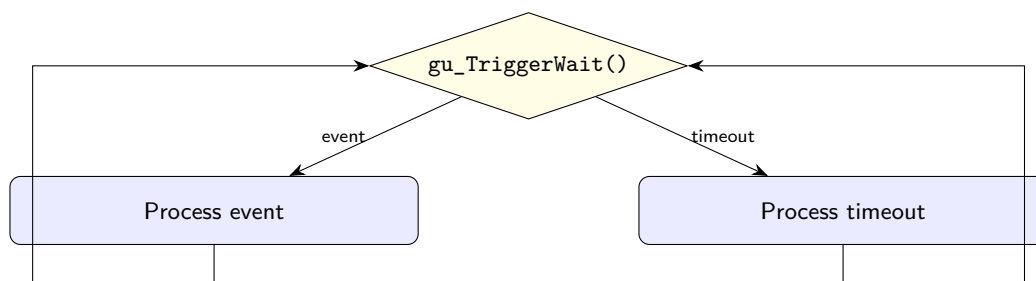


Figure 10.4: Event based task layout

In GeMRTOS, this event-based task is naturally implemented using GeMRTOS triggers.

Reading from JTAG UART using Trigger: A routine for handling JTAG UART read service should run whenever the device issues a read interrupt. It reads from the read FIFO, appending characters to the received message buffer. When an end-of-message character is detected or a timeout occurs, the message is processed.

Figure 10.5 shows a diagram of the reading JTAG UART routine of code 10.5. The routine registers the task with the trigger of the JTAG UART received character and start an infinite-loop that suspends itself waiting for a receiving trigger. The task is resumed from the waiting for a trigger state either because an character is received or because a timeout. When a character is received, it is stored in the local buffer. The buffer is printed in the stderr either a and of message character ('\\r' or '\\n') is received or if an timeout takes place.

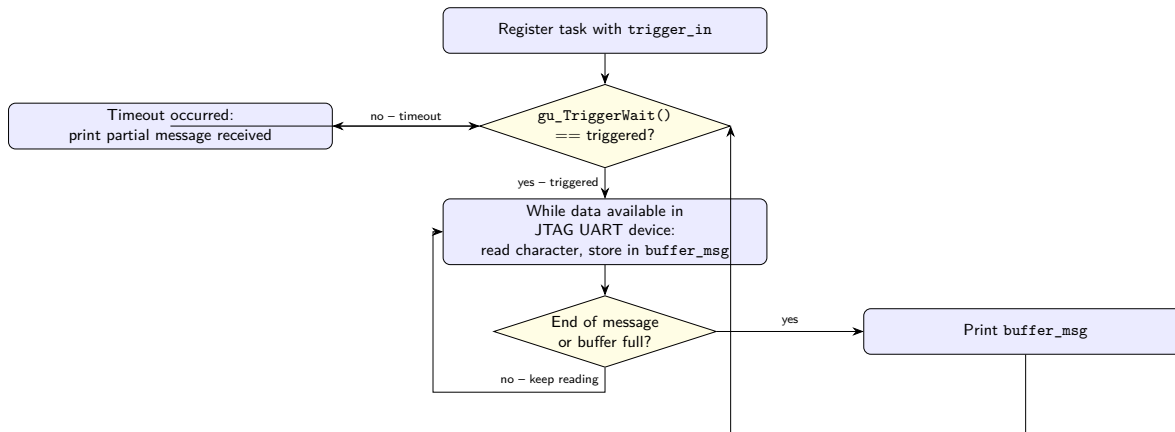


Figure 10.5: JTAG UART read template

Code 10.5 show the main functions of the receiving routine. Show verifying instructions where remove for the sake of simplicity. The task receives the the configuration information by pointing to a `jtag_uart_info` structure. This structure is initialized in the main routine, and the pointer is passed to the task when it is created.

```

void GeMRTOS_IRQ_READ_SERVER(void *pdata)
{
    /* Point to the structure with the JTAG_UART information */
    struct jtag_uart_info * jtag_uart_0_info = (struct jtag_uart_info *) pdata;

    unsigned int index_msg = 0;
    unsigned int uart_data;

    /* Create a local buffer to hold the messages */
    char *buffer_msg = (char *) malloc(jtag_uart_0_info->buffer_length_in);

    /* Get the pointer to the TCB of the current \gls{task} */
    GS_TCB *ptcb = gu_TaskGetCurrentTCB();

    /* Define this \gls{task} as an ISR associated to trigger_in of jtag_uart_0 */
    gu_TriggerRegisterTask(ptcb, jtag_uart_0_info->trigger_in);

    /* Infinite-loop */
    while (1) {

```

```

/* wait for trigger to read the JTAG UART input data */
if (gu_TriggerWait() != G_TRGStatus_triggered) {
    /* There was a timeout */
    gu_fprintf("Read timeout in JTAG UART\n");
    if (index_msg != 0) { /* Print the characters received */
        buffer_msg[index_msg] = '\0'; // end of string
        gu_fprintf("\nINPUT FROM JTAG-UART = %s\n", buffer_msg);
        index_msg = 0;
    }
}
else
{
    /* while there exist valid data in JTAG UART device */
    while ((uart_data = IORD_ALTERA_AVALON_JTAG_UART_DATA(JTAG_UART_O_BASE)) &
        ALTERA_AVALON_JTAG_UART_DATA_RVALID_MSK) {

        /* Read the data in the JTAG UART and put it as char */
        char data = (char) ((uart_data & ALTERA_AVALON_JTAG_UART_DATA_DATA_MSK) >>
            ALTERA_AVALON_JTAG_UART_DATA_DATA_OFST);

        /* Store the char as next char received */
        buffer_msg[index_msg++] = data;

        /* if end of message is received or buffer is near full, print the message in
            stderr */
        if ((data == '\r') || (data == '\n') || (index_msg >=
            jtag_uart_0_info->buffer_length_in - 2)) {
            buffer_msg[index_msg] = '\0'; // end of string

            /* Print in stderr the message received */
            gu_fprintf("\nINPUT FROM JTAG-UART = %s\n", buffer_msg);

            index_msg = 0;
        }
    }
}
}
}
}
}
}

```

Code 10.5: JTAG UART read routine

The receiving routine should be triggering each time that the ISR of the JTAG UART (Figure 10.3) detects that the read interrupt of the JTAG UART device is active. When the reading routine is triggered, the JTAG UART read interrupt should be disabled to avoid reentrancy. Consistently, the reading interrupt should be enabled when the reading routine trigger is re-enabled when all the tasks registered to it are waiting for the next event in the trigger. Trigger hooks are proposed for easy implementation of this mechanism.

A trigger executes the enabling hook when the trigger is enabled (all registered tasks wait for the trigger) , and the disabling hook when then trigger is disabled (when trigger is released). Functions `gu_TriggerEnableHook` and `gu_TriggerDisableHook` functions are used to define the functions to be executed when trigger is enabled and disabled, respectively. Trigger hook codes are executed in kernel model to avoid race conditions.

Code 10.6 shows the enable hooks of the reading routine. It executes the enabling of the read interrupt of the JTAG UART.

```
void *GeMRTOS_IRQ_READ_SERVER_ENABLE(void *pdata)
{
    // Enable JTAG UART read interrupt
    unsigned int status = IORD_ALTERA_AVALON_JTAG_UART_CONTROL(JTAG_UART_O_BASE);
    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(JTAG_UART_O_BASE, status |
        ALTERA_AVALON_JTAG_UART_CONTROL_RE_MSK);

    return ((void *) 0);
}
```

Code 10.6: JTAG UART read interrupt enable hook

Code 10.7 is similar, it is the disabling hook executed to disable the read interrupt of the JTAG UART when trigger is released.

```
void *GeMRTOS_IRQ_READ_SERVER_DISABLE(void *pdata)
{
    // Disable JTAG UART read interrupt
    unsigned int status = IORD_ALTERA_AVALON_JTAG_UART_CONTROL(JTAG_UART_O_BASE);
    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(JTAG_UART_O_BASE, status &
        ~(ALTERA_AVALON_JTAG_UART_CONTROL_RE_MSK));

    return ((void *) 0);
}
```

Code 10.7: JTAG UART read interrupt enable hook

By defining the read interrupt enable and disable hooks, the JTAG UART is properly enable and disable for reading preserving coherency with a very efficient and flexibly mechanism. Several tasks may be register with the trigger event for JTAG UART reading preserving the mechanism for enabling and disabling the read interrupt that share the same IRQ with the write interrupt.

Writing to JTAG UART using Trigger and Message Queue :

A writing routine should be executed each time that it is wanted to transmit a message through the JTAG UART device. Traditional RTOSs, use synchronization mechanism to synchronize the reception of the message to transmit and the ISR to manage the write FIFO of the JTAG UART device.

GeMRTOS trigger allows implementation of access to the message and the management of the write JTAG UART interrupt in the same infinite cycle as shown in Figure 10.6.

The writing JTAG UART routine waits for a message from the Message Queue. When message is received, the routine fills the write FIFO of the JTAG UART. If write FIFO is full and characters remain in the message to be sent, then the routine suspends itself waiting for the trigger of write interrupt signaling more space is available in the write FIFO of the JTAG UART device. When all the characters of the message are written in the write FIFO of the JTAG UART device, then the routine suspends itself waiting for a new message from the Message Queue to be sent.

Code 10.8 implements the writing routine of message from the queue_out Message Queue. Some verification instructions where remove for a sake of easy understanding.

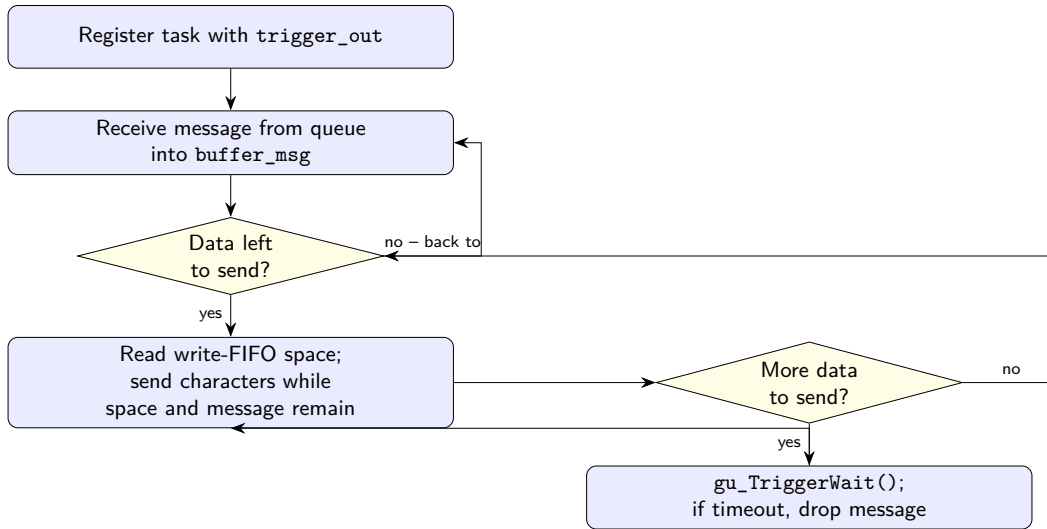


Figure 10.6: JTAG UART write template

```

void GeMRTOS_JTAG_UART_WRITE_SERVER(void *pdata)
{
    struct jtag_uart_info * jtag_uart_0_info = (struct jtag_uart_info *) pdata;

    /* Get a buffer to store the message to send to the jtag-uart */
    char *buffer_msg = (char *) malloc(jtag_uart_0_info->buffer_length_out);

    int size;
    int index_msg;

    /* Get the pointer to the TCB of the current \gls{task} */
    GS_TCB *ptcb = gu_TaskGetCurrentTCB();

    /* subscribe \gls{task} as receiver to the pqueue_out message queue */
    gu_MessageQueueSubscribe(jtag_uart_0_info->pqueue_out, (void *) buffer_msg,
        jtag_uart_0_info->buffer_length_out);

    /* Define this \gls{task} as an ISR associated to trigger_out jtag_uart_0_out */
    gu_TriggerRegisterTask(ptcb, jtag_uart_0_info->trigger_out);

    while (1) {
        /* receive next message from queue_out Message Queue */
        size = gu_MessageQueueReceive(jtag_uart_0_info->pqueue_out);

        /* Iterate for each character of the message, but the ending '\0' */
        index_msg = 0;
        while (index_msg < size-1)
        {
            // int space_fifo;
            /* Get the available space in jtag uart buffer */
            int space_fifo = (((G_UINT32)
                IORD_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base)) &

```

```

((G_UINT32) ALTERA_AVALON_JTAG_UART_CONTROL_WSPACE_MSK )) >>
ALTERA_AVALON_JTAG_UART_CONTROL_WSPACE_OFST;

// if (space_fifo > size) space_fifo = size - 2;
/* Send as many characters up to the end of message or as possible to jtag uart
*/
while (space_fifo > 0 && index_msg < size-1)
{
    /* write next message character to jtag uart output fifo */
    IOWR_ALTERA_AVALON_JTAG_UART_DATA(jtag_uart_0_info->jtag_base,
        buffer_msg[index_msg++]);

    /* Update iteration variables */
    space_fifo--;
}
/* There exist more characters to send, so wait to next trigger_out from JTAG
UART device */
if (index_msg < size-1) {
    if (gu_TriggerWait() != G_TRGStatus_triggered) {
        size = 0; // Timeout happens, remove message;
    }
}
}
}
}
}

```

Code 10.8: JTAG UART write interrupt routine

The enabling and disabling of the JTAG UART write interrupt is performed using the trigger hooks in similar way as done for read interrupt. Codes 10.9 and 10.10 shows the enable and disable trigger hook routines for enabling and disabling, respectively, the write interrupt of the JTAG UART device.

```

void *GeMRTOS_JTAG_UART_WRITE_SERVER_ENABLE(void * pdata)
{
    // Enable interrupt in JTAG UART component
    unsigned int status = IORD_ALTERA_AVALON_JTAG_UART_CONTROL(JTAG_UART_0_BASE);
    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(JTAG_UART_0_BASE, status |
        ALTERA_AVALON_JTAG_UART_CONTROL_WE_MSK);

    return ((void *) 0);
}

```

Code 10.9: JTAG UART write interrupt enable hook

```

void *GeMRTOS_JTAG_UART_WRITE_SERVER_DISABLE(void *pdata)
{
    // Disable interrupt in JTAG UART component
    unsigned int status = IORD_ALTERA_AVALON_JTAG_UART_CONTROL(JTAG_UART_0_BASE);
    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(JTAG_UART_0_BASE, status &
        ~(ALTERA_AVALON_JTAG_UART_CONTROL_WE_MSK));

    return ((void *) 0);
}

```

}

Code 10.10: JTAG UART write interrupt disable hook

The trigger hooks allow enabling and disabling the write interrupt of the JTAG UART when the trigger is enabled (all registered tasks are waiting for the event), and when task is released, respectively.

As can be noted, the flow of the write routine for JTAG UART device is streamlining according to the flow of getting the message and writing into the write FIFO. No complicated synchronization mechanism is required to deal with FIFO size, output interrupt and timeouts. The writing routine contains all the write flow required to manage efficiently the JTAG UART writing mechanism.

The GeMRTOS trigger efficiency may be applied to diverse use cases to improve the efficiency and ease of resource management.

The JTAG UART service proposed requires its configuration in the main routine of the application. Figure 10.7 shows a layout of the triggering mechanism to manage the JTAG UART.

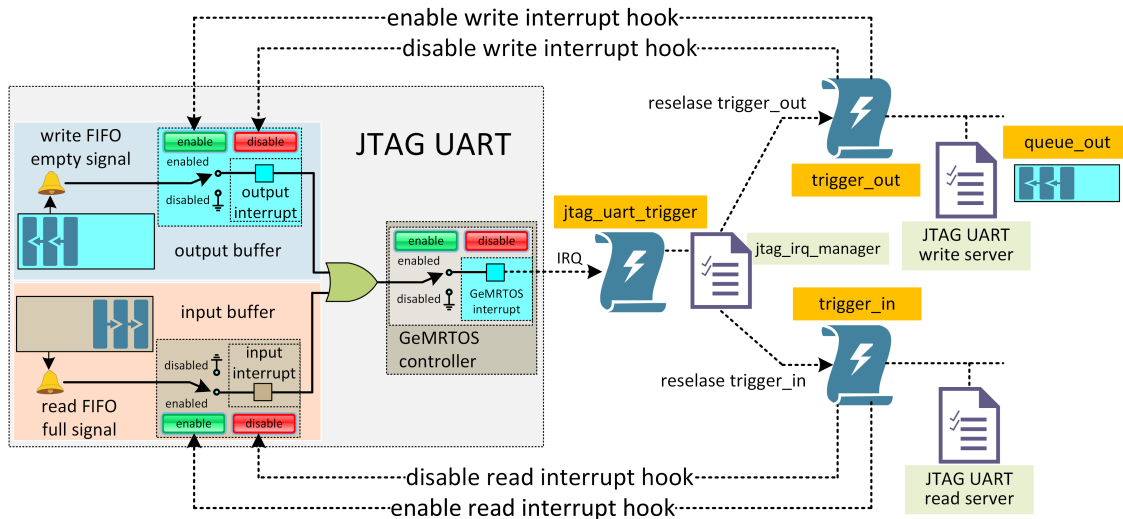


Figure 10.7: JTAG UART configuration layout

The Figure 10.8 shows a structure to hold all the information required for the architecture of Figure 10.7.

```
struct jtag_uart_info {
    GS_RCB *trigger_out;
    GS_RCB *trigger_in;
    GS_RCB *pqueue_out;
    GS_RCB *pqueue_in;
    unsigned int buffer_length_out;
    unsigned int buffer_length_in;
    unsigned int jtag_base;
};
```

Figure 10.8: JTAG UART configuration structure

The `jtag_uart_0_info` global variable is the pointer to the structure that holds all the layout architecture to manage the JTAG UART device:

```
struct jtag_uart_info *jtag_uart_0_info;
```

Creating the `jtag_uart_info` structure in the main routine of the application:

```
jtag_uart_0_info = (struct jtag_uart_info *) malloc(sizeof(struct jtag_uart_info));
```

The `jtag_uart_0_info` structure is then initialized to store the layout information of the JTAG UART managing architecture in its fields:

- **trigger_out**
it is the handle to the trigger for the write task. When the JTAG UART write FIFO has space to accept additional data, the IRQ is issued, the `jtag_uart_trigger` issued, the `jtag_irq_manager` task released, and it releases the `trigger_out` trigger because the write interrupt is active in the JTAG UART status register. The trigger resource is created without associating it with an external interrupt (it is triggered by the task), and without timeout (passing 0 as timeout argument to `gu_TriggerCreate` function):

```
jtag_uart_0_info->trigger_out = gu_TriggerCreate(-1, 0);
```

The write server function should register to `jtag_uart_0_info->trigger_out` to be executed each time that the `trigger_out` resource is released.
- **trigger_in**
it is the handle to the trigger for the read task. When the JTAG UART read FIFO has received data, the IRQ is issued, the `jtag_uart_trigger` issued, the `jtag_irq_manager` task released, and it releases the `trigger_in` trigger because the read interrupt is active in the JTAG UART status register. The trigger resource is created without associating it with an external interrupt (it is triggered by the task), and with a timeout equal to 100ms :

```
G_UINT64 timeout_trigger_in = gu_ConvertTime(0, 0, 0, 100);
jtag_uart_0_info->trigger_in = gu_TriggerCreate(-1, timeout_trigger_in);
```

The read server function should register to `jtag_uart_0_info->trigger_in` to be executed each time that the `trigger_in` resource is released.
- **pqueue_out**
it is the handle to the message queue for the write server. The write server waits until a message is received in the message queue to write it to the write FIFO of the JTAG UART device. The `pqueue_out` message queue is created in the main routine:

```
jtag_uart_0_info->pqueue_out = gu_MessageQueueCreate();
```

The message queue may be used by several tasks to send messages to the write server to be transmitted through the JTAG UART device.
- **pqueue_in**
it is the handle to the message queue for the read server. The read server sends the messages received in the read FIFO to this message queue in order to be delivered to all the tasks that register to the queue to read the messages. If no task registered to the message queue, then the message is removed. The `pqueue_out` message queue is created in the main routine:

```
jtag_uart_0_info->pqueue_in = gu_MessageQueueCreate();
```

By using the `pqueue_in` message queue, the message can be delivered to as many task as it is required.
- **buffer_length_in**
defines the length of the read buffer of the read server. Messages will be sent to the message queue when an end of message is received (`'\r'` or `'\n'` characters), when the read timeout is issued, or when the length of this buffer is reached. For instance, a buffer of 100 bytes is defined with:

```
jtag_uart_0_info->buffer_length_in = 100;
```

- `buffer_length_out`
defines the length of the write buffer of the write server. Messages will be sent to the write FIFO of the JTAG UART device when and end of message is received ('\\r' or '\\n' characters), or when the length of this buffer is reached. For instance, a buffer of 100 bytes is defined with:
`jtag_uart_0_info->buffer_length_out = 100;`
- `jtag_base`
It is the base address of the JTAG UART device in the system. It is defined in the `system.h` file produced by Platform Designer. Because, the JTAG UART device was named `JTAG_UART_0`, the base address of the device is defined as `JTAG_UART_0_BASE`, and consequently:
`jtag_uart_0_info->jtag_base = JTAG_UART_0_BASE;`
The `jtag_uart_0_info` structure holds all the information for the JTAG UART managing architecture shown in Figure 10.7. The pointer to this structure should be passed as argument to all the tasks involved in this architecture for clean management.

JTAG UART architecture code

GeMRTOS allows implementing the whole JTAG UART management architecture of Figure 10.7 in a very efficient, scalable and modular way.

The `jtag_irq_manager` task The `jtag_uart_manager` task (code 10.11) is released each time that the JTAG UART issues an IRQ signal (either because read interrupt, write interrupt, or both).

```
void jtag_irq_manager(void *pjtag_info)
{
    struct jtag_uart_info *jtaguartinfo = (struct jtag_uart_info *) pjtag_info;

    /* Read the state of interrupt flags from the JTAG UART status register */
    volatile G_UINT32 uart_ctrl_register = (G_UINT32)
        IORD_ALTERA_AVALON_JTAG_UART_CONTROL(jtaguartinfo->jtag_base);

    /* Get the state of the write interrupt (output buffer empty) */
    volatile G_UINT32 write_interrupt = (uart_ctrl_register & ((G_UINT32)
        ALTERA_AVALON_JTAG_UART_CONTROL_WI_MSK )) >>
        ALTERA_AVALON_JTAG_UART_CONTROL_WI_OFST;

    /* Get the state of the read interrupt (input buffer full) */
    volatile G_UINT32 read_interrupt = (uart_ctrl_register & ((G_UINT32)
        ALTERA_AVALON_JTAG_UART_CONTROL_RI_MSK )) >>
        ALTERA_AVALON_JTAG_UART_CONTROL_RI_OFST;

    /* If write interrupt is active, release the jtag_uart_0_out trigger */
    if ((write_interrupt == (G_UINT32) 1)) {
        gu_TriggerRelease (jtaguartinfo->trigger_out);
    }

    /* If read interrupt is active, release the jtag_uart_0_in trigger_in */
    if ((read_interrupt == (G_UINT32) 1)) {
        gu_TriggerRelease (jtaguartinfo->trigger_in);
    }
}
```

Code 10.11: jtag_uart_manager function code

The task received as argument the pointer to the `jtag_uart_info` of the `JTAG_UART_0` device. It is executed each time that the IRQ signal of the JTAG UART device is issued. It reads the control register (in the `jtaguartinfo->jtag_base` address memory) and checks whether the read interrupt or the write interrupt are active. The `jtaguartinfo->trigger_in` trigger is released when read interrupt is active, whilst the `jtaguartinfo->trigger_out` trigger is released when write interrupt is active.

The write server task The `GeMRTOS_JTAG_UART_WRITE_SERVER` task (code 10.12) is released each time that the `jtag_uart_manager` task checks that the write interrupt is asserted in the JTAG UART device.

```
void GeMRTOS_JTAG_UART_WRITE_SERVER(void *pjtag_info)
{
    /* Point to the JTAG UART information structure */
    struct jtag_uart_info * jtag_uart_0_info = (struct jtag_uart_info *) pjtag_info;

    /* Get a buffer to store the message to send to the jtag-uart */
    char *buffer_msg = (char *) malloc(jtag_uart_0_info->buffer_length_out);

    int size;
    int index_msg;

    /* Get the pointer to the TCB of the current \gls{task} */
    GS_TCB *ptcb = gu_TaskGetCurrentTCB();

    /* suscribe \gls{task} as receiver to the pqueue_out message queue */
    gu_MessageQueueSubscribe(jtag_uart_0_info->pqueue_out, (void *) buffer_msg,
        jtag_uart_0_info->buffer_length_out);

    /* Define this \gls{task} as an ISR associated to trigger_out jtag_uart_0_info */
    gu_TriggerRegisterTask(ptcb, jtag_uart_0_info->trigger_out);

    while (1) {
        /* receive next message from queue_out Message Queue */
        size = gu_MessageQueueReceive(jtag_uart_0_info->pqueue_out);

        /* Iterate for each character of the message */
        index_msg = 0;
        while (index_msg < size-1)
        {
            /* Get the available space in jtag uart write FIFO */
            int space_fifo = (((G_UINT32)
                IORD_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base)) &
                ((G_UINT32) ALTERA_AVALON_JTAG_UART_CONTROL_WSPACE_MSK )) >>
                ALTERA_AVALON_JTAG_UART_CONTROL_WSPACE_OFST;

            /* Send as many characters up to the end of message or as possible for jtag
                uart space */
            while (space_fifo > 0 && index_msg < size-1)
            {
```

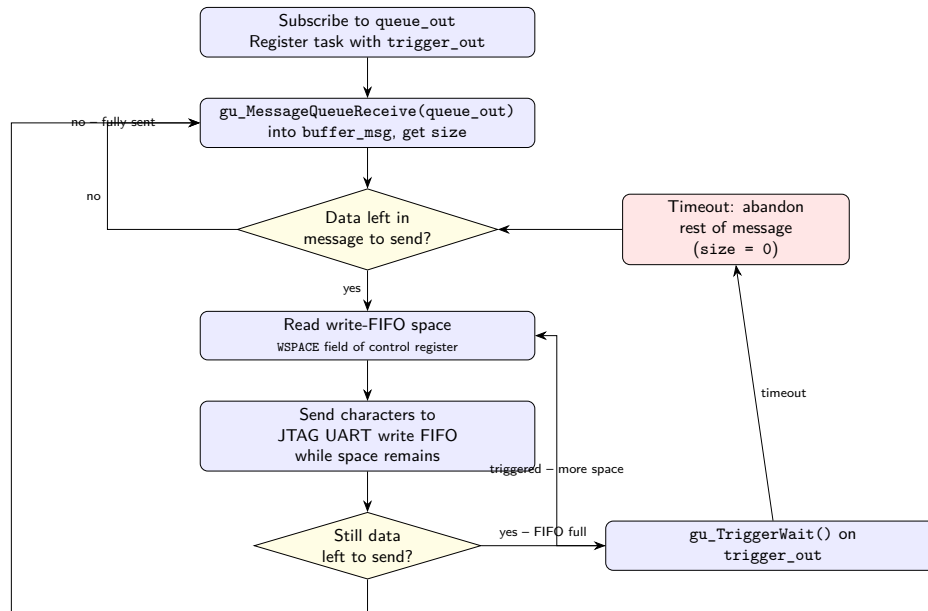


Figure 10.9: Flow of the JTAG UART write server

```

/* write next message character to jtag uart output fifo */
IOWR_ALTERA_AVALON_JTAG_UART_DATA(jtag_uart_0_info->jtag_base,
    buffer_msg[index_msg++]);

/* Update iteration variables */
space_fifo--;
}

/* There exist more characters to send, so wait to next trigger_out from JTAG
   UART device */
if (index_msg < size-1) {
    if (gu_TriggerWait() != G_TRGStatus_triggered) {
        size = 0; // Timeout happens, remove message;
    }
}
}
}
}
}
}

```

Code 10.12: write server function code

The Figure 10.9 shows the flow of the JTAG UART write server task. The task suspends until a message is received in the Message Queue `queue_out`, in which it was previously registered. Then the message is written character by character into the write FIFO of the JTAG UART device until the end of the message or until there is not more space in the write FIFO. If the write FIFO is full and more message characters remain, then the task waits for the `trigger_out` to be released indicating the there is more space into de write FIFO of the JTAG UART device. If the timeout of the `trigger_out` is issued, in this case as to show an alternative of action, the message is removed because it indicates the communication link is down and the task should continue with the rest of messages.

GeMRTOS triggers allows implementing timed events in a very clean and neat structure making it easy to understand and implement. What remains is to manage the enabling and disabling of the write interrupt of the JTAG UART device by defining the enabling and disabling hook functions, respectively:

```
void GeMRTOS_JTAG_UART_WRITE_SERVER_ENABLE(void * pntag_info)
{
    /* Point to the JTAG UART information structure */
    struct jtag_uart_info * jtag_uart_0_info = (struct jtag_uart_info *) pntag_info;

    /* Enable interrupt in JTAG UART component */
    unsigned int status =
        IORD_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base);

    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base, status |
        ALTERA_AVALON_JTAG_UART_CONTROL_WE_MSK);
}
```

Code 10.13: write server enable hook function.

```
void GeMRTOS_JTAG_UART_WRITE_SERVER_DISABLE(void *pntag_info)
{
    /* Point to the JTAG UART information structure */
    struct jtag_uart_info * jtag_uart_0_info = (struct jtag_uart_info *) pntag_info;

    /* Disable interrupt in JTAG UART component */
    unsigned int status =
        IORD_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base);

    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base, status &
        ~(ALTERA_AVALON_JTAG_UART_CONTROL_WE_MSK));
}
```

Code 10.14: write server disable hook function.

Codes 10.13 and 10.14 executes the JTAG UART instructions to enable and disable the write interrupt of the JTAG UART, respectively. The enable hook function is configured in the main function of the application to be executed when the trigger_out trigger is enabled when all the tasks registered to the trigger_out trigger are waiting for the trigger. The disabled hook function is configured in the main function of the application, and it is executed when the trigger_out trigger is released.

Multiple tasks can be registered to the trigger_in signal, enabling processing of trigger events for various purposes. However, the enable and disable hook functions will be executed reliably to ensure consistent system behavior, maintaining proper coordination so that all tasks are either waiting for or released from the trigger in a coherent manner.

The read server task The GeMRTOS_JTAG_UART_READ_SERVER task (code 10.15) is released each time that the jtag_uart_manager task checks that the read interrupt is asserted in the JTAG UART device.

```
void GeMRTOS_JTAG_UART_READ_SERVER(void *pdata)
{
    /* Point to the structure with the JTAG_UART information */
    struct jtag_uart_info * jtag_uart_0_info = (struct jtag_uart_info *) pdata;
```

```

unsigned int index_msg = 0;
unsigned int uart_data;

/* Create a local buffer to hold the messages */
char *buffer_msg = (char *) malloc(jtag_uart_0_info->buffer_length_in);

/* Get the pointer to the TCB of the current \gls{task} */
GS_TCB *ptcb = gu_TaskGetCurrentTCB();

/* Define this \gls{task} as an ISR associated to trigger_in of jtag_uart_0 */
gu_TriggerRegisterTask(ptcb, jtag_uart_0_info->trigger_in);

/* Infinite-loop */
while (1) {
    /* wait for trigger to read the JTAG UART input data */
    if (gu_TriggerWait() != G_TRGStatus_triggered) {
        /* There was a timeout */
        if (index_msg != 0) { /* Print the characters received */
            buffer_msg[index_msg] = '\0'; // end of string
            /* Print in stderr the message received */
            gu_fprintf("\nINPUT FROM JTAG-UART = %s\n", buffer_msg);
            index_msg = 0;
        }
    }
    else
    {
        /* while there exist valid data in JTAG UART read FIFO */
        while ((uart_data = IORD_ALTERA_AVALON_JTAG_UART_DATA(JTAG_UART_0_BASE)) &
            ALTERA_AVALON_JTAG_UART_DATA_RVALID_MSK) {

            /* Read the data in the JTAG UART and put it as char */
            char data = (char) ((uart_data & ALTERA_AVALON_JTAG_UART_DATA_DATA_MSK) >>
                ALTERA_AVALON_JTAG_UART_DATA_DATA_OFST);

            /* Store the char as next char received */
            buffer_msg[index_msg++] = data;

            /* if end of message is received or buffer is near full */
            /* print the message in stderr */
            if ((data == '\r') || (data == '\n') || (index_msg >=
                jtag_uart_0_info->buffer_length_in - 2)) {

                buffer_msg[index_msg] = '\0'; // end of string
                /* Print in stderr the message received */
                gu_fprintf("\nINPUT FROM JTAG-UART = %s\n", buffer_msg);
            }
        }
    }
}
}

```

Code 10.15: read server function code

The Figure 10.10 shows the flow of the JTAG UART read server tasks. The task suspends waiting for a trigger from the `jtag_uart_manager` when the read interrupt is active. It is resumed either when the timeout occurs or the `trigger_in` is released by the `jtag_uart_manager` function. If timeout occurs, the partial message is printed in `stderr`. If `trigger_in` is released, all the message is printed in `stderr`.

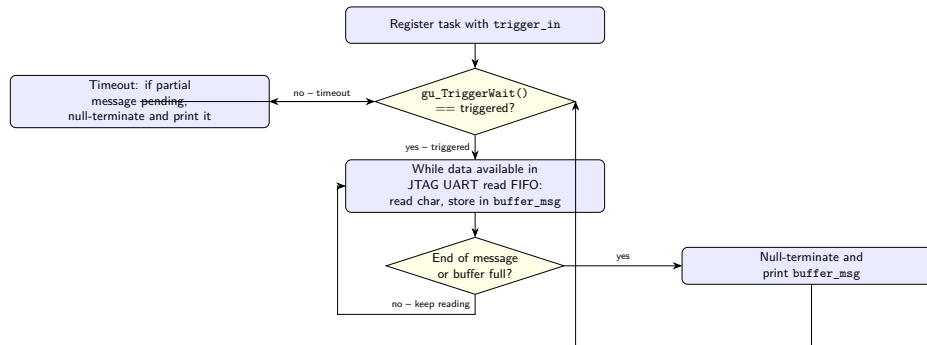


Figure 10.10: Flow of the JTAG UART read server

GeMRTOS triggers allows implementing timed events in a very clean and neat structure making it easy to understand and implement. What remains is to manage the enabling and disabling of the read interrupt of the JTAG UART device by defining the enabling and disabling hook functions, respectively.

```

void GeMRTOS_JTAG_UART_READ_SERVER_ENABLE(void * ptag_info)
{
    /* Point to the JTAG UART information structure */
    struct jtag_uart_info * jtag_uart_0_info = (struct jtag_uart_info *) ptag_info;

    /* Enable JTAG UART read interrupt */
    unsigned int status =
        IORD_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base);

    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base, status |
        ALTERA_AVALON_JTAG_UART_CONTROL_RE_MSK);
}

```

Code 10.16: read server enable hook function

```

void GeMRTOS_JTAG_UART_READ_SERVER_DISABLE(void * ptag_info)
{
    /* Point to the JTAG UART information structure */
    struct jtag_uart_info * jtag_uart_0_info = (struct jtag_uart_info *) ptag_info;

    /* Disable JTAG UART read interrupt */
    unsigned int status =
        IORD_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base);

    IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(jtag_uart_0_info->jtag_base, status &
        ~(ALTERA_AVALON_JTAG_UART_CONTROL_RE_MSK));
}

```

}

Code 10.17: read server disable hook function

Codes 10.16 and 10.17 executes the JTAG UART instructions to enable and disable the read interrupt of the JTAG UART, respectively. The enable hook function is configured in the main function of the application to be executed when the `trigger_in` trigger is enabled when all the tasks registered to the `trigger_in` trigger are waiting for the trigger. The disabled hook function is configured in the main function of the application, and it is executed when the `trigger_out` trigger is released.

Several tasks may be registered to the `trigger_in` signal allowing processing trigger events for different purposes. However, enable and disable hook functions will be executed consistently to proper consistency that all tasks are released or waiting for trigger consistently.

The main function The JTAG UART management architecture is configured in the main function of the application. The code 10.18 is an excerpt that summarizes the configuration of the management of the JTAG UART management architecture.

```
int main(void)
{
    . . .

    /* create a jtag_uart_info structure to store the information for output server
       consumer */
    jtag_uart_0_info = (struct jtag_uart_info *) malloc(sizeof(struct jtag_uart_info));

    /* Create a trigger_out for JTAG UART writting */
    /* -1 because it is not associated with hardware interrupt */
    jtag_uart_0_info->trigger_out = gu_TriggerCreate(-1, 0);

    /* Create a trigger_in for JTAG UART reading */
    /* -1 because it is not associated with hardware interrupt */
    G_UINT64 timeout_trigger_in = gu_ConvertTime(0, 0, 0, 100);
    jtag_uart_0_info->trigger_in = gu_TriggerCreate(-1, timeout_trigger_in);

    /* Create a message queue for JTAG-UART jtag_uart_0 output */
    jtag_uart_0_info->pqueue_out = gu_MessageQueueCreate();

    /* Create a message queue for JTAG-UART jtag_uart_0 input */
    jtag_uart_0_info->pqueue_in = gu_MessageQueueCreate();

    jtag_uart_0_info->buffer_length_in = 100;
    jtag_uart_0_info->buffer_length_out = 100;
    jtag_uart_0_info->jtag_base = JTAG_UART_0_BASE;

    /* Create \gls{task} for jtag output server */
    ptcb = gu_TaskCreate((void *) GeMRTOS_JTAG_UART_WRITE_SERVER,
                        (void *) jtag_uart_0_info,
                        "GeMRTOS_JTAG_UART_WRITE_SERVER");
    if (ptcb != (void *) 0)
    {
```

```

    gu_TaskTypeSet(ptcb, G_TCBType_OneShot);

    gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 0);
    gu_TaskRunPrioritySet(ptcb, (G_UINT64) 0);

    /* Start execution of \gls{task} with an offset */
    gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0);
}

/* Set the jtag write server hook functions */
gu_TriggerEnableHook(jtag_uart_0_info->trigger_out,
    GeMRTOS_JTAG_UART_WRITE_SERVER_ENABLE, (void *) jtag_uart_0_info);
gu_TriggerDisableHook(jtag_uart_0_info->trigger_out,
    GeMRTOS_JTAG_UART_WRITE_SERVER_DISABLE, (void *) jtag_uart_0_info);

/* Create \gls{task} for jtag output server */
ptcb = gu_TaskCreate((void *) GeMRTOS_JTAG_UART_READ_SERVER,
    (void *) jtag_uart_0_info,
    "GeMRTOS_JTAG_UART_READ_SERVER");
if (ptcb != (void *) 0)
{
    gu_TaskTypeSet(ptcb, G_TCBType_OneShot);

    gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 0);
    gu_TaskRunPrioritySet(ptcb, (G_UINT64) 0);

    /* Start execution of \gls{task} with an offset */
    gu_TaskStartWithOffset(ptcb, 0, 0, 5, 0);
}

/* Set the jtag output server hooks */
gu_TriggerEnableHook(jtag_uart_0_info->trigger_in,
    GeMRTOS_JTAG_UART_READ_SERVER_ENABLE, (void *) jtag_uart_0_info);
gu_TriggerDisableHook(jtag_uart_0_info->trigger_in,
    GeMRTOS_JTAG_UART_READ_SERVER_DISABLE, (void *) jtag_uart_0_info);

/* Create an ISR \gls{task} for jtag uart hardware interrupt */
ptcb = gu_TaskCreate((void *) jtag_irq_manager,
    (void *) jtag_uart_0_info,
    "jtag_irq_manager");

if (ptcb != (void *) 0)
{
    gu_TaskTypeSet(ptcb, G_TCBType_OneShot);
    gu_TaskReadyPrioritySet(ptcb, (G_UINT64) 0);
    gu_TaskRunPrioritySet(ptcb, (G_UINT64) 0);
}

/* Associate the \gls{task} to be triggered with the jtag uart IRQ interrupt */
GS_RCB *jtag_uart_trigger = gu_TriggerCreate(JTAG_UART_0_IRQ, 0);

```

```
/* Register the jtag_uart_manager function to JTAG UART IRQ */
gu_TriggerRegisterTask(ptcb, jtag_uart_trigger);

/* Start system with JTAG UART IRQ enabled */
gu_TriggerEnable(jtag_uart_trigger);

. . .
return (0);
}
```

Code 10.18: JTAG UART management configuration in main function

In code 10.18, the `jtag_uart_0_info` structure is initialized, the tasks created, the hook function configured and the triggers set. With this configuration, runtime behavior is established.

This example demonstrates the creation of a JTAG UART architecture, showcasing how GeMRTOS Triggers enable efficient mechanisms for event handling and timeouts. The JTAG UART shares the same IRQ signal for both write and read interrupts. Sophisticated synchronization is required when implemented with traditional operating systems to manage these interrupts for different purposes and with timeouts. However, GeMRTOS provides a neat approach, managing events and timeouts in the same task flow without the need for complex synchronization mechanisms that may be difficult, and prone to failures that require complex debugging techniques.

10.5 Example 4: Ethernet server with LwIP with TSE

In modern embedded system design, the integration of an Ethernet interface and IP-based communication is of paramount importance, as it enables seamless connectivity, interoperability, and remote accessibility. Ethernet provides a robust and standardized physical layer for high-speed data transfer, while IP communication ensures compatibility with existing network infrastructures and internet-based services. This combination allows embedded devices to exchange information efficiently, support distributed architectures, and be integrated into larger systems such as industrial automation, IoT ecosystems, and real-time monitoring platforms. Furthermore, the use of Ethernet and IP protocols enhances scalability and maintainability, offering designers a reliable foundation for implementing secure, flexible, and future-proof-embedded solutions.

The Altera TSE IP core provides a flexible and efficient solution for implementing Ethernet communication in FPGA-based embedded systems. It supports three standard data rates—10 Mbps, 100 Mbps, and 1 Gbps—allowing designers to adapt system performance to the requirements of diverse applications. The TSE IP core integrates seamlessly with MAC and PHY layers, offering a reliable hardware implementation of Ethernet protocols while reducing development time and complexity. Its configurability and scalability make it suitable for a wide range of designs, from low-speed control networks to high-throughput data transmission systems, ensuring robust connectivity and compliance with industry standards.

The Lightweight IP (LwIP) stack is an open-source implementation of the TCP/IP protocol suite specifically designed for embedded systems with limited resources. Its compact architecture and reduced memory footprint make it highly suitable for microcontrollers and FPGA-based platforms where efficiency is critical. LwIP enables embedded devices to support essential networking capabilities such as TCP, UDP, and IP communication, facilitating integration with Ethernet and other network interfaces. By providing a standards-compliant yet resource-efficient solution, LwIP allows designers to implement internet connectivity, remote monitoring, and distributed control in embedded applications without the overhead of full-scale networking stacks.

In this example, the implementation of a multiprocessor system using GeMRTOS with IP-based communication will be described, combining the Lightweight IP (LwIP) stack and the Altera TSE IP core. LwIP provides an efficient and resource-aware TCP/IP stack for enabling network connectivity, while the TSE core ensures reliable Ethernet communication across multiple data rates. By integrating these components within GeMRTOS, the design demonstrates how real-time scheduling and multiprocessor management can be seamlessly combined with networking capabilities. The example ultimately highlights the ease and flexibility of implementation achieved when using GeMRTOS as the underlying real-time operating system.

The TSE IP core

The TSE IP core is integrated into the system design using Platform Designer, where it is configured to operate with two Scatter-Gather DMA controllers: one dedicated to transmission and the other to reception. This configuration allows the TSE core to efficiently manage data flow in both directions, offloading the processor from intensive data transfer tasks and ensuring smooth, high-throughput communication.

To support this architecture, two on-chip memory blocks should be included in the design. The first memory is reserved for storing communication descriptors, which define the structure and organization of the data packets. The second memory is allocated for the transfer and storage of the actual messages to be transmitted or received. This division ensures a clear separation of control information and payload data, simplifying management and improving performance under real-time operating conditions.

It is crucial that both memories are implemented on-chip, as this guarantees the required bandwidth to sustain communication at speeds of up to 1 Gbps. Off-chip memory access could introduce delays and bottlenecks, thereby compromising throughput and system reliability. By maintaining these memories within the FPGA, the design achieves the necessary low-latency response and deterministic behavior demanded by high-speed Ethernet communication.

Figure 10.11 illustrates the interconnection of all the components involved, including the interrupts generated by the scatter-gather DMA controllers and their integration with GeMRTOS. The architecture remains identical to that of a uniprocessor system; the only difference is that all GeMRTOS processors are connected to the shared peripherals of the design. This approach preserves architectural consistency while enabling multiprocessor scalability, ensuring that communication, memory access, and interrupt handling are seamlessly extended across the entire system. It is important to note that this integration has been correctly implemented in the delivered project, which includes the complete example for reference.

The development of the remainder of the system follows the same methodology as any other embedded application designed in Platform Designer. The integration of peripherals, interconnects, and memory components adheres to the standard design flow provided by the tool, ensuring consistency and reliability. This similarity allows developers to reuse established practices and design strategies, reducing complexity and facilitating the implementation of the complete system within a familiar framework.

10.5.1 TSE initialization for LwIP

The TSE core requires configuration for managing different speed modes. Code 10.19 shows how the TSE core is initialized with the information of the `gemrtos_tse_mac_device`.

```
int tse_mac_init(struct netif *netif)
{
    int x;
```

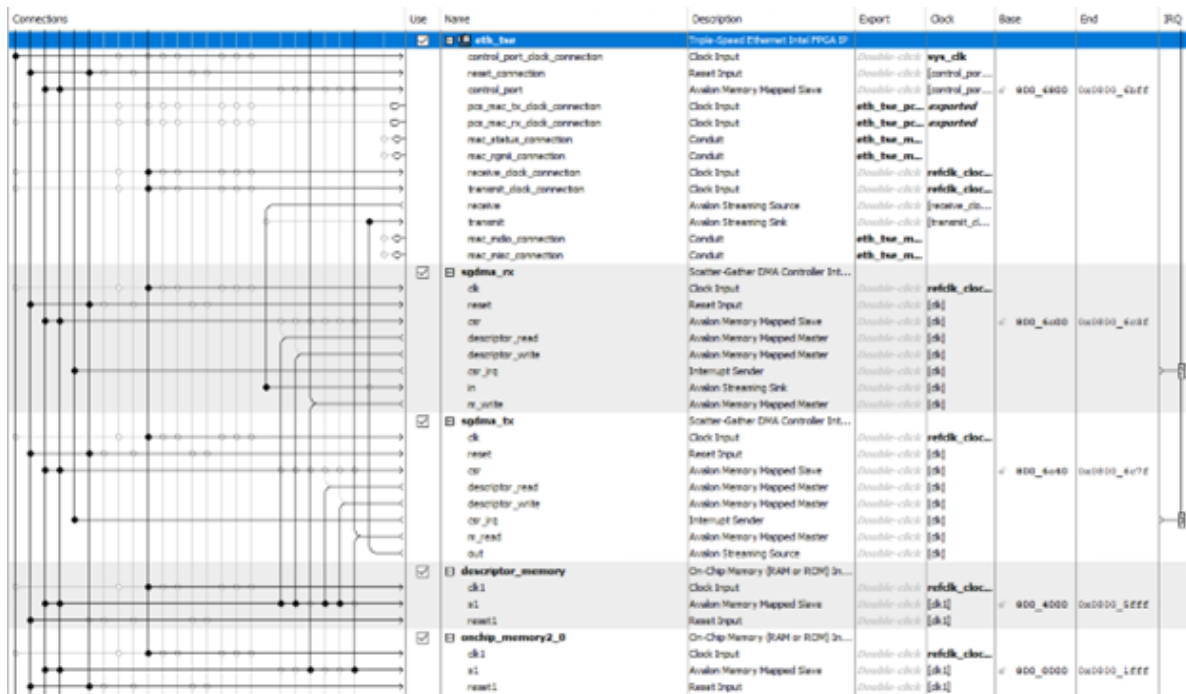


Figure 10.11: TSE Platform Designer configuration

```

gemrtos_tse_system_info *gemrtos_tse = gemrtos_GetTSEInfo (netif,
    (gemrtos_tse_system_info *) &gemrtos_tse_mac_device);

// Triple-speed Ethernet MegaCore base address
int * tse_phy = (int *) gemrtos_tse->tse_mac_base;
// MDIO PHY space in TSE
unsigned int phy_mdio_space = gemrtos_tse->phy_tse_mdio_register_space;

/* reset the mac and wait until reset is gone*/
*(tse_phy + 0x02) = ALTERA_TSEMAC_CMD_SW_RESET_MSK | ALTERA_TSEMAC_CMD_TX_ENA_MSK
    | ALTERA_TSEMAC_CMD_RX_ENA_MSK;

for (x=0; ((*tse_phy + 0x02) & ALTERA_TSEMAC_CMD_SW_RESET_MSK) && (x < 10000));
    x++);
if (x >= 10000) GU_FPRINTF(stderr, "TSEMAC SW reset bit never cleared!\n");

while (((*(tse_phy + phy_mdio_space + 1) & (unsigned int) pow(2,5))) == 0);

// Enable Transmit CRC generation ALTERA_TSEMAC_CMD_CRC_FWD_MSK
*(tse_phy + 0x3A) = *(tse_phy + 0x3A) & ~ALTERA_TSEMAC_TX_CMD_STAT_OMITCRC_MSK;

// Update the link information
if (gemrtos_UpdateTSEInfo (netif) == G_TRUE) {
    gemrtos_InitialiseTSE(netif); // Initialise TSE if info changed
}
return gemrtos_tse->link_alive;
}

```

Code 10.19: TSE initialization for TSE code

The TSE information is set in the `gemrtos_tse_system_info` structure:

```
gemrtos_tse_system_info gemrtos_tse_mac_device[MAXNETS] = {
    GEMRTOS_TSE_SYSTEM("en", ETH_TSE, 0, 0, SGDMA_TX, SGDMA_RX, 0, DESCRIPTOR_MEMORY)
};
```

This structure should be initiated with the parameters according to the physical connections of the board.

Integration LwIP in GeMRTOS

GeMRTOS automatically integrates the LwIP stack whenever the project includes a TSE component and the command `gemrtos_build` is executed. This automation simplifies the configuration process by ensuring that the networking stack is properly connected to the Ethernet interface without requiring manual intervention. Alternatively, the LwIP module can also be incorporated through BSP editing, following the same procedure available for any standard Platform Designer project. This dual approach provides both convenience and flexibility, allowing developers to choose between automated integration and manual configuration depending on their design needs.

The example proposed in this example demonstrates the use of LwIP in "bare-metal" (NOSYS) mode to illustrate the ease and flexibility that GeMRTOS provides for implementing adaptations with different software modules.

Figure 10.12 illustrates an architecture in which LwIP is integrated into GeMRTOS through two tasks: one responsible for receiving Ethernet packets (the `GeMRTOS_TSE_RECEIVE_SERVER` task), and another for transmitting packets to the Ethernet (the `GeMRTOS_TSE_TRANSMIT_SERVER` task).

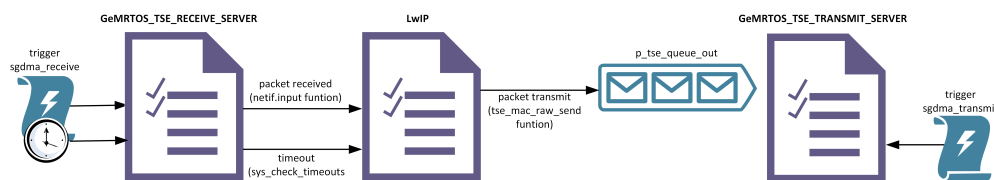


Figure 10.12: Integrating LwIP

The `GeMRTOS_TSE_RECEIVE_SERVER` task is in charge of receiving packets from the TSE IP core and send it to the LwIP executing the `netif.input` function. The task is also responsible for executing the `sys_check_timeouts` function every 250ms as it is specified for LwIP. All these requirements are easily implemented associating the `sdma_receive` trigger to the interrupt of the receiving SGDMA core.

LwIP processes the events produced either by the incoming packets or the timer ticks and produces outputs packets to be transmitted, executing the `tse_mac_raw_send` function. This function is in charge of pushing the packet to be transmitted into the `p_tse_queue_out` message queue.

The `GeMRTOS_TSE_TRANSMIT_SERVER` task receives the message pushed into the `p_tse_queue_out` message queue and send it to the TSE IP. The `GeMRTOS_TSE_TRANSMIT_SERVER` task suspends itself waiting for a message in the message queue and suspends itself until the message has been transmitted and the `sgdma_transmit` trigger interrupt indicating that the packet has been transmitted or an error was produced.

This architecture adapts easily and efficiently the LwIP requirements. Tasks are executed only when it is required and suspended when they are not necessary to be executed.

Known Limitations and Work-in-Progress Status

- **LwIP integration is a work in progress.** The HAL/arch/lwip_main.c file contains partial implementations marked with `!!!!D` (deferred) comments. The integration described in this section has been validated at the architecture level but has not been fully tested under all operating conditions. Users implementing production Ethernet with LwIP should treat this example as a reference template and perform end-to-end validation on their target hardware.
- **Frame buffer sizing.** The DMA descriptor buffer is sized at `TSE_DMA_FRAME_BUFFER_SIZE` (1700 bytes), which covers a standard 1500-byte Ethernet payload plus header, VLAN tag, and FCS. Jumbo frames are not supported. Received frames larger than `TSE_DMA_FRAME_BUFFER_SIZE` are silently truncated at the DMA level.
- **TX queue overflow protection.** `gu_MessageQueueSend` will reject frames whose `msg\_length` exceeds `TSE_TX_QUEUE_BUFFER_SIZE`. The TSE transmit server task logs an error and discards the oversized frame. Applications generating frames above this limit must reduce payload size or increase `TSE_TX_QUEUE_BUFFER_SIZE` and rebuild.
- **Single-speed configuration.** The example is configured for 100 Mbps operation. Adapting to 10 Mbps or 1 Gbps requires reconfiguring the TSE MAC speed register and the matching PLL/SGMII settings in Platform Designer.

GeMRTOS Example Applications

11.1 Introduction

The previous chapter presented the code patterns that recur throughout GeMRTOS applications: infinite loops, priority configuration, event waiting, and the like. This chapter goes one step further and presents eleven complete, runnable applications, each built entirely from those patterns and from the public GeMRTOS API described throughout this manual. Where the previous chapter answered "how do I write an infinite-loop task?", this chapter answers "what does a real GeMRTOS application, exercising synchronization, multiprocessor scheduling, or networking end to end, actually look like?"

Figure 11.1 groups all eleven by theme, with a one-line summary of what each demonstrates – an expanded version of the same decision figure in Chapter 2, Getting Started for Evaluators, for a reader arriving at this chapter directly.

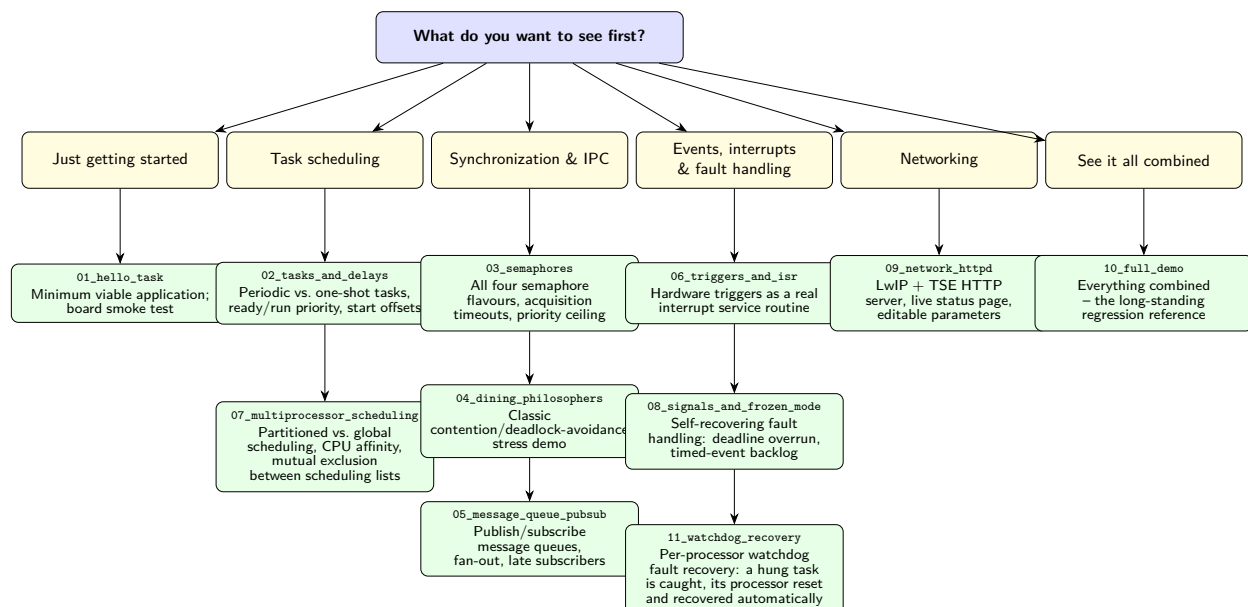


Figure 11.1: Which example do I need? (expanded, with what each one demonstrates)

Every application described here ships with the GeMRTOS distribution, under `ip/GeMRTOS_Multiprocessor/`

examples/, one self-contained directory per example. Each carries its own README.md, documenting the exact expected output on the JTAG UART and a short list of things to try once it is running — this chapter explains the mechanism and the reasoning behind each example; the per-example README.md is the place to look for the literal terminal transcript. Chapter 2, Getting Started for Evaluators, covers how to list, select, build, and run any of them; this chapter assumes that part is already familiar and concentrates on what each example teaches.

Deliberately, no supporting code is shared between examples. If two examples need the same kind of helper function, each carries its own copy rather than reaching into a shared library. This is not an oversight: being able to open exactly one directory and read everything that a given example does, without chasing definitions across the tree, is worth more here than the small amount of duplication it costs. Three examples — 04_dining_philosophers, 06_triggers_and_isr, and 09_network_httpd — were originally part of one large combined demo (10_full_demo, described in Section 11.11) and were later extracted into their own directories without changing a single line of the logic that moved; only the file boundaries changed.

11.2 01_hello_task — The Minimum Viable Application

Every GeMRTOS distribution needs one example that answers a single question as quickly as possible: does the toolchain work, is the board wired correctly, and does a task actually run? 01_hello_task is exactly that, and nothing more — one task, created in main(), printing a greeting and an incrementing counter over the JTAG UART once a second, forever.

```
void hello_task(void *pdata)
{
    GS_TCB *ptcb = gu_TaskGetCurrentTCB();
    unsigned int count = 0;

    while (1) {
        gu_printf("[%s] Hello from GeMRTOS! count=%u\n",
                  ptcb->TCB_description, count);
        count++;
        gu_TaskDelay(0, 0, 0, HELLO_TASK_DELAY_MS);
    }
}
```

Code 11.1: The complete hello_task task body.

Everything a first-time reader needs to recognize a GeMRTOS task is already here: the task function takes one void * argument and returns nothing; gu_TaskGetCurrentTCB() retrieves the calling task's own control block, which carries its human-readable description string; and gu_TaskDelay() suspends the task for a wall-clock duration expressed as hours, minutes, seconds, and milliseconds, here just under one second. Creating and starting this task takes two calls in main():

```
ptcb = gu_TaskCreate((void *) hello_task, (void *) 0, "hello_task");
if (ptcb != (GS_TCB *) 0) {
    gu_TaskTypeSet(ptcb, G_TCBType_OneShot);
    gu_TaskStartWithOffset(ptcb, 0, 0, 0, 0);
} else {
    gu_printf("ERROR: could not create hello_task (TCB pool exhausted)\n");
    while (1);
}
```

Code 11.2: Creating and starting `hello_task`.

A freshly created task is neither ready nor running until `gu_TaskStartWithOffset()` releases it — here with a zero offset, so it becomes ready immediately. `hello_task` is marked `G_TCBType_OneShot` rather than periodic because it manages its own repetition with its own `while(1)` loop and its own `gu_TaskDelay()` call; this is, in fact, how most “recurring” tasks in GeMRTOS applications are written, and Section 11.3 contrasts it directly against the alternative.

11.3 02_tasks_and_delays — Periodic and One-Shot Tasks, Side by Side

GeMRTOS offers two genuinely different ways for a task to run repeatedly, and choosing between them is one of the first design decisions any application makes. `02_tasks_and_delays` puts both side by side so the difference is visible, not just described: `task_periodic_beacon` does its work and returns, and the scheduler (scheduler) re-releases it automatically every period; `task_worker` never returns at all — it loops forever, delaying itself with `gu_TaskDelay()` exactly as `hello_task` did in the previous section.

```
void task_periodic_beacon(void *pdata)
{
    GS_TCB *ptcb = gu_TaskGetCurrentTCB();
    static unsigned int tick = 0;

    gu_printf("[%s] tick %u\n", ptcb->TCB_description, tick);
    tick++;
}
```

Code 11.3: A strictly periodic task: does its work, then returns.

Configuring a task as periodic is one additional call beyond what Section 11.2 already showed:

```
gu_TaskTypeSet(ptcb, G_TCBType_Periodic);
gu_TaskPeriodSet(ptcb, 0, 0, 0, BEACON_PERIOD_MS);
gu_TaskStartWithOffset(ptcb, 0, 0, BEACON_START_OFFSET_S, 0);
```

Code 11.4: Making a task periodic.

Once started, GeMRTOS's kernel re-releases `task_periodic_beacon` from scratch every `BEACON_PERIOD_MS` milliseconds, with no further action required from the task itself — the kernel, not the task's own code, owns the repetition. This distinction matters beyond style: only a genuinely periodic task can miss its own deadline in a way the kernel can detect and report, which is exactly the mechanism `08_signals_and_frozen_mode` (Section 11.9) builds on.

The third task in this example, `task_supervisor`, demonstrates a different kind of control entirely: rather than managing its own execution, it periodically suspends and resumes `task_worker` from the outside, using `gu_TaskSuspend()` and `gu_TaskResume()` — one task exercising direct control over another's readiness, distinct from anything a task does to itself.

11.4 03_semaphores — Every Synchronization Primitive GeMRTOS Offers

GeMRTOS provides four flavors of semaphore, and 03_semaphores dedicates one self-contained scenario to each: a binary semaphore used as a one-shot signal between two tasks (ping/pong); a counting semaphore managing a bounded resource pool, including the acquisition-timeout path a task takes when the pool is exhausted; a plain Mutex (Mutex) together with GeMRTOS's static priority-ceiling protocol, which prevents a low-priority task holding a Mutex from causing unbounded priority inversion (priority inversion) against a higher-priority one; and a recursive Mutex, taken and given twice in a row from the same task without deadlocking against itself.

```
g_sem_ping      = gu_SemaphoreCreateBinary(0);
g_sem_pool      = gu_SemaphoreCreateCounting(SYNC_POOL_SLOTS, SYNC_POOL_SLOTS);
g_mutex_shared  = gu_SemaphoreCreateMutex();
g_recursive_mutex = gu_SemaphoreCreateRecursiveMutex();
```

Code 11.5: Creating one of each semaphore flavor.

Every flavor is acquired and released through the same pair of calls, `gu_SemaphoreTake()` and `gu_SemaphoreGive()`, with the difference between flavors captured entirely at creation time — an application never needs to know, at the call site, which kind of semaphore it is holding:

```
if (gu_SemaphoreTake(g_mutex_shared, timeout_ticks) == G_TRUE) {
    /* critical section */
    gu_SemaphoreGive(g_mutex_shared);
} else {
    gu_printf("UNSUCCESSFULL SEMAPHORE TAKEN\n");
}
```

Code 11.6: Taking a semaphore with a bounded timeout.

A timeout of `G_LATEST_TIME` blocks indefinitely; any other value bounds the wait and returns `G_FALSE` if it expires — the pattern 04_dining_philosophers (Section 11.5) relies on to avoid deadlocking altogether rather than merely detecting it after the fact.

11.5 04_dining_philosophers — Contention and Deadlock Avoidance

The dining-philosophers problem is the classic illustration of resource contention among concurrent processes, and 04_dining_philosophers implements the equally classic solution: thirty tasks ("philosophers") arranged in a ring, each needing two shared binary semaphores ("forks") — its own and its neighbor's — to proceed. Acquired in the same order by every philosopher, this would deadlock the moment every philosopher holds one fork and waits forever for the other; acquired in an order that alternates by philosopher index instead (even philosophers take left-then-right, odd philosophers take right-then-left), the circular wait that causes deadlock can never form.

```
if ((task_index & 1) == 0) {
    first_sem = forks[left_fork(task_index)];
    second_sem = forks[right_fork(task_index)];
} else {
    second_sem = forks[right_fork(task_index)];
```

```

    first_sem = forks[left_fork(task_index)];
}

```

Code 11.7: Asymmetric fork acquisition order breaks the circular wait.

Both `gu_SemaphoreTake()` calls use a bounded timeout rather than blocking forever, so a philosopher unable to acquire both forks in time gives up cleanly and retries a second later instead of risking an unbounded wait — the same timeout mechanism Section 11.4 introduced, here doing real work under genuine multiprocessor contention rather than a single controlled scenario. Running this example on hardware, the terminal shows a steady stream of philosophers eating, interspersed with occasional timeout messages exactly where contention for a shared fork was real — evidence the mechanism works, not merely that it compiles.

11.6 05_message_queue_pubsub — Publish/Subscribe Messaging

A detail that surprises newcomers familiar with other real-time operating systems: GeMRTOS message queues are publish/subscribe, not point-to-point. A single `gu_MessageQueueSend()` (or its formatted convenience wrapper, `gu_MessageQueuePrintf()`) delivers one independent copy of the message to every task currently subscribed — not to whichever task happens to call `gu_MessageQueueReceive()` first. `05_message_queue_pubsub` is built specifically to make this visible, with three scenarios: one publisher broadcasting to two independent subscribers; a fast and a slow subscriber sharing one queue, so a bounded-timeout send can be observed reaching the fast one immediately while genuinely waiting on the slow one; and a subscriber that joins late, demonstrating that subscription is never retroactive — a task that subscribes after a message was sent never receives that particular message.

```

gu_MessageQueuePrintf(g_queue_sensor, "reading %u", reading);

```

Code 11.8: One publish, delivered independently to every current subscriber.

```

gu_MessageQueueSubscribe(ptcb, g_queue_sensor, (void *) recv_buf, PUBSUB_RECV_BUF_LEN);

while (G_TRUE) {
    (void) gu_MessageQueueReceive(g_queue_sensor);
    gu_printf("logged: %s\n", recv_buf);
}

```

Code 11.9: Subscribing, then receiving in an ordinary loop.

`gu_MessageQueueSubscribe()` associates a task with a queue and a receive buffer once, before the task's main loop begins; every subsequent `gu_MessageQueueReceive()` call blocks until the next message this particular subscriber has not yet seen lands in that buffer. Compare this against `03_semaphores'` counting-semaphore pool (Section 11.4): both manage concurrent access to a shared resource, but a semaphore pool hands out one of a fixed number of interchangeable permits, while a message queue delivers the same message to every interested party — different tools for genuinely different problems, not two ways of writing the same thing.

11.7 06_triggers_and_isr — Hardware Triggers as a Real Interrupt Service Routine

GeMRTOS's trigger mechanism is the abstraction through which a task waits for a hardware event, and 06_triggers_and_isr demonstrates the mechanism with a real one: the JTAG UART's own hardware interrupt, not a simulated or software-only stand-in. One task is registered directly against the interrupt itself and acts as the literal interrupt service routine, dispatching to two further software triggers that two ordinary tasks wait on independently — one for incoming characters, one for outgoing FIFO space.

```
GS_RCB *jtag_uart_trigger = gu_TriggerCreate(DEVICE_IRQ(JTAG_UART_SERVER), 0);
gu_TriggerRegisterTask(ptcb, jtag_uart_trigger);
gu_TriggerEnable(jtag_uart_trigger);
```

Code 11.10: A trigger created against a real hardware interrupt.

A non-zero second argument to gu_TriggerCreate() gives the trigger a timeout, turning gu_TriggerWait() into a bounded wait rather than an indefinite one — the read-side trigger in this example uses exactly that, so that with nobody typing on the terminal, the timeout path fires deterministically and repeatably, once per window, without requiring a human to sit at the keyboard for the example to demonstrate anything:

```
if (gu_TriggerWait() == G_TRGStatus_triggered) {
    /* real data arrived -- happy path */
} else {
    gu_printf("Read timeout in JTAG UART\n");
}
```

Code 11.11: Happy path and timeout path from one gu_TriggerWait() call.

gu_TriggerEnableHook() and gu_TriggerDisableHook() let a trigger's creator plug directly into the peripheral's own interrupt-enable bits, so the generic trigger mechanism itself never needs to know anything about UART registers — the peripheral-specific detail lives entirely in the hook function, keeping the trigger abstraction genuinely peripheral-independent.

11.8 07_multiprocessor_scheduling — Partitioned, Global, and Exclusive Scheduling

If one capability sets GeMRTOS apart from a conventional single-processor RTOS adapted to run on several cores, it is the hybrid scheduling list (hybrid scheduling list) mechanism, and 07_multiprocessor_scheduling is built specifically to make it observable rather than merely documented. Three configurations run side by side, each printing which processor actually executed it on every iteration: an ordinary task in the default scheduling list, free to run on whichever processor is available (global scheduling); a task assigned to a scheduling list associated with only one processor, which as a direct consequence can never run anywhere else (partitioned scheduling, giving that task fixed processor affinity); and two tasks sharing a scheduling list associated with every processor, but capped so that at most one of them may run anywhere in the system at a time — mutual exclusion between processors achieved through scheduling-list configuration, with no Mutex involved at all.

```
g_lcb_affinity = gu_SchedulingListCreate(GS_LCBTypeFP);
gu_SchedulingListAssociateProcessor(g_lcb_affinity, TOPOLOGY_AFFINITY_PROCESSOR,
                                   TOPOLOGY_AFFINITY_PRIORITY);
```

Code 11.12: Pinning a scheduling list, and therefore its tasks, to one processor.

```
g_lcb_exclusive = gu_SchedulingListCreate(GS_LCBTypeFP);
for (cpu_id = 1U; cpu_id <= (G_UINT32) G_NUMBER_OF_PCB; cpu_id++) {
    gu_SchedulingListAssociateProcessor(g_lcb_exclusive, cpu_id,
                                       TOPOLOGY_EXCLUSIVE_PRIORITY);
}
gu_SchedulingListExclusionSet(g_lcb_exclusive, TOPOLOGY_EXCLUSION_LIMIT);
```

Code 11.13: A list associated with every processor, capped to one runner system-wide.

A task joins a scheduling list with one further call, `gu_SchedulingListAssociateTask(ptcb, plcb)`, made before the task is started; from that point on, which processor it is even eligible to run on — and, for the exclusive list, how many of its siblings may run alongside it anywhere in the system — is governed entirely by the scheduling-list configuration, not by anything the task's own code does. Running this example on a real board and watching the reported processor number change (or, for the pinned task, deliberately never change) is the most direct demonstration this manual offers of what multiprocessor system (multiprocessor system) scheduling in GeMRTOS actually buys an application.

11.9 08_signals_and_frozen_mode — Fault Handling and Robustness

Production systems eventually meet conditions their designer did not plan for, and GeMRTOS's answer is not to pretend they cannot happen but to make them observable and recoverable. `08_signals_and_frozen_mode` induces two such conditions on purpose, on every run, and shows the system recovering from both without any special handling beyond what the kernel already provides.

The first is a genuinely missed deadline: a periodic task (Section 11.3 explained why it must be periodic, not one-shot, to be reachable this way) deliberately overruns its own period on a scheduled cycle. The kernel resets it and, before its next release resumes, delivers a signal reporting exactly what happened:

```
gu_SignalCreate(G_SCBType_TCB_ABORTED, 0, (void *) ptcb,
               (void *) fault_signal_task_aborted, (void *) 0);
```

Code 11.14: Registering a signal for a missed deadline.

The second is frozen mode (frozen mode): a hardware watchdog that measures how far behind the system has fallen in servicing its own timed events, not merely whether a task is still running. This example deliberately creates a burst of many tasks releasing on the identical instant against a tightened threshold, so the resulting backlog trips frozen mode reliably, then watches the system report non-frozen mode (non-frozen mode) again, unattended, the moment the backlog drains:

```
gu_FrozenModeThresholdSet(gu_ConvertTime(0, 0, 0, FAULT_FROZEN_THRESHOLD_MS));
gu_SignalCreate(G_SCBType_FROZEN_MODE, 0, (void *) 0,
               (void *) fault_signal_frozen, (void *) 0);
gu_SignalCreate(G_SCBType_UNFROZEN_MODE, 0, (void *) 0,
               (void *) fault_signal_unfrozen, (void *) 0);
```

Code 11.15: Registering signals for entering and leaving frozen mode.

Neither fault touches a critical section (critical section) or disables interrupts anywhere in this example's own code; both are consequences of ordinary, documented task scheduling under deliberately concentrated load, and both recover on their own, which is precisely the point this example exists to make.

11.10 09_network_httpd — A Networked Status and Configuration Page

09_network_httpd is the networking showcase: LwIP running over the TSE Ethernet core, serving a live status page over HTTP, with two independent mechanisms working together on that one page. A status table refreshes automatically in the visitor's browser, driven by LwIP's Server-Side Includes and a small extensible registry — adding a new value to the page is one function pair added to that registry, nothing else. A configuration form lets the same page change two of the running application's parameters — a periodic task's period, and a self-looping task's delay — live, without rebuilding or reflashing anything.

```
static G_UINT32 gemrtos_web_uptime_row_count(void)
{
    return 1U;
}

static u16_t gemrtos_web_uptime_render_row(G_UINT32 row_index, char *buf, u16_t buf_len)
{
    G_UINT32 uptime_s = (G_UINT32) (gu_SystemTimeGet() / (G_UINT64) G_TICKS_PER_SECOND);
    return (u16_t) snprintf(buf, buf_len,
        "<tr><td>Uptime (s)</td><td>%lu</td></tr>", (unsigned long) uptime_s);
}
```

Code 11.16: One status-table row: how many rows, and how to render one.

Every registered source contributes one or more rows the same way, and the dispatch logic that walks the registry and assembles the page never needs to change when a new source is added — a direct application of the same registry pattern the underlying architecture documentation (see the Network/Web Interface architecture) describes in more detail than a usage-focused manual chapter needs to. The receive and transmit sides of the networking stack itself — the mSGDMA-driven tasks that move Ethernet frames in and out of LwIP — follow the pattern already described earlier in this manual's networking discussion; this example is where that pattern is exercised by a complete, runnable application rather than described in isolation.

11.11 10_full_demo — The Complete Reference Application

10_full_demo is where every mechanism this chapter described so far runs together in one application: the dining philosophers of Section 11.5, the periodic and one-shot tasks of Section 11.3, the JTAG UART trigger server of Section 11.7, and the networked status page of Section 11.10, all at once, on real hardware. It predates the rest of this catalogue — the other nine examples were carved out of it, one coherent slice at a time, each verified to still behave exactly as it did inside the combined demo before being called a separate example in its own right — and it remains the project's long-standing regression reference: whenever anything in GeMRTOS changes, confirming that 10_full_demo still behaves the same way is the single strongest piece of evidence that nothing broke. Selecting no example at all when building — omitting the selection option described in Chapter 2 entirely — builds exactly this application, unchanged from every GeMRTOS release before this example catalogue existed.

11.12 11_watchdog_recovery — Per-Processor Watchdog Fault Recovery

Every example so far recovers from a fault that stays inside the kernel's own bookkeeping – a missed deadline, a timed-event backlog. 11_watchdog_recovery demonstrates the deeper case: a task that stops cooperating with the kernel entirely, disables its own interrupts, and hangs forever. Ordinary scheduling has no way to reach a task in that state; the GeMRTOS controller's per-processor hardware watchdog does, by resetting the processor itself and handing control back to the kernel to recover.

faulty_cpu1, pinned to processor 1, runs a few healthy iterations, then arms its own processor's watchdog and hangs on purpose – with interrupts masked, so nothing refreshes the watchdog before it expires:

```
gm_WatchdogTimeoutSet(WATCHDOG_DEMO_TIMEOUT_REG << G_WATCHDOG_TIMEOUT_SHIFT);
gm_PrcWatchdogEnable();
GRTOS_DISABLE_PRC_GENERAL_IRQ;
while (1) {
    /* never halts, never yields, never touches the kernel */
}
```

Code 11.17: Arming the watchdog immediately before the deliberate hang.

When the watchdog expires, the controller resets processor 1 alone – every other processor, and every task running on it, is untouched. The reset-vector redirection registered earlier (via `avalon_monitor`, not shown here) sends that one processor's reboot through the same boot path already used for processor 2 and above, which does not clear `.bss`; the kernel control block therefore survives the reset intact. `gk_START_KERNEL()` recognizes this as a recovery rather than a cold boot, kills the abandoned task, restores the kernel's invariants, and invokes a handler the application registers up front:

```
gu_WatchdogHandlerSet(watchdog_demo_handler);
```

Code 11.18: Registering the recovery handler.

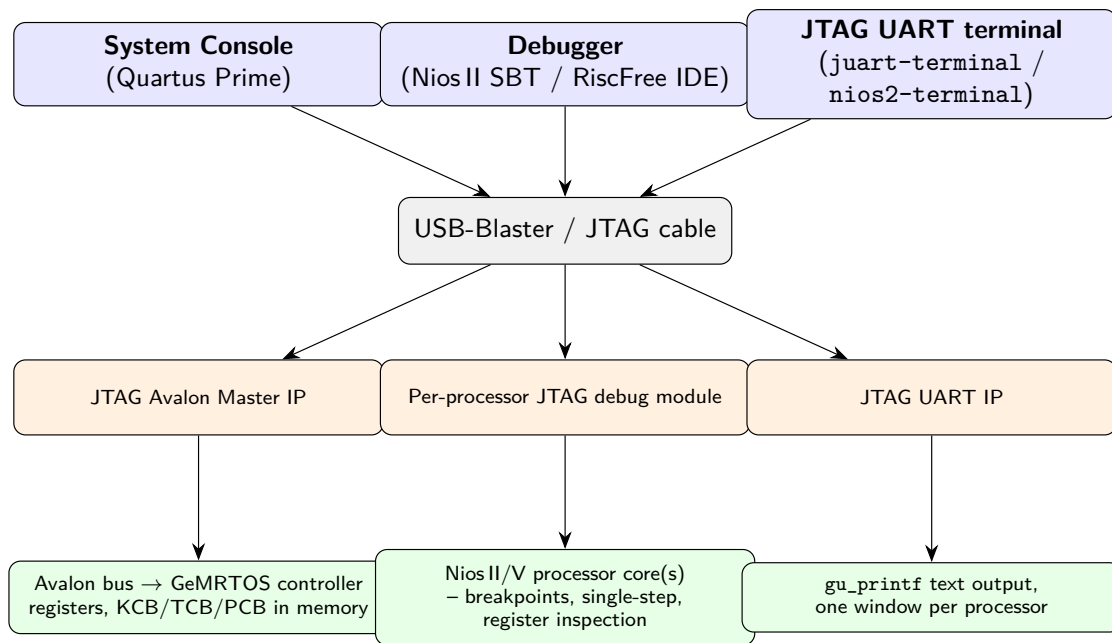
Meanwhile `steady_worker`, pinned to processor 2, prints once a second for the entire run. Its output never stops, never repeats, and never shows a gap – direct, continuous evidence that recovering one processor leaves the rest of the multiprocessor system undisturbed. This example requires two processors (`G_NUMBER_OF_PCB >= 2`) and is the reason `07_multiprocessor_scheduling` is not the only entry in this catalogue that needs more than one.

11.13 Conclusion

Eleven examples, one API. Every mechanism GeMRTOS offers — task management, every synchronization primitive, publish/subscribe messaging, hardware-triggered interrupt service, multiprocessor system scheduling, fault recovery, and networking — appears here doing real, observable work on real hardware, not as an isolated code fragment but as part of a complete, runnable application a reader can build, flash, and watch. Reading this chapter alongside the examples themselves, rather than instead of them, is the intended way to use it: the prose here explains why each example is built the way it is; the source, and the JTAG UART output it produces, is where that explanation is proven true.

Debugging GeMRTOS

A single JTAG connection carries three logically independent debugging paths at once, described in detail throughout this chapter: Figure 12.1 shows all three together for a reader new to embedded JTAG debugging, before the sections below cover each in turn.



All three paths share one physical JTAG connection but are logically independent: System Console can read controller registers while the debugger holds a processor at a breakpoint, and the JTAG UART keeps carrying `gu_printf` output throughout.

Figure 12.1: JTAG/toolchain data flow: three independent paths, one physical connection.

12.1 General Recommendations

GeMRTOS applications can be debugged using the same tools as uniprocessor NiosII/V designs (e.g., the Eclipse-based NiosII Software Build Tools debugger or the Ashling RiscFree IDE for Nios V). Two compile-time

macros in `gemrtos_config.h` control debug-friendliness:

- **G_DEBUG_WAITING_LOOP (set to 1 for debugging):** When set to 1, idle processors spin in a software while loop instead of asserting `waitrequest` to halt the bus. Hardware halt is not visible to the debugger; the while-loop alternative allows breakpoints and register inspection while the processor is idle.
- **G_DEBUG_WHILEFOREVER_ENABLE (set to 1 for debugging):** When enabled, internal consistency checks (`PRINT_ASSERT`) are active. Any violated invariant prints a diagnostic message to `stdout`, follows it with the kernel function-call stack on `stderr`, and then loops forever at the failing line, making it easy to locate the source of corruption. Both go through the non-blocking console (Section 12.3), so a failing assertion reports without stalling the system — but note that the two halves of the report arrive on *different* JTAG UART instances, so capture both.

Important: Set `G_DEBUG_WAITING_LOOP = 1` and `G_DEBUG_WHILEFOREVER_ENABLE = 1` during development. Revert both to 0 before production builds to restore full performance and hardware-halt idle behavior.

12.2 Debugging Multiprocessor Applications

Debugging a multiprocessor system introduces challenges absent in single-processor designs. The following practices mitigate the most common issues:

- **Single-processor bring-up:** Configure the GeMRTOS IP for one processor (`NPROCESSORS = 1`) and verify application logic before scaling up. Many task-ordering and resource bugs are easier to identify without concurrent execution on multiple processors.
- **gu_printf for tracing:** Use `gu_printf` rather than `printf` for diagnostic output inside tasks. `gu_printf` serializes output across processors with the global GeMRTOS mutex, preventing interleaved messages that are otherwise unreadable in a multiprocessor context. Unlike `printf`, it also never blocks — see Section 12.3.
- **Critical section tracking:** Excessive time inside a critical section starves all other processors. Instrument entry/exit with `gm_StatusCountingTimeGet` and `gm_StatusResetCountingTimeGet` to measure critical-section duration.
- **Frozen mode tracing:** If the system enters frozen mode unexpectedly, read `gu_FrozenModeTimeGet` to measure total frozen time and `gu_FrozenModeThresholdGet` to verify the threshold setting. An unexpectedly large frozen time indicates that the next timed event was missed.
- **Mutex grant register:** Reading `GRTOS_MTX_PRC_GRANTED` returns the CUID of the processor currently holding the GeMRTOS mutex. A value that never changes during a stall indicates a deadlock in the critical section.

12.3 Console Output Guarantees

The GeMRTOS console functions — `gu_printf` and `gu_fprintf` — make one guarantee and deliberately withhold another. Understanding both is necessary before relying on console output as evidence.

- **They never block.** In any build configuration, on any path, a console call returns in bounded time.
- **They do not guarantee delivery.** Output that the JTAG UART cannot accept, because no host is draining its FIFO, is discarded rather than waited for.

12.3.1 Why Delivery Is Not Guaranteed

The trade is forced by the hardware, not chosen for convenience. The JTAG UART is a *system-wide* resource: one instance serves standard output and one serves standard error, shared by every processor. A console write must therefore hold the global GeMRTOS mutex to keep messages from interleaving.

That makes a blocking write unsafe. While a processor owns the mutex, every other processor that needs it is stalled — and the GeMRTOS controller suspends a processor's watchdog countdown while it holds the mutex, so such a stall cannot be recovered by the watchdog either. A single blocking `printf` on a board with no JTAG host attached, which is the normal deployed configuration, would stop the entire system permanently.

Discarding output is the lesser cost, and it is the only alternative to stalling that does not require buffering.

12.3.2 What Is Discarded, and When

The console writes only what the FIFO is observed to have room for, and re-polls for more until the message is complete. It gives up on the remainder of a message only after `GK_CONSOLE_STALL_MS` of continuous time during which the FIFO freed no space at all. Any byte written restarts that deadline, so a host that is draining — however slowly — always receives complete messages.

`GK_CONSOLE_STALL_MS` is set in `gemrtos_config.h`:

- **Development builds: 2000 ms.** Losing a diagnostic message costs more than a slow boot.
- **Production builds: 1 ms.** Determinism wins; the console must not perturb timing.

It is bounded even in development. An unbounded console wait is precisely the defect this design removes.

Two properties hold regardless of the setting. A message is never interleaved with another processor's output, because the mutex is held throughout. And a message is either delivered whole or truncated at its tail — it is never delivered with a hole in the middle.

12.3.3 Steady-State Output Versus Bursts

The distinction that matters in practice is between steady output and bursts.

Measured JTAG UART host drain latency is approximately **1.07 s per 512-byte FIFO**. Ordinary periodic tracing — a line every few hundred milliseconds — is far below that rate and is delivered intact. A large burst is not: a 4 KB start-up banner needs roughly eight drain cycles, about 8.7 s, and no bounded wait can emit it. Raising `GK_CONSOLE_STALL_MS` does not fix this; it only trades lost output for a multi-second system-wide stall.

If your application must emit bursts without loss, buffer them in the application — for example behind a message queue drained by a dedicated low-priority task, as `06_triggers_and_isr` demonstrates — rather than issuing them directly to the console.

12.3.4 Detecting Lost Output

Because loss is a normal outcome, GeMRTOS counts it. The kernel maintains a monotonic counter of discarded console bytes, readable at runtime over the JTAG-to-Avalon master through Quartus System Console (see Section 12.4). A counter that is not advancing confirms that nothing has been lost; one that is advancing tells you that missing lines are a console limitation rather than a logic error — a distinction that is otherwise impossible to make from the output alone.

Practical consequence. Never treat absent console output as proof that code did not execute. Confirm against the discard counter, or against state read directly from memory, before concluding that a task did not run.

12.4 System Console

GeMRTOS incorporates a JTAG Avalon Master IP core that connects to the Avalon bus of all processors in the system. This enables Quartus System Console to access all memory-mapped devices — including the GeMRTOS controller registers — at runtime without halting the processors.

Useful System Console operations for GeMRTOS applications:

- **Inspect control block state:** Read the KCB, TCB, and PCB structures directly from memory to observe task states, scheduling list memberships, and processor assignments.
- **Read GeMRTOS controller registers:** Access the global region (GEMRTOS_DRIVER_BASE) and processor shadow region (GEMRTOS_DRIVER_PROCESSOR_BASE) registers to inspect mutex ownership, IRQ enable masks, time counters, and frozen mode status.
- **Pause and resume individual processors:** System Console supports pausing a single Nios II processor while others continue, allowing inspection of one processor's register state without stopping the entire system.

To use System Console, open it from Quartus Prime via *Tools* → *System Debugging Tools* → *System Console*, connect to the JTAG cable, and open the .sof file used to program the FPGA.

12.5 HPS Compatibility

The Altera Hard Processor System (HPS) can be connected to the GeMRTOS bus by enabling the **Enable HPS Internal Access** parameter in the GeMRTOS IP configuration window (see Section 6.3). When enabled, an Avalon bridge is added that allows the HPS to read and write all memory-mapped devices on the GeMRTOS processor bus.

HPS access provides a non-intrusive runtime inspection path:

- The HPS Linux userspace can `mmap` the GeMRTOS controller base address and read status registers without halting the soft-processor application.
- Combined with a small HPS-side monitoring daemon, this enables continuous logging of GeMRTOS task states, IRQ activity, and time counter values for post-mortem analysis.

12.6 Common Failure Modes

Table 12.1: Common GeMRTOS failure modes and remediation.

Symptom	Likely cause	Remediation
System hangs at startup	Processor with CPUID != 1 not released from reset	Verify PRC_CDE_ADDRx reset code and GeMRTOS Controller connection in Platform Designer

Continued on next page

Table 12.1: Common GeMRTOS failure modes and remediation. (Continued)

Symptom	Likely cause	Remediation
All tasks starved; only idle runs	Critical section never released	Search for <code>gm_GeMRTOSCriticalSectionEnter</code> without matching <code>Exit</code> ; verify no early return inside critical section
Unexpected frozen mode	Frozen threshold too small or timed event missed	Increase <code>gu_FrozenModeThreshold</code> Set value or reduce task period load
<code>gu_printf</code> output garbled	<code>printf</code> called without serialization	Replace all bare <code>printf</code> calls with <code>gu_printf</code>
Periodic task misses deadline	Stack overflow or excessive critical-section hold	Increase task stack size (<code>G_TASK_STACKSIZE</code>); profile critical-section duration
<code>PRINT_ASSERT</code> fires on startup	<code>G_DEBUG_WHILEFOREVER_ENABLE = 0</code> masks the message	Set <code>G_DEBUG_WHILEFOREVER_ENABLE = 1</code> and rebuild
Task spins forever, never yields	Missing <code>gu_TaskDelay</code> /blocking call in a loop, or a genuine logic bug	Arm the task's per-processor watchdog (<code>gm_WatchdogTimeoutSet/gm_PrcWatchdogEnable</code> , §4.5) so the processor recovers automatically instead of hanging permanently; register <code>gu_WatchdogHandlerSet()</code> to observe or halt on recovery

12.7 Production Build Configuration

`gemrtos_config.h` is the single file an integrator is expected to review and tune: static object-pool sizes, task stack sizes, and the compile-time debug flags below. All of the flags in this section live in that file's `COMPILATION_OPTIONS` block. Before shipping a GeMRTOS application to production, verify the following:

Table 12.2: Compile-time debug macros in `gemrtos_config.h` and their production values.

Macro	Debug	Production	Effect in production
<code>G_DEBUG_SYSTEM_CHECK_ENABLE</code>	1	0	The one-time structural integrity check at kernel start-up is skipped. Cost when enabled is negligible (runs once); not part of the flag group the publication pipeline rewrites (see below).

Continued on next page

Table 12.2: Compile-time debug macros in `gemrtos_config.h` and their production values. (Continued)

Macro	Debug	Production	Effect in production
<code>G_DEBUG_WHILEFOREVER_ENABLE</code>	1	0	<code>PRINT_ASSERT</code> becomes a no-op; <code>G_DEBUG_WHILEFOREVER</code> becomes an infinite loop (hardware halt equivalent). <code>gemrtos_monitor.c</code> 's runtime checks, call-stack recording, and pool-exhaustion <code>malloc</code> fallback are all disabled — pool exhaustion halts instead.
<code>G_DEBUG_GET_POINTED_FIELD_CHECK</code>	1	0	<code>get_pointed_field()</code> stops validating the <code>BLOCK_HASH</code> sentinel before every kernel field read. High cost when enabled (a check on every read); direct access with no overhead when disabled.
<code>G_DEBUG_SET_POINTED_FIELD_CHECK</code>	1	0	<code>set_pointed_field()</code> stops validating the <code>BLOCK_HASH</code> sentinel before every kernel field write. High cost when enabled (a check on every write); direct access with no overhead when disabled.
<code>G_DEBUG_WAITING_LOOP</code>	1	0	Idle processors assert <code>waitrequest</code> to halt the bus instead of spinning in a software while loop.
<code>G_NUMBER_OF_TCB</code>	—	<i>size correctly</i>	Must cover all simultaneously active tasks plus one IDLE task per processor. Pool exhaustion in production triggers <code>G_DEBUG_WHILEFOREVER</code> .
<code>G_NUMBER_OF_ECB</code>	—	<i>size correctly</i>	ECB pool must cover all simultaneous semaphore-waits, trigger registrations, and message-queue send/receive operations.

Important: With `G_DEBUG_WHILEFOREVER_ENABLE = 0`, pool exhaustion halts the system via `G_DEBUG_WHILEFOREVER` rather than falling back to `malloc`. Size all object pools (`G_NUMBER_OF_TCB`, `G_NUMBER_OF_ECB`, `G_NUMBER_OF_RCB`, `G_NUMBER_OF_SCB`) generously before deployment and validate during bring-up with `G_DEBUG_WHILEFOREVER_ENABLE = 1`.

12.7.1 Two different “production” configurations exist today

Readers building from the canonical GeMRTOS source tree and readers building from the GitHub-published copy are not, at present, building the same configuration — this is a known, disclosed gap rather than a documented feature, and integrators should be aware of it before assuming either copy reflects the values in Table 12.2:

- **The canonical source tree** — the tree every host test, Windows simulation run, and on-hardware

verification in this project's development history has actually exercised — ships with `G_DEBUG_SYSTEM_CHECK_ENABLE`, `G_DEBUG_WHILEFOREVER_ENABLE`, `G_DEBUG_GET_POINTED_FIELD_CHECK`, and `G_DEBUG_SET_POINTED_FIELD_CHECK` all set to 1 (checks **on**), and `G_DEBUG_WAITING_LOOP` set to 0. This is the configuration that has been verified at every tier.

- **The GitHub-published copy** is produced by an automated publication pipeline that, as one of its build steps, rewrites four of those flags directly in the staged copy it publishes: `G_DEBUG_WHILEFOREVER_ENABLE`, `G_DEBUG_GET_POINTED_FIELD_CHECK`, `G_DEBUG_SET_POINTED_FIELD_CHECK`, and `G_DEBUG_WAITING_LOOP` are all forced to 0 (`G_DEBUG_SYSTEM_CHECK_ENABLE` is left at its canonical value of 1, unmodified by this step). This never touches the canonical tree itself — only the staged copy that gets zipped and pushed.
- **The resulting “checks off” configuration has never itself been run through the host test suite, the Windows simulation, or on real hardware** — every verification tier this project uses runs against canonical, which never carries this combination of values. An integrator who downloads the published copy and builds it as-is is, in that specific respect, running a configuration this project has not independently confirmed correct.

This is not a defect in the pipeline's logic — flipping expensive debug checks off for a published artifact is a reasonable default — but the gap between “verified” and “published” should be closed by verifying the off-configuration directly, rather than assumed safe by extrapolation from the on-configuration. Until then, an integrator building for real production deployment should, at minimum, re-run their own application's verification against whichever configuration they actually intend to ship, rather than trust either copy's defaults unexamined.

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

```

1 #define gm_GeMRTOSCriticalSectionEnter \
2     if (!(gm_TimeCountersResetGet())) { \
3         GRTOS_CMD_CRITICAL_SECTION_GET; // Acquire hardware mutex \
4         if ((gm_KernelModeGet() == G_FALSE)) { \
5             gm_kcb.G_PCBTbl[gm_ProcessorId()-1].PCB_EXECTCB->TCB_MTX_NESTED++; \
6         } \
7     }

```

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

```

1 #define gm_GeMRTOSCriticalSectionExit \
2     if (!(gm_TimeCountersResetGet())) { \
3         if ((gm_KernelModeGet() == G_FALSE)) { \
4             g_kcb.G_PCBTbl[gm_ProcessorId()-1].PCB_EXECTCB->TCB_MTX_NESTED--; \
5             gm_MutexNestedValue(g_kcb.G_PCBTbl[gm_ProcessorId()-1].PCB_EXECTCB->TCB_MTX_NESTED); \
6         } \
7         GRTOS_MTX_RLS; \
8     }

```

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

```

1 static inline void gm_MutexNestedValue(G_UINT32 data)
2 {
3     const G_UINT32 value =
4         ((G_UINT32)1u << CTRL_MTX_RLS_DSB_BIT) |
5         ((data != 0u) ? (G_UINT32)1u : (G_UINT32)0u);
6
7     gm_IOWR(GEMRTOS_DRIVER_BASE, ADDR_CTRL_STATUS, value);
8 }

```

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 ← 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 ← 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 enqueueing 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_PrcInterruptDisable
gm_PrcInterruptEnable
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_EREAddressSet
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
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

```
static inline void gm_FrozenModeDisable(void);
```

Description: The `gm_FrozenModeDisable` function disables the frozen mode event. By default, the frozen mode starts in a disabled state.

Parameters

The `gm_FrozenModeDisable` function does not accept any parameters.

Returns

The `gm_FrozenModeDisable` function 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

```
static inline void gm_FrozenModeEnable(void);
```

Description: The `gm_FrozenModeEnable` function enables the frozen mode event. By default, the frozen mode starts in a disabled state. Before invoking this function, ensure that the frozen threshold is properly configured using the `gu_FrozenModeThresholdSet` function. The `gm_FrozenModeDisable` function disables the frozen mode.

Parameters

The `gm_FrozenModeEnable` function does not accept any parameters.

Returns

The `gm_FrozenModeEnable` function 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

```
static inline void gm_GatedClockDisable(void);
```

Description: The gm_GatedClockDisable function disables the output clock of the the GeMRTOS controller. It may be used in special applications.

Parameters

The gm_GatedClockDisable function does not accept any parameters.

Returns

The gm_GatedClockDisable 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

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

```
static inline void gm_GatedClockEnable(void);
```

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

Parameters

The `gm_GatedClockEnable` function does not accept any parameters.

Returns

The `gm_GatedClockEnable` 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

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

```
static inline G_UINT32 gm_StatusFrozenModeEnableGet(void);
```

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` function returns `G_TRUE` if frozen mode is enabled, `G_FALSE` if disabled.

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

```
static inline void gm_SystemReset(void);
```

Description: The gm_SystemReset function resets the GeMRTOS controller. Is is executed only by the GeMRTOS kernel.

Parameters

The gm_SystemReset function does not accept any parameters.

Returns

The gm_SystemReset function 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

```
static inline void gm_TimeCountersReset(void);
```

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

Parameters

The `gm_TimeCountersReset` function does not accept any parameters.

Returns

The `gm_TimeCountersReset` function 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

```
static inline void gm_TimeCountersHold(void);
```

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

Parameters

The gm_TimeCountersHold function does not accept any parameters.

Returns

The gm_TimeCountersHold 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

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

```
static inline void gm_TimeCountersUnhold(void);
```

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

Parameters

The `gm_TimeCountersUnhold` function does not accept any parameters.

Returns

The `gm_TimeCountersUnhold` function 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_TimeCountersUnreset`
`gu_FrozenModeThresholdGet`
`gu_FrozenModeThresholdSet`
`gu_FrozenModeTimeGet`

13.4.12 `gm_TimeCoutersUnreset`

Prototype

```
static inline void gm_TimeCoutersUnreset(void);
```

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

Parameters

The `gm_TimeCoutersUnreset` function does not accept any parameters.

Returns

The `gm_TimeCoutersUnreset` function 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

```
static inline void gm_IrqDisable(G_UINT32 irq);
```

Description: The `gm_IrqDisable` function disables the specified device interrupt request event (IRQ) in the GeMRTOS controller. The `gm_IrqEnable(irq)` function enables the `irq` external interrupt in the GeMRTOS controller. This function is essential for managing interrupt handling and preventing specified DIRQs from triggering.

Parameters

The `gm_IrqDisable` function accepts the following parameter:

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

Returns

The `gm_IrqDisable` function 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

```
static inline void gm_IrqEnable(G_UINT32 irq);
```

Description: The `gm_IrqEnable` function enables the specified device interrupt request event (DIRQ) in the GeMRTOS controller. This function is crucial for allowing specified DIRQ to trigger. The `gm_IrqDisable(irq)` function disables the `irq` external interrupt in the GeMRTOS controller.

Parameters

The `gm_IrqEnable` function accepts the following parameter:

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

Returns

The `gm_IrqEnable` function 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` function 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

```
static inline void gm_TriggerPrcIRQ(G_UINT32 proc);
```

Description: The `gm_TriggerPrcIRQ` function interrupts the processor `proc`. If `proc` interrupt is enabled in the

GRTOS controller, then the processor is interrupted.

Parameters

The `gm_TriggerPrvIRQ` function accepts the following parameter:

- `proc:CPU_ID` of processor to interrupt.

Returns

The `gm_TriggerPrvIRQ` function 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_RESOURCE` structure. 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_
```

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 the TaskControl 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 the GS_ECB structure 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_PrcInterruptDisable

Prototype

```
static inline void gm_PrcInterruptDisable(void);
```

Description: The gm_PrcInterruptDisable function disables the processor interrupt from the GeMRTOS controller.

Parameters

The gm_PrcInterruptDisable function does not require any parameters.

Returns

The gm_PrcInterruptDisable function does not return any value; it writes the processor interrupt-disable bit directly and returns immediately.

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_PrcInterruptEnable
gm_ProcessorHalt
gm_ProcessorId
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq

13.4.26 gm_PrcInterruptEnable

Prototype

```
static inline void gm_PrcInterruptEnable(void);
```

Description: The gm_PrcInterruptEnable function enables the processor interrupt from the GeMRTOS controller.

Parameters

The gm_PrcInterruptEnable function does not require any parameters.

Returns

The gm_PrcInterruptEnable function does not return any value; it writes the processor interrupt-enable bit directly and returns immediately.

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_PrcInterruptDisable`
`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 compliance 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_PrcInterruptDisable`

gm_PrcInterruptEnable
gm_ProcessorId
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq

13.4.28 gm_ProcessorId

Prototype

```
static inline G_UINT32 gm_ProcessorId(void);
```

Description: The gm_ProcessorId function retrieves the ID of the current processor. This function is useful for identifying the processor.

Parameters

The gm_ProcessorId function does not accept any parameters.

Returns

The gm_ProcessorId function 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_PrcInterruptDisable
gm_PrcInterruptEnable
gm_ProcessorHalt
gm_ProcessorInterrupt
gu_ProcessorWaitForIrq

13.4.29 gm_ProcessorInterrupt

Prototype

```
static inline void gm_ProcessorInterrupt(G_UINT32 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 function 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 function 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_PrcInterruptDisable
gm_PrcInterruptEnable
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_PrcInterruptDisable
gm_PrcInterruptEnable
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 the `GS_LCB` structure 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 the GS_LCB structure representing the scheduling list to which the task should be added.

Returns

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

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

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

Returns

The function returns a `G_TRUE`.

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 ≥ 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() or gu_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 semaphore's 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(): see
```

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 and returns immediately. 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 *s
```

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_EREAddressSet

Prototype

```
static inline void gm_EREAddressSet(G_UINT32 value);
```

Description: The gm_EREAddressSet function 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_EREAddressSet has no parameter.

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

Returns

The gm_EREAddressSet function 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

```
static inline void gm_MutexNestedValue(G_UINT32 data);
```

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

```
static inline G_UINT32 gm_ReadInputs(void);
```

Description: The `gm_ReadInputs` function transfer the data from the `gemrtos_phy` input conduit of GeMRTOS controller.

Parameters

The `gm_ReadInputs` function 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

```
static inline G_UINT32 gm_StatusFrozenModeActiveGet(void);
```

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 function returns G_TRUE if the frozen mode event is active, G_FALSE if inactive.

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

```
static inline void gm_WriteOutputs(G_UINT32 data);
```

Description: The gm_WriteOutputs function transfer the data input to the gemrtos_phy output conduit of GeMRTOS controller.

Parameters

The gm_WriteOutputs function requires the following parameter:

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

Returns

The `gm_WriteOutputs` 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

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

None. The `gu_fprintf` function returns `void`.

Thread-safety

Thread-safe. Serialized by the global GeMRTOS mutex, which the function acquires and releases around the write, so a message is never interleaved with output from another processor.

Blocking behaviour

Never blocks, in any build configuration. The JTAG UART is shared by every processor and a console write necessarily holds the global mutex, so a blocking write would stall the whole system — see Section 12.3.

Error conditions

None returned. Output that the JTAG UART FIFO cannot accept while the host is not draining it is discarded rather than waited for; discarded bytes are counted. See Section 12.3.

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

None. The `gu_printf` function returns `void`.

Thread-safety

Thread-safe. Serialized by the global GeMRTOS mutex, which the function acquires and releases around the write, so a message is never interleaved with output from another processor.

Blocking behaviour

Never blocks, in any build configuration. The JTAG UART is shared by every processor and a console write necessarily holds the global mutex, so a blocking write would stall the whole system — see Section 12.3.

Error conditions

None returned. Output that the JTAG UART FIFO cannot accept while the host is not draining it is discarded rather than waited for; discarded bytes are counted. See Section 12.3.

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() + confi
```

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

Parameters

The function uses the following parameters:

- **TaskCode:** A pointer to the function that implements the task's code (the task's entry point). It is the name of the function that implements the task code.
- **p_arg:** A pointer to an argument that will be passed to the `TaskCode` function each time the task is invoked. This is a `void *` and can be cast to other types within the task code.
- **format:** A format string, similar to `printf`, used to create a description string for the task (up to `G_TCB_DESCRIPTION_LENGTH` characters). This string can contain format specifiers that are replaced by subsequent arguments.

Returns

The `gu_TaskCreate` function returns a pointer to the `GS_TCB` structure of the newly created task. This pointer should be used in all subsequent calls related to that task. A `NULL` return indicates task creation failure.

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 the TaskControl 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 seconds, unsigned int ms);
```

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

Parameters

The function uses the following parameters:

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

Returns

The `gu_TaskStartWithOffset` function returns `G_TRUE` upon successful task startup.

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

```
static inline G_UINT32 gm_StatusCountingTimeGet(void);
```

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` function returns `G_TRUE` if the `TIME_HOLD` bit is set, `G_FALSE` otherwise.

Thread-safety

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

Error conditions

None.

Preconditions

GeMRTOS mutex must be held for a meaningful reading.

Postconditions

None; this is a read-only status query.

See also

gm_StatusResetCountingTimeGet
gm_SystemTimePrescaleGet
gm_SystemTimePrescaleSet
gu_SystemTotalTimeGet

13.4.63 gm_StatusResetCountingTimeGet

Prototype

```
static inline G_UINT32 gm_StatusResetCountingTimeGet(void);
```

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 function has no parameter.

Returns

The gm_StatusResetCountingTimeGet function returns G_TRUE if the time counters are held in reset, G_FALSE otherwise.

Thread-safety

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

Error conditions

None.

Preconditions

GeMRTOS mutex must be held.

Postconditions

None; this is a read-only status query.

See also

gm_StatusCountingTimeGet
gm_SystemTimePrescaleGet
gm_SystemTimePrescaleSet
gu_SystemTotalTimeGet

13.4.64 gm_SystemTimePrescaleGet

Prototype

```
static inline G_UINT32 gm_SystemTimePrescaleGet(void);
```

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

Parameters

The gm_SystemTimePrescaleGet function has no parameter.

Returns

The `gm_SystemTimePrescaleGet` function 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`
`gu_SystemTotalTimeGet`

13.4.65 `gm_SystemTimePrescaleSet`

Prototype

```
static inline void gm_SystemTimePrescaleSet(G_UINT32 scale);
```

Description: The `gm_SystemTimePrescaleSet` function sets the system clock prescale to get the system time unit. By default, the prescale is set to configure a 10MHz frequency for system time units. The `gm_SystemTimePrescaleGet` function returns the time prescale.

Parameters

The `gm_SystemTimePrescaleSet` has one parameter:

- `scale`: The prescale configuration.

Returns

The `gm_SystemTimePrescaleSet` function 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`
`gu_SystemTotalTimeGet`

13.4.66 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 function 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 function 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

13.4.67 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`

13.4.68 `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`

13.4.69 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`

13.4.70 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`

13.4.71 `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 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 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`

13.4.72 `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`

13.4.73 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

13.4.74 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

13.4.75 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

```

13.4.76 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.

13.4.77 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.

13.4.78 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.

13.4.79 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.

13.4.80 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.

13.4.81 gm_WatchdogTimeoutSet

Prototype

```
static inline void gm_WatchdogTimeoutSet(G_UINT32 timeout);
```

Description: Configures the per-processor watchdog timeout value, written to CTRL_WATCHDOG_BIT in the Control Register (§4.2.4). The name is spelled Watchdog, not Watchdog – the real primitive name, not a typo introduced by this reference. Setting the timeout alone does not arm the watchdog; gm_PrcWatchdogEnable must also be called. See §4.5 for the full design.

Parameters

timeout — raw timeout value; only bits [G_WATCHDOG_TIMEOUT_SHIFT+11 : G_WATCHDOG_TIMEOUT_SHIFT] (12 bits, shifted right by G_WATCHDOG_TIMEOUT_SHIFT = 16 before being written) reach the hardware register. At the shared 10MHz tick base, this gives a representable range of approximately 6.55ms (register value 1) to 26.8s (register value 0xFFFF).

Returns

Does not return a value.

Thread-safety

Affects only the calling processor's own watchdog configuration; safe to call without the GeMRTOS mutex.

Error conditions

None – values outside the representable range are silently truncated to the low 12 bits of the shifted result.

Preconditions

None.

Postconditions

The processor's configured watchdog timeout is updated. The countdown itself does not start unless `gm_PrcWatchdogEnable` has also been called (independently, in either order).

See also

`gm_PrcWatchdogEnable`
`gm_PrcWatchdogDisable`
`gu_WatchdogHandlerSet`

13.4.82 `gm_PrcWatchdogEnable`

Prototype

```
static inline void gm_PrcWatchdogEnable(void);
```

Description: Arms the calling processor's own watchdog countdown, using whatever timeout was last configured via `gm_WatchdogTimeoutSet`. The countdown reloads to the full timeout continuously while the processor is halted or disabled, and only counts down while both enabled and actively running – there is no explicit software “pet” call. See §4.5.

Parameters

The `gm_PrcWatchdogEnable` function does not accept any parameters.

Returns

Does not return a value.

Thread-safety

Affects only the calling processor's own watchdog state; safe to call without the GeMRTOS mutex. Not meaningful to call on behalf of another processor – always applies to `gm_ProcessorId()` of the caller.

Error conditions

None.

Preconditions

An application-appropriate timeout should already be configured via `gm_WatchdogTimeoutSet` – enabling with the power-up default timeout value arms at that default instead. **Never call this from a task running on processor 1, on real embedded hardware** – confirmed 2026-08-26, a processor-1 watchdog reset hangs the entire board (see §4.5). Safe on processors 2 through N, and on the Windows simulation for any processor including 1.

Postconditions

The calling processor's watchdog countdown is active. If it is not halted again, or explicitly disabled, before the configured timeout elapses, the processor is reset by the controller.

See also

`gm_WatchdogTimeoutSet`
`gm_PrcWatchdogDisable`
`gu_WatchdogHandlerSet`

13.4.83 gm_PrcWatchdogDisable

Prototype

```
static inline void gm_PrcWatchdogDisable(void);
```

Description: Stops the calling processor's watchdog countdown without clearing its configured timeout value – a later gm_PrcWatchdogEnable reuses the same timeout unless gm_WatchdogTimeoutSet is called again first.

Parameters

The gm_PrcWatchdogDisable function does not accept any parameters.

Returns

Does not return a value.

Thread-safety

Affects only the calling processor's own watchdog state; safe to call without the GeMRTOS mutex.

Error conditions

None.

Preconditions

None – safe to call even if the watchdog is already disabled.

Postconditions

The calling processor's watchdog countdown is stopped; the configured timeout value is unchanged.

See also

gm_WatchdogTimeoutSet
gm_PrcWatchdogEnable
gu_WatchdogHandlerSet

13.4.84 gu_WatchdogHandlerSet

Prototype

```
void gu_WatchdogHandlerSet(  
    enum g_watchdog_action (*handler)(const struct g_watchdog_report *report));
```

Description: Registers (or, passed NULL, uninstalls) the system-wide watchdog-recovery handler, invoked after a watchdog-triggered processor reset – but only *after* the kernel has already restored its own invariants (the abandoned task unlinked and freed, or IDLE's state left untouched if it was the abandoned task). The handler receives a read-only snapshot, never a live pointer into kernel structures, and chooses whether the system continues normally or halts. See §9.3 for the full usage pattern with a worked example.

Parameters

handler — function to install, matching `enum g_watchdog_action (*)(const struct g_watchdog_report *report)`; or NULL to uninstall (restores the default: always G_WATCHDOG_ACTION_RESUME, the same behavior as before this feature existed).

Returns

Does not return a value.

Thread-safety

Not synchronized against a concurrent call to gu_WatchdogHandlerSet from another processor. Intended to be set once during application initialization, before any processor's watchdog is armed.

Error conditions

None.

Preconditions

None, though setting this before arming any processor's watchdog (rather than after) is the only usage pattern that guarantees no recovery is missed.

Postconditions

The registered handler (or no handler, if NULL) is invoked on every subsequent watchdog recovery, on any processor – one global handler, not per-processor; `report->CPUID` distinguishes which processor recovered if the application needs to react differently per processor.

See also

`gm_WatchdogTimeoutSet`

`gm_PrcWatchdogEnable`

`gm_PrcWatchdogDisable`

Windows Emulation Layer

14.1 Overview

The GeMRTOS system is designed to run on Altera FPGA platforms with Nios II or Nios V soft processors connected to a hardware GeMRTOS controller (implemented in VHDL). For development, testing, and validation purposes before FPGA deployment, GeMRTOS can also be emulated on Windows: OS threads act as processors and Windows fibers act as tasks, executing the *same* kernel/HAL sources that run on the FPGA.

Figure 14.1 shows where this equivalence comes from: every layer above the Platform Abstraction Layer (PAL) — the application, the user API, the kernel services, and the kernel core — is portable C, compiled unchanged for either target. Only the PAL's own implementation, and what sits below it, differs between the two.

This chapter documents:

- Known limitations and differences from hardware
- How to install and configure the tools needed for the Windows build (Visual Studio Code and a MinGW GCC toolchain)
- How to build, run, and debug the Windows simulation

14.2 Known Limitations

The Windows emulation cannot replicate all FPGA-specific behaviors. Users should be aware of these limitations:

14.2.1 ERET Instruction Behavior

Hardware: RISC-V `mret` instruction triggers atomic return from interrupt with mutex release.

Windows: No direct equivalent. Return-from-interrupt is simulated via function return. Release logic is triggered by software release request, not `mret` instruction.

Impact: Interrupt return timing may differ from hardware. Test on FPGA before production deployment.

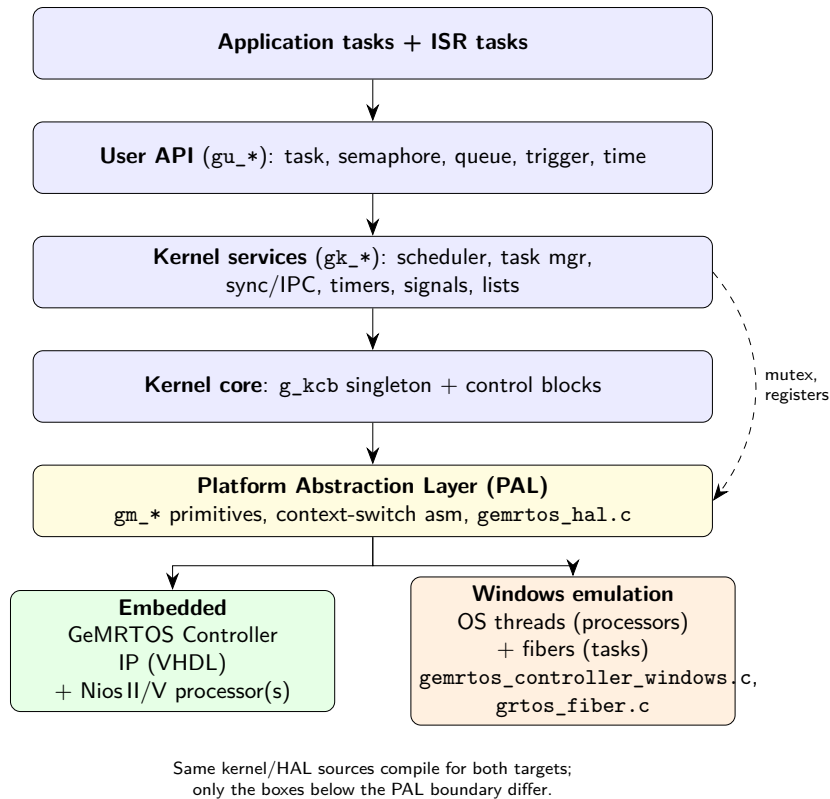


Figure 14.1: Dual-platform layering: the same kernel above the PAL boundary, two different implementations below it.

14.2.2 Interrupt Timing

Hardware: Hardware can interrupt immediately when IRQ asserted.

Windows: Interrupt delivery depends on Windows thread scheduling. No guarantees on interrupt latency.

Impact: Timing-sensitive code may behave differently. Use Windows emulation for functional testing, not timing validation.

14.2.3 Clock Accuracy

Hardware: Precise hardware clock with predictable cycle timing.

Windows: Timer interrupts subject to OS scheduling and system load.

Impact: Time-based tests may be less reliable. Use qualitative tests (“did task complete?”) not quantitative (“did task complete in exactly N cycles?”). A concrete instance: the watchdog monitor thread polls via `Sleep(1)`, which stock Windows does not guarantee is exactly 1ms – a configured timeout of roughly one second was observed taking several real seconds to actually fire. Adequate for confirming the watchdog fires and recovers at all during development; not a substitute for confirming the configured timeout duration itself on real hardware.

14.2.4 Processor Reset

Hardware: Can reset individual processors via controller – used, among other purposes, by the per-processor watchdog (§4.5).

Windows: Emulated specifically for the watchdog use case (`GRTOS_Watchdog_Monitor/GRTOS_Watchdog_Fire`, `gemrtos_controller_windows.c`): the target processor's OS thread is terminated and a fresh one created, re-entering `gk_START_KERNEL()` exactly as a genuine boot would – the same, unmodified kernel recovery code runs on both platforms. No general-purpose “reset any processor for any reason” primitive exists outside this watchdog path.

Impact: The watchdog's own reset behavior can be developed and functionally verified on Windows before ever touching hardware (as it was for this feature). Timing is wall-clock-approximate, not cycle-accurate – see §14.2.3. A narrow, acknowledged residual risk remains: if the terminated thread happened to be inside one of the emulation's own short internal critical sections (not the GeMRTOS-level mutex, which is hardware-guaranteed never held at a legitimate firing) at the exact moment of termination, that lock stays held forever – bounded only by how short those regions already are by design, not eliminated. Real hardware has no equivalent failure mode. See `gemrtos.md` §14 for the full embedded-vs-Windows comparison.

14.3 Installing the Required Tools

Building and running the Windows simulation requires two things: an editor/build environment (Visual Studio Code) and a 32-bit MinGW GCC toolchain. Both are free and only need to be installed once per development machine.

14.3.1 Installing Visual Studio Code

1. Download the Windows installer from <https://code.visualstudio.com/>.
2. Run the installer and accept the default options (it is convenient to enable “Add to PATH” and “Register Code as an editor for supported file types” if offered).
3. Launch VS Code, open the Extensions view (**Ctrl+Shift+X**), search for **C/C++** (published by Microsoft, extension id `ms-vscode.cpptools`), and install it. This extension provides IntelliSense and reads the project's `c_cpp_properties.json` (see below) and recognizes the build task defined in `tasks.json`.

14.3.2 Installing the MinGW GCC Toolchain

The Windows simulation is built with a 32-bit MinGW GCC toolchain. The recommended way to obtain one is via MSYS2:

1. Download and run the installer from <https://www.msys2.org/> and follow the wizard (default install location `C:\msys64`).
2. From the “MSYS2” start-menu shortcut it installs, run:

```
1 pacman -S mingw-w64-i686-gcc
```

3. Add the toolchain's bin directory (C:\msys64\mingw32\bin by default) to the Windows PATH environment variable (Windows Settings → "Edit environment variables for your account" → edit Path, add the entry, then open a *new* terminal for it to take effect).
4. Verify the installation by opening a new terminal (PowerShell or Command Prompt) and running:

```
1 gcc --version
```

This should print the installed GCC version. If gcc is not found, double-check the PATH entry above.

14.4 Windows Build and Testing

14.4.1 Package Composition

The Windows simulation package lives in `ip/GeMRTOS_Multiprocessor/gemrtos_windows/` and contains *only* the Windows-exclusive files: the port (`gemrtos_controller_windows.c/.h`, `grtos_fiber.c/.h`) and the VS Code project definition (`.vscode/tasks.json`, `.vscode/c_cpp_properties.json`). The shared kernel/HAL sources and the application are *not* duplicated here — they are provided by the BSP and application that the normal embedded design flow generates under `software/`. The complete Windows project is therefore assembled from these exclusive files plus the generated `software/hellogemrtos_bsp/` (kernel/HAL) and `software/hellogemrtos/` (application).

14.4.2 First-Run Setup

On its first invocation, `gemrtos_build.bat` copies `ip/GeMRTOS_Multiprocessor/gemrtos_windows/` to `software/gemrtos_windows/` (only if it does not already exist), placing the Windows-exclusive files beside the generated BSP and application:

```
1 gemrtos_build.bat -std -qpf <your_qpf> -qsys <your_qsys>
```

14.4.3 Opening and Configuring the VS Code Workspace

1. In VS Code, choose **File** → **Open Folder** and select `software/gemrtos_windows/` as the workspace folder.
2. The workspace already includes a `.vscode/` folder (copied in the first-run step above) with two files pre-configured for this project:
 - `tasks.json` — defines the default build task (invokes `gcc.exe` from your PATH with all the source files and include paths the Windows simulation needs).
 - `c_cpp_properties.json` — tells the C/C++ extension where to find headers for IntelliSense, and which compiler to use for code analysis. Its `compilerPath` defaults to `C:\msys64\mingw32\bin\gcc.exe`; edit that field if your MSYS2 installation lives elsewhere.

No manual project setup is required beyond installing the tools above.

14.4.4 Build Commands

With the workspace open, run the default build task (**Terminal** → **Run Build Task**, or the shortcut **Ctrl+Shift+B**). Equivalently, from a terminal opened in that folder:

```
1 cd software/gemrtos_windows
2 gcc -g ../hellogemrtos/hellogemrtos.c \
3     ../hellogemrtos_bsp/drivers/src/gemrtos_kernel.c \
4     ../hellogemrtos_bsp/drivers/src/gemrtos_core.c \
5     ../hellogemrtos_bsp/drivers/src/gemrtos_list.c \
6     ../hellogemrtos_bsp/drivers/src/gemrtos_sem.c \
7     ../hellogemrtos_bsp/drivers/src/gemrtos_mq.c \
8     ../hellogemrtos_bsp/drivers/src/gemrtos_trigger.c \
9     ../hellogemrtos_bsp/drivers/src/gemrtos_task.c \
10    ../hellogemrtos_bsp/drivers/src/gemrtos_monitor.c \
11    ../hellogemrtos_bsp/drivers/src/gemrtos_hal.c \
12    gemrtos_controller_windows.c grtos_fiber.c \
13    -I ../hellogemrtos_bsp/ -I ../hellogemrtos_bsp/HAL/inc \
14    -I ../hellogemrtos_bsp/drivers/inc -I ../hellogemrtos -I . \
15    -I ../hellogemrtos_bsp/HAL/apps/http \
16    -lgcov -o hellogemrtos.exe
```

This produces `software/gemrtos_windows/hellogemrtos.exe`.

14.4.5 Run Commands

To execute the Windows application:

```
1 cd software/gemrtos_windows
2 ./hellogemrtos.exe
```

14.4.6 Debugging

To debug the emulation:

1. Add `printf` statements in the application or kernel source as needed
2. Rebuild with **Ctrl+Shift+B** (or the `gcc` command above)
3. Run: `./hellogemrtos.exe 2>&1 | tee output.log`
4. Analyze the output log for state transitions

