FreeRTOS Dual-Core MIPS like Architecture Implementation

Pool : PESHWAS

September 2, 2025

Contents

1 Introduction

This documentation provides a detailed description of the implementation of FreeRTOS for a dual-core MIPS like architecture. We explain the crucial components such as the scheduler, context switcher, and Board Support Package (BSP) with complete code examples. In addition, we include our attempt to integrate FreeRTOS with the x86 architecture (Windows) to simulate and test the operating system's functionality and correctness before deploying it on the actual MIPS like hardware. We also present an example task scenario used for benchmarking the OS and analyze its performance.

2 System Overview

2.1 Dual-Core MIPS like Architecture

Our system consists of a dual-core MIPS like processor, which is responsible for executing tasks. Each core operates independently, but they share memory and other system resources. The FreeRTOS-based kernel manages the execution of tasks by distributing them across both cores. Special care is taken to manage synchronization and context switching, ensuring that the two cores operate seamlessly in parallel without conflicts. The scheduler must ensure tasks are correctly distributed across the cores and that high-priority tasks on either core get CPU time as needed.

2.2 FreeRTOS on MIPS like

FreeRTOS was initially designed for single-core systems. For a dual-core system, we made modifications to the FreeRTOS kernel to ensure proper task management on both cores. Key changes were made to:

- **Scheduler Behavior**: Modified to account for multiple cores and allow simultaneous task execution on two processors.
- **Context Switching**: Extended to save and restore context on a per-core basis, so that tasks on different cores can be preempted and resumed independently.

- **Synchronization Mechanisms**: Enhanced (using mutexes and semaphores) to allow safe communication between tasks possibly running on different cores, preventing race conditions.

3 Implementation of Key Components

3.1 Scheduler

The scheduler is responsible for managing the execution of tasks on both cores. Each task is assigned a priority, and the scheduler ensures that the highest-priority ready task is executed first on an available core. For our dual-core MIPS like architecture, the scheduler was modified to handle tasks across two cores while ensuring fair and efficient distribution.

When initializing the scheduler, we set up the data structures needed for two cores (such as ready queues for each core or a combined priority list with core affinity information). An idle task is created for each core to ensure the core is never idle without a task. The task creation function is also adapted for the dual-core context; tasks can be created and will be managed by the scheduler which may assign them to a core (either statically or dynamically based on load).

Below is the code for initializing the scheduler and creating tasks in our dual-core system:

```
// Initialize the FreeRTOS scheduler for dual-core MIPS like system
void scheduler_init(void) {
    // Initialize task lists or structures for two cores
    initCoreStructures(); // e.g., separate ready lists for each core
    // Create idle task for each core to run when no other task is available
    createIdleTask(0);
    createIdleTask(1);
    // (Additional initialization as needed)
}

// Task creation API for both cores
// In this simplified model, tasks are created without binding to a specific core.
// The scheduler will assign tasks to cores dynamically based on availability and priority.
```

```
void create_task(void (*task_function)(void *), const char *
    name, uint32_t priority) {
     // Use FreeRTOS API to create a task. The new task is
    added to the ready list.
     BaseType_t result = xTaskCreate(task_function, name,
    configMINIMAL_STACK_SIZE, NULL, priority, NULL);
     if (result != pdPASS) {
         // Handle task creation failure (e.g., out of memory)
     }
}

// Start FreeRTOS scheduler for dual-core system
void start_scheduler(void) {
     // Start the scheduler which will begin tick interrupts
    on both cores
     // and allow tasks to start executing.
     vTaskStartScheduler();
     // vTaskStartScheduler will not return unless there is
    insufficient heap.
}
```

In our implementation, the FreeRTOS kernel was extended so that when the scheduler runs, it can choose a task for each core. For example, if two high-priority tasks are ready, the scheduler will run one on core 0 and the other on core 1 simultaneously. If a single highest-priority task is ready, one core will run that task while the other core might run a lower-priority task or its idle task. This dual-core scheduling ensures the CPU resources are utilized effectively.

3.2 Context Switcher

The context switcher is a core part of FreeRTOS. It is responsible for saving the state (CPU registers, stack pointer, etc.) of a running task before switching to another task, and restoring the state of the task being resumed. In a dual-core system, context switching must occur independently on each core. Each core can perform a context switch when its scheduler decides to run a different task (for example, during a tick interrupt or when a high-priority task becomes ready).
Below is the code (simplified) for the context switching mechanism in our dual-core MIPS like FreeRTOS port. We use macros `portSAVE_CONTEXT()` and `portRESTORE_CONTEXT()` (which are implemented in assembly for MIPS like) to save and load task contexts. The variable `pxCurrentTCB` holds a pointer to the

current task's control block (which includes the task's stack pointer), and `pxNextTCB` is a pointer to the next task chosen to run by the scheduler.

```
void vTaskSwitchContext(void) {
    // Select the highest priority task that is ready to run
    TCB_t *pxNextTCB = getHighestPriorityTask();

    // If the next task is different from the current task, perform a switch
    if (pxCurrentTCB != pxNextTCB) {
        // Save context of current task (push registers onto its stack, save stack pointer in TCB)
        portSAVE_CONTEXT();

        // Update the current task pointer to the new task
        pxCurrentTCB = pxNextTCB;

        // Restore the context of the new task (pop registers from its stack, set up stack pointer, etc.)
        portRESTORE_CONTEXT();
        // After this macro, the CPU registers have been loaded for the new task, and it will resume execution.
    }
}
```

Each core will perform this context switch routine independently. For instance, when core 0's tick interrupt fires, it may switch tasks on core 0 if needed, while core 1 can continue running its task (or also perform a switch if a tick on core 1 causes a higher priority task to be ready on that core).

3.3 Board Support Package (BSP)

The BSP is responsible for low-level hardware initialization, such as setting up interrupt vectors, timers, clocks, and memory, as well as booting secondary cores in a multi-core system. For the dual-core MIPS like architecture, we extended the BSP to initialize both cores and ensure the system is set up properly before the scheduler starts.

Key responsibilities of the BSP initialization include:

- Setting the CPU clock frequency and configuring the system timer (SysTick) for periodic tick interrupts on each core.
- Initializing the interrupt controller and defining the interrupt service

routines (ISRs) for timer interrupts and software interrupts (used for yields or inter-core interrupts).

- Releasing the second core from reset and starting its execution. Often, the primary core (core 0) brings up core 1 by setting its start address or using a special inter-processor interrupt.

- Initializing hardware peripherals (like UART for debugging, GPIOs, etc.) and setting up any required memory regions or cache configuration.

Below is a simplified code snippet for the BSP initialization on the dual-core MIPS like system:

```
void bsp_init(void) {
    // Initialize system clock and timer
    initSystemClock();
    initSysTickTimer(); // Configure SysTick timer for
   periodic interrupts

    // Set up the interrupt vector table and handlers for
   both cores
    initInterruptController();
    registerInterruptHandler(SYS_TICK_INT, SysTick_Handler);
    registerInterruptHandler(SOFTWARE_INT, Yield_Handler);

    // Boot up the second core (core 1) if not already
   running
    startSecondaryCore();  // e.g., release core 1 from reset
    and set its start address

    // Initialize memory management unit (if applicable) and
   caches
    initMemorySubsystem();

    // Initialize other board peripherals (UART, GPIO, etc.)
    initUART();
    initGPIO();
}
```

In the above code, `SysTick_Handler` would be the ISR that triggers on each timer tick to increment the OS tick count and possibly request a context switch. The `Yield_Handler` might be a software interrupt used to handle explicit yield calls (for example, from `portYIELD()` in the context switcher or

when a task yields). The function `startSecondaryCore()` is hardware-specific; on our platform, it writes the starting program counter for core 1 and then releases core 1 from reset, causing it to begin execution (where it will then call `bsp_init()` for core 1 or a simplified startup routine that joins the scheduler).

By the end of `bsp_init()`, both cores are up and running, the system timer and interrupts are configured, and the FreeRTOS scheduler can be started to begin multitasking on both cores.

3.4 Synchronization Mechanisms

In a dual-core system, tasks may need to communicate or share resources (like memory, peripherals, etc.), so synchronization is crucial. We utilize FreeRTOS mechanisms like semaphores and mutexes to prevent race conditions and ensure that tasks on different cores do not interfere with each other. FreeRTOS provides thread-safe synchronization primitives which we use without modification, as they are inherently safe to use across multiple cores as long as the kernel is properly managing critical sections and interrupts.

For example, if two tasks (possibly running on different cores) need to access a shared resource, we protect that access with a mutex. If one task holds the mutex, another task (even on the other core) that tries to take the mutex will block until the mutex is released. The FreeRTOS kernel handles the blocking and waking of tasks under the hood, even in our dual-core context.

Below is a simple example of a task function using a mutex for synchronization:

```
1  SemaphoreHandle_t xMutex; // global mutex handle
2
3  void task_function(void *pvParameters) {
4      // Task code that needs to access a shared resource
5      for(;;) {
6          xSemaphoreTake(xMutex, portMAX_DELAY); // Acquire the
       mutex (block indefinitely until available)
7          // --- Begin Critical Section: access shared resource
       here ---
8          sharedVariable++;
9          // (perform other operations on shared resource)
10         // --- End Critical Section ---
11         xSemaphoreGive(xMutex); // Release the mutex
12
```

```
        // Continue with non-critical work or yield
        vTaskDelay(10);
    }
}
```

In this example, `xMutex` is a binary mutex (created elsewhere using `xSemaphore` CreateMutex()) that protects access to `sharedVariable`. Even if two instances of `task_function` run on different cores, the mutex ensures that only one at a time enters the critical section. FreeRTOS handles the blocking of the second task and the resumption once the mutex is free. This prevents race conditions and ensures data integrity.

For other synchronization needs, we similarly use semaphores (for signaling events between tasks or ISRs and tasks) and queues (for message passing). These facilities are part of FreeRTOS and work in a multi-core setup with the kernel managing atomic access by briefly disabling interrupts or using atomic instructions where appropriate.

4 Definitions

The following definitions are crucial for the FreeRTOS implementation on the dual-core MIPS like system (these come from FreeRTOS configuration and port layer):

- `configUSE_16_BIT_TICKS`: Defines whether the system uses 16-bit or 32-bit tick counters. If set to 1, the tick count (and related timing calculations) use 16-bit integers, otherwise 32-bit (default for our 32-bit MIPS like system is 0, using 32-bit ticks).

- `portNUM_CONFIGURABLE_REGIONS`: Defines the number of memory regions that can be configured for Memory Protection Unit (MPU) support. This is relevant if using FreeRTOS MPU features. For our purposes, this may remain at the default (typically 0 or a small number) since standard FreeRTOS ports without MPU do not use configurable memory regions.

- `TickType_t`: The data type used for the OS tick count. It is an unsigned integer type. With `configUSE_16_BIT_TICKS` set to 0, `TickType_t` is typically defined as `uint32_t` (32-bit unsigned).

- `UBaseType_t`: An unsigned base type used by FreeRTOS for variables like loop counters, task priorities, and handle counts. On a 32-bit architecture like MIPS like, `UBaseType_t` is typically a `uint32_t`. (On 64-bit or 8-bit architectures, this type adjusts accordingly to the natural word size for efficiency.)

- `StackType_t`: The data type used for a task's stack entries. On a 32-bit system, this is usually defined as `uint32_t` (since the stack is an array of 32-bit words). Each task has an array of `StackType_t` allocated for its stack.

These definitions ensure that the kernel and application use consistent data types. For example, using a 32-bit tick count allows our system to run for a very long time before the tick count overflows (which is important for long uptimes), whereas a 16-bit tick count would overflow faster (after 65535 ticks). We chose to use 32-bit ticks given the capability of our 32-bit MIPS like cores and the requirement for long-running tasks.

5 Integration with x86 Architecture

5.1 Purpose of Integration

Before deploying our FreeRTOS-based OS on the actual dual-core MIPS like hardware, we wanted to thoroughly test its functionality and correctness. For this purpose, we integrated FreeRTOS with an x86 architecture environment (specifically, using the Windows FreeRTOS simulator). This allowed us to simulate the dual-core behavior and verify the scheduler, context switching, and synchronization in a controlled setting where debugging is easier. By using a PC-based simulation, we could also gather performance metrics and identify issues early, without needing the physical hardware powered on for every test.

5.2 Changes to FreeRTOS for x86

Running FreeRTOS on a Windows (x86) environment required some adaptations since a standard PC/Windows is not a real-time system and does not provide the same hardware features as a microcontroller. We made the following changes for the simulation:

- **Timer Simulation**: Replaced the hardware SysTick timer with the Windows `SetTimer` function to generate periodic callbacks. This simulates the periodic tick interrupt by invoking a callback function at a regular interval (in our case, 1 ms for a 1 kHz tick rate).

- **Interrupt Handling**: Because we cannot directly manipulate hardware interrupts in user-mode Windows, we simulated interrupts using threads or timer callbacks. The FreeRTOS tick increment and context switch trigger are called from the `SetTimer` callback (which acts as our tick ISR). Additionally, to simulate a software interrupt or yield, we used a Windows event or a higher-priority thread to trigger context switches.

- **Task Execution as Threads**: Each FreeRTOS task is run as a separate Windows thread at a low priority. The FreeRTOS scheduler doesn't truly context-switch the CPU as it would on a microcontroller; instead, it controls the suspension and resumption of these threads to simulate task switching. This approach is used in the FreeRTOS Windows port so that tasks can be managed in a way that respects the FreeRTOS scheduling decisions.

- **Simulated Peripherals**: For example, where our MIPS like hardware might use a UART or GPIO, in the simulation we use standard I/O calls. For instance, a task toggling an LED on hardware would simply print a message or toggle a GUI element on Windows. We implemented dummy functions for GPIO and UART that print messages to the console, to simulate the hardware behavior.

These modifications allow the core logic of our OS to run unmodified, with only the hardware-specific parts abstracted. The FreeRTOS kernel itself (scheduler, queues, semaphores, etc.) remains the same; we primarily changed the *port layer* (the part of FreeRTOS that is specific to the CPU architecture).

5.3 Simulation Code

In the Windows simulation, the key part of the port layer is setting up the timer to generate tick interrupts and handling context switches. We use the Win32 API function `SetTimer` to call a function `vPortIncrementTick` at a

regular interval (1 ms). This function increments the FreeRTOS tick and calls the scheduler. If a context switch is needed (i.e., a higher-priority task was woken by the tick), it triggers a yield to switch tasks.

Below is a snippet of the code used in the Win32 port for the tick simulation and scheduler start:

```
// This function is called by the Windows timer every 1ms to simulate the SysTick.
void vPortIncrementTick(HWND hwnd, UINT uMsg, UINT_PTR idEvent, DWORD dwTime) {
    // Increment the FreeRTOS tick count
    if (xTaskIncrementTick() != pdFALSE) {
        // If a context switch is required (i.e., a task with higher priority than current was unblocked)
        portYIELD(); // request a context switch yield
    }
}

// Override the FreeRTOS scheduler start function for Windows port
void vTaskStartScheduler(void) {
    // Create a Windows timer that calls vPortIncrementTick every 1 ms
    SetTimer(NULL, 0, 1, (TIMERPROC) vPortIncrementTick);
    // Start the first task manually (this sets up the initial task context on the current thread)
    vTaskStartFirstTask();
    // The scheduler is now running. In the Windows port, tasks are managed as threads.
    // vTaskStartScheduler will not return unless an error occurs.
}
```

In the above code:

- `xTaskIncrementTick()` is a FreeRTOS kernel function that increments the tick count and checks if any task's delay has expired or if any timeouts occurred. It returns `pdTRUE` if a context switch is needed at the end of the tick.
- `portYIELD()` in the Windows port will signal the FreeRTOS scheduler to switch tasks. In the MIPS like hardware, this would correspond to triggering a software interrupt or using the `Yield` assembly instruction.

- `vTaskStartFirstTask()` is a FreeRTOS internal function that starts the first task running. In the Windows port, this sets up the current thread context to the first FreeRTOS task. After this, the FreeRTOS scheduler logic (now driven by the Windows timer interrupts) will take over managing which task (thread) runs.

With this setup, our tasks can be created and run in the Windows environment as if they were running under the FreeRTOS scheduler on actual hardware. The timing won't be real-time exact (Windows is not a real-time OS, so the 1ms tick can jitter), but it's sufficient for functional testing and approximate performance measurement.

5.4 Testing and Results

To validate the system on the x86 simulation, we set up a benchmark scenario with two example tasks. The goal was to stress the scheduler and measure context switch times, CPU utilization, and general stability.

5.4.1 Benchmark Task Scenario

We created two tasks for the test:

- **TaskA**: A high-priority task that continually increments a counter and periodically yields. This task simulates a CPU-intensive high-priority workload.
- **TaskB**: A low-priority task that periodically prints the counter value. This task simulates a background task that should be preempted whenever TaskA is ready to run.

By running these two tasks, we force frequent context switches (since TaskA will run, then yield or block briefly, allowing TaskB to run, and then TaskA immediately preempts again due to its higher priority). This pattern is ideal for measuring the context switch overhead and scheduler efficiency.

Below is the code for these two tasks and the `main` function that sets up the test in the simulation environment:

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
```

```
5
6  // Shared counter and mutex for thread-safe access
7  static int sharedCounter = 0;
8  static SemaphoreHandle_t xMutex;
9
10 // High-priority task: increments a counter continuously
11 void vHighPriorityTask(void *pvParameters) {
12     for (;;) {
13         // Simulate work by incrementing a shared counter
14         xSemaphoreTake(xMutex, portMAX_DELAY);
15         sharedCounter++;
16         xSemaphoreGive(xMutex);
17         // Yield to let other tasks run (simulating periodic
       work)
18         vTaskDelay(1); // delay 1 tick (1ms in our tick rate)
19     }
20 }
21
22 // Low-priority task: periodically prints the counter value
23 void vLowPriorityTask(void *pvParameters) {
24     for (;;) {
25         xSemaphoreTake(xMutex, portMAX_DELAY);
26         int value = sharedCounter;
27         xSemaphoreGive(xMutex);
28         // Print the counter (simulating output, e.g., to
       UART)
29         printf("LowPriorityTask: Counter = %d\n", value);
30         // Delay for some ticks to simulate slower periodic
       activity
31         vTaskDelay(5); // delay 5 ticks (5ms)
32     }
33 }
34
35 int main(void) {
36     // Initialize the mutex before creating tasks
37     xMutex = xSemaphoreCreateMutex();
38
39     // Create the high and low priority tasks
40     // Higher number means higher priority in FreeRTOS (
       configMAX_PRIORITIES is configured accordingly)
41     xTaskCreate(vHighPriorityTask, "HighTask",
       configMINIMAL_STACK_SIZE, NULL, 2, NULL);
42     xTaskCreate(vLowPriorityTask, "LowTask",
       configMINIMAL_STACK_SIZE, NULL, 1, NULL);
43
```

```
    // Start the FreeRTOS scheduler (this will also start the
    Windows timer for ticks)
    vTaskStartScheduler();

    // We should never reach here as the scheduler will be
    running
    // If we do, it likely means there was insufficient heap
    memory to start the scheduler.
    return 0;
}
```

In the above code, `vHighPriorityTask` and `vLowPriorityTask` operate on a shared counter protected by a mutex (`xMutex`). TaskA (`HighTask`) runs with higher priority (priority 2) and increments the counter every iteration, yielding briefly with `vTaskDelay(1)` to allow TaskB to run. TaskB (`LowTask`) runs at lower priority (priority 1) and prints the counter value every 5 ticks. In a correct implementation, TaskB will only run when TaskA is in *delay* state, and as soon as TaskA becomes ready again (after 1 tick), it will preempt TaskB.

We instrumented the code to measure performance metrics. For example, to measure context switch latency, we recorded timestamps (using high-resolution timers provided by Windows) around the `portYIELD()` call and the actual task start to calculate how long the context switch took. We also measured CPU utilization of the simulation program to estimate scheduler overhead.

6 Conclusion

In this project, we extended FreeRTOS to support a dual-core MIPS like architecture by modifying the scheduler, context switching mechanism, and BSP initialization. We tried validating our implementation by integrating it with an x86 (Windows) simulation, which allowed us to run the OS in a controlled environment. The simulation was done to prove that tasks are scheduled correctly across both cores, context switching overhead is low, and synchronization between tasks works reliably.

We expect normal behavior on hardware, with potentially even better performance due to the real MIPS like cores and lack of underlying OS overhead. This implementation demonstrates that FreeRTOS can be adapted to multi-core processors. Future work could involve further optimizing the

scheduler for load balancing between cores and exploring more complex inter-core communication mechanisms for enhanced performance.