Operating Systems

Lab Project

Real Time Operating System

By: Namra Basharat, Faiqa Zarar Noor,

Alishbah Sajid, Ghania Munir

Roll No.: 476203, 471543, 482058, 460673

Date: 17 December 2025

Class: BS Bioinformatics UG-1

School of Interdisciplinary Engineering & Sciences

National University of Sciences and Technology

1 Introduction

A **Real-Time Operating System (RTOS)** is a lightweight operating system designed to respond to events within strict and predictable timing constraints. In contrast to general-purpose operating systems such as Linux or Windows, which optimize for throughput and average performance, an RTOS is optimized for *determinism*-the guarantee that operations will complete within a known maximum time. This makes RTOS systems essential for safety-critical and timing-sensitive applications such as aerospace controllers, automotive systems, medical devices, robotics, industrial automation, and embedded sensor networks.

At the core of an RTOS is a preemptive scheduler that manages multiple tasks, each representing an independent thread of execution. Tasks are assigned priorities, and the scheduler ensures that the highest-priority runnable task always executes first. Unlike traditional multitasking systems that rely on fairness or time-sharing, an RTOS focuses on meeting deadlines, minimizing latency, and guaranteeing bounded response times. This allows developers to design systems where timing behavior is as important as functional correctness.

Most RTOS kernels also provide essential real-time mechanisms, including:

- Task management for creation, deletion, and priority assignment.
- Deterministic context switching to ensure timely task swaps.
- Inter-task communication using queues, semaphores, and mutexes for safe data sharing.
- Software timers for scheduling periodic events.
- Interrupt handling with predictable latency.

Depending on timing requirements, real-time systems are categorized into:

- Hard real-time systems - missing deadlines causes system failure.
- Firm real-time systems - occasional missed deadlines degrade performance.
- Soft real-time systems - deadlines are preferred but not always enforced.

Thus an RTOS provides the structural foundation required to run multiple operations concurrently while ensuring the system responds within guaranteed and predictable time frames.

Among various RTOS options, **FreeRTOS** is one of the most widely adopted solutions due to its portability, simplicity, and deterministic behavior. Therefore, this project uses FreeRTOS to explore real-time scheduling, task creation, priority handling, and inter-task communication.

2 FreeRTOS

FreeRTOS is an open-source, lightweight, real-time operating system kernel specifically designed for microcontrollers and small embedded systems. It is one of the most widely adopted

RTOS solutions in both industry and academia due to its small memory footprint, modularity, portability, and highly deterministic behavior. The kernel supports more than 40 hardware architectures-including ARM Cortex-M, RISC-V, AVR, PIC, and ESP32-making it a universal RTOS for low-power embedded devices.

Key Characteristics of FreeRTOS

- **Preemptive Scheduling:** FreeRTOS uses priority-based preemptive scheduling. The highest-priority ready task runs immediately, ensuring deterministic timing.
- **Lightweight Kernel:** The minimal configuration can run in a few kilobytes of RAM and ROM, making it ideal for microcontrollers.
- **Configurable Tick Rate:** The system uses a periodic tick interrupt to manage delays and time-based scheduling.
- **Portability:** Only a small architecture-specific "portable layer" changes across CPUs; the core kernel remains identical.
- **Predictable Inter-task Communication:** FreeRTOS provides queues, semaphores, mutexes, event groups, and task notifications to synchronize tasks without unpredictable delays.

FreeRTOS Components

- **Tasks:** Independent threads of execution with assigned priorities.
- **Scheduler:** Ensures the highest-priority ready task always runs.
- **Queues:** FIFO communication mechanism for exchanging messages between tasks.
- **Semaphores and Mutexes:** Used for synchronization, resource protection, and interrupt-to-task signaling.
- **Software Timers:** Allow events to occur at fixed intervals without task overhead.
- **Idle Task:** Runs when no other tasks are ready; used for cleanup and optional low-power functions.

Why FreeRTOS?

This project specifically investigates how FreeRTOS schedules tasks and manages priority-based execution on the RISC-V architecture. FreeRTOS is ideal because:

- It has deterministic behavior, allowing accurate observation of timing and scheduling.
- It is easy to modify, enabling addition of user-defined tasks.
- It exposes students to industry-standard embedded development workflows.

Thus, FreeRTOS provides a clean, well-structured platform to explore real-time scheduling, task creation, context switching, and embedded concurrency.

3 FreeRTOS GitHub Repository

The official FreeRTOS GitHub repository is the primary source from which developers obtain the FreeRTOS kernel, hardware ports, memory managers, and example demos. The repository is structured in a modular and clean way so that anyone can understand how the kernel works, how it is ported to different architectures, and how to start building a FreeRTOS application.

Unlike a traditional monolithic codebase, the FreeRTOS repository is organized to highlight three important ideas:

1. The kernel is very small
2. All hardware-specific code is kept separate
3. Ready-to-build demo projects are included for every supported architecture

Everything else in the repository primarily exists to support these three goals.

3.1 The FreeRTOS Kernel

This is the most important directory for understanding the RTOS. It contains the logic that makes FreeRTOS an RTOS.

It contains task management, scheduling, software timers, and queue mechanisms. This code is fully portable, meaning it does not depend on any CPU architecture.

Directory Tree

Essential Files

File	Description
`tasks.c`	Implements task creation, delays, blocking, context switching, and scheduling algorithms.
`queue.c`	Provides queues, semaphores, and mutexes.
`list.c`	Internal linked-list operations used for managing ready, delayed, and blocked task lists.
`timers.c`	Software timers and timer service task.
`croutine.c`	Co-routines (rarely used today).

Table 1: Core FreeRTOS Kernel Files

Why this matters

The entire RTOS kernel functionality is built from only a few small C files. This demonstrates FreeRTOS's philosophy:

- minimal
- modular
- easy to understand
- deterministic
- highly portable

3.2 The Portable Layer (Architecture-Specific Support)

This section is responsible for adapting the FreeRTOS kernel to different CPU architectures and compilers.

Every processor architecture requires unique assembly code to perform:

- task context switching
- saving/restoring CPU registers
- interrupt control
- starting the scheduler

Directory Tree

FreeRTOS keeps this hardware-dependent code inside `portable`.

Inside this directory, you find subfolders for each supported architecture and compiler e.g. RISC-V.

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Source/portable/GCC/RISC-V$ tree
.
├── Documentation.url
├── chip_extensions.cmake
├── chip_specific_extensions
│   ├── Pulpino_Vega_RV32M1RM
│   │   └── freertos_risc_v_chip_specific_extensions.h
│   ├── RISCV_MTIME_CLINT_no_extensions
│   │   └── freertos_risc_v_chip_specific_extensions.h
│   ├── RISCV_no_extensions
│   │   └── freertos_risc_v_chip_specific_extensions.h
│   ├── RV32I_CLINT_no_extensions
│   │   └── freertos_risc_v_chip_specific_extensions.h
│   └── readme.txt
├── port.c
├── portASM.S
├── portContext.h
├── portmacro.h
└── readme.txt

6 directories, 12 files
```

RISC-V Port Folder

File	Description
`port.c`	CPU-specific code for context switching, scheduler startup, and interrupt management.
`portASM.S`	Critical assembly routines for saving and restoring registers during task switches.
`portmacro.h`	Macro definitions controlling CPU features such as stack alignment and interrupt masking.

Table 2: Portable Files

These define how FreeRTOS talks to the CPU.

Why this matters

This clean separation means:

- FreeRTOS kernel does not depend on CPU architecture
- Only the portable layer changes when switching from ARM to RISC-V to other supported architectures.
- It is possible to port FreeRTOS to new CPUs without touching kernel code A major reason FreeRTOS is widely adopted.

As the Kernel is CPU-independent & the Port layer is CPU-dependent, this makes FreeRTOS runnable on dozens of architectures simply by adding a new port folder without modifying the kernel.

3.3 Memory Allocation Modules

Embedded systems vary in RAM size, fragmentation, and allocation patterns. To support all use-cases, FreeRTOS provides five memory allocation strategies found in `MemMang`.

Directory Tree:

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Source/portable/MemMang$ tree
.
├── ReadMe.url
├── heap_1.c
├── heap_2.c
├── heap_3.c
├── heap_4.c
└── heap_5.c

1 directory, 6 files
```

Heap Implementations Strategies

Heap Type	Features	Use Case
`heap_1.c`	Alloc only, no free	Very small microcontrollers
`heap_2.c`	Simple allocate/free	Low fragmentation, simple tasks
`heap_3.c`	Wraps malloc/free	Systems with standard library
`heap_4.c`	Best fit allocator	Most commonly used
`heap_5.c`	Multi-region allocation	Large or fragmented memory layouts

Table 3: FreeRTOS Memory Allocation Strategies

Why this matters

FreeRTOS lets you choose the best strategy without modifying kernel code.

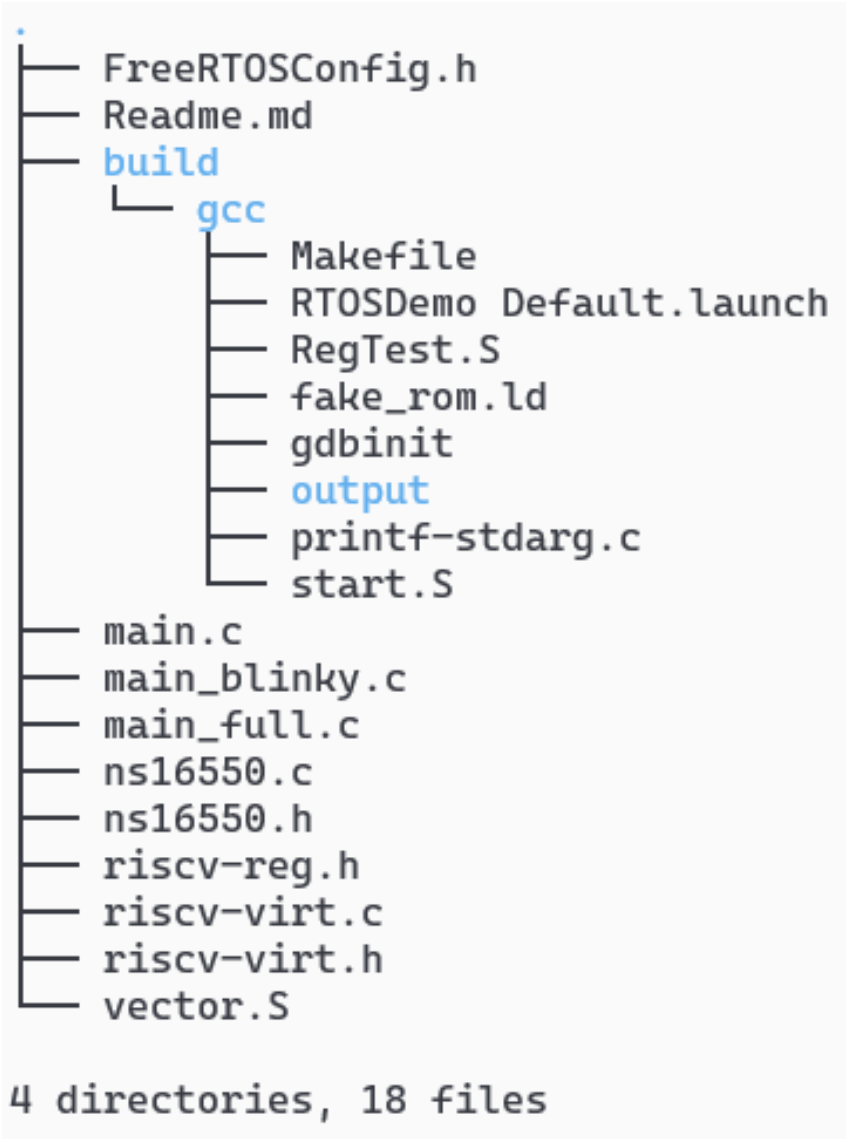
3.4 Demo Directory

This is where actual projects exist. The demo directory includes:

- Complete example applications
- Board support packages
- Correct linker scripts
- Hardware initialization
- FreeRTOSConfig.h
- Hardware drivers (UART, timers)

- Ready-to-build projects for each architecture

Directory Tree



Why this directory is essential

Note: Never start a FreeRTOS project from scratch.

FreeRTOS documentation recommends starting with a demo because it has:

- A fully configured working FreeRTOS RISC-V environment
- Ready scheduling
- UART output working
- The blinky demo (‘main_blinky’)
- The full test suite (‘main_full’)

- Correct architecture port files
- A working build system

3.5 Configuration File

Every FreeRTOS application is controlled by:

`FreeRTOSConfig.h`

This file determines:

- whether preemptive scheduling is used
- tick frequency
- stack sizes
- priority limits
- which APIs are enabled
- whether timers, queues, and interrupts are included
- memory allocator selection

Why this matters

This file adapts FreeRTOS to a specific microcontroller.

It changes the behavior of FreeRTOS more than kernel code does. Many differences between two FreeRTOS projects come from this configuration alone.

3.6 Why the FreeRTOS GitHub Repository Matters for Understanding RTOS

The organization of the FreeRTOS repository demonstrates key RTOS principles:

1. Clean separation of kernel vs. hardware
    - Kernel is independent of hardware i.e. portable
    - Port layer is architecture-specific
    - Demos are application-specific

    This shows excellent software modularity.

2. Minimal kernel design

    Only a few files implement all RTOS features. This allows understanding of the scheduler and tasks in short time.

3. High transparency

   Everything is open and inspectable:

   - Assembly context switching
   - Priority scheduling
   - Queue implementation
   - Timer service mechanism
   - Memory allocation strategies
   - Test code

   This transparency makes FreeRTOS ideal for learning operating system internals.

4. Ready-to-use demonstration environments

   The demos show:

   - Typical FreeRTOS usage
   - Correct initialization
   - Inter-task communication patterns
   - Timer callbacks
   - Scheduling behavior

   These demos form the base from which real projects can be developed.

4 Why We Used RISC-V, QEMU, and Supporting Tools

Before detailing the implementation, first is the introduction of the target hardware architecture (RISC-V) and emulation platform (QEMU) that enable studying FreeRTOS without physical hardware.

4.1 Why RISC-V Was Chosen

RISC-V is an open, extensible Instruction Set Architecture (ISA) designed for education, research, and industry-grade embedded systems. Unlike proprietary ISAs such as ARM or x86, RISC-V is:

- **Open-source:** No licensing cost, making it ideal for academic experimentation.
- **Modular and simple:** The base ISA is intentionally clean and small, which allows easy understanding how an operating system interacts with hardware.

- **Widely supported by FreeRTOS:** FreeRTOS maintains an official RISC-V port, including context-switch assembly routines and timer configurations.
- **Architecture-neutral learning:** Its clarity makes it easy to examine scheduling, task switching, interrupt control, and memory management.

RISC-V is the best choice because it reveals the internal mechanisms of an RTOS without hiding hardware behavior behind proprietary architectures.

4.2 Why QEMU Was Used

QEMU is a full-system virtual machine emulator that can simulate processors, boards, peripherals, and memory. For this project, QEMU is essential because:

- It provides a complete RISC-V virtual board (virt machine) that behaves like physical hardware.
- It runs FreeRTOS images without requiring real RISC-V hardware.
- It supports UART output, which allows us to print task messages and verify scheduling behavior.
- It easily loads ELF files (Executable and Linkable Format) produced by GCC.
- It supports debugging flags (‘`-s`’, ‘`-S`’, GDB), which are extremely useful for OS-level learning.

Because FreeRTOS is deeply tied to CPU instructions, interrupts, and memory layout, QEMU provides a controlled, replicable environment to safely test these low-level features.

4.3 Why a RISC-V Cross Compiler Was Needed

Normal compilers (like `gcc`) generate binaries for your host machine (x86 Linux or Windows).

To compile FreeRTOS applications for RISC-V, we need a cross-compiler i.e. `riscv-none-elf-gcc`

This toolchain:

- Produces bare-metal binaries (no operating system)
- Generates correct RISC-V machine code
- Links code with the FreeRTOS port layer
- Creates an output ELF file loadable in QEMU

Without this cross-compiler, the FreeRTOS kernel and tasks would not run on the emulated RISC-V machine.

5 Environment Setup

The methodology section explains the step-by-step process followed to set up the environment, compile FreeRTOS, integrate our task, and run it on QEMU.

5.1 Creating the Working Directory

The setup begins with creating a dedicated workspace for the RTOS project:

```
$ mkdir os_project
$ export OS_PROJECT=~/os_project
$ cd os_project
```

- Define an environment variable, OS_PROJECT, which holds the absolute path to the project's root directory. This variable is exported to the shell's environment so that it can be easily referenced by subsequent commands, scripts, and other variables (like the `$PATH` variable) to simplify navigation and reference the project workspace.

This ensures that all downloaded tools, cloned repositories, and build outputs remain organized and isolated.

5.2 Cloning the FreeRTOS Repository

Inside the project directory:

```
$ git clone https://github.com/FreeRTOS/FreeRTOS.git
```

This creates a subdirectory named FreeRTOS, containing all the Github files including:

- The Kernel
- The portable layer
- Demo folder
- The RISC-V port
- The QEMU RISC-V demo project
- The FreeRTOS configuration files

5.3 Installing the RISC-V Cross Compiler

The RISC-V GNU toolchain is required to compile FreeRTOS applications into RISC-V machine code.

Download the compiler

```
$ wget https://github.com/xpack-dev-tools/riscv-none-elf-gcc-xpack/
   ↪ releases/download/v14.2.0-2/xpack-riscv-none-elf-gcc-14.2.0-2-
   ↪ linux-x64.tar.gz
```

This utilizes the `wget` utility to download a prebuilt version of the RISC-V compiler (v14.2.0-2) from official GitHub release.

Extract the archive

```
$ tar -zxvf xpack-riscv-none-elf-gcc-14.2.0-2-linux-x64.tar.gz
```

The `tar` utility is executed to decompress and extract the downloaded the toolchain into a directory containing:

- `riscv-none-elf-gcc`
- `riscv-none-elf-gdb`
- Binaries, libraries, and include files

Add compiler to PATH

```
$ export PATH=$PATH:$OS_PROJECT/xpack-riscv-none-elf-gcc-14.2.0-2/
   ↪ bin
```

This allows the terminal to find the compiler globally.

Installing QEMU RISC-V

QEMU emulates the RISC-V hardware environment.

Download QEMU

```
$ wget https://github.com/xpack-dev-tools/qemu-riscv-xpack/releases/
   ↪ download/v8.2.2-1/xpack-qemu-riscv-8.2.2-1-linux-x64.tar.gz
```

Extract the archive

```
$ tar -zxvf xpack-qemu-riscv-8.2.2-1-linux-x64.tar.gz
```

Add QEMU to PATH

```
$ export PATH=$PATH:$OS_PROJECT/xpack-qemu-riscv-8.2.2-1/bin
```

This ensures the command `qemu-system-riscv32` runs correctly from any directory.

6 Operationalization and Troubleshooting of FreeRTOS

Following the environment setup, the next phase involved correctly configuring, restoring, and executing the FreeRTOS project for the RISC-V QEMU virtual platform. Since the FreeRTOS build system relies on modular sources and version-controlled subdirectories, several structural corrections and configuration adjustments were required before the RTOS could successfully compile and execute on QEMU.

6.1 Restoring Missing Source Files via Git Submodules

Upon initial inspection, the `FreeRTOS/Source` directory appeared empty. This behavior is expected because FreeRTOS uses Git submodules, which do not automatically populate during a standard `git clone`.

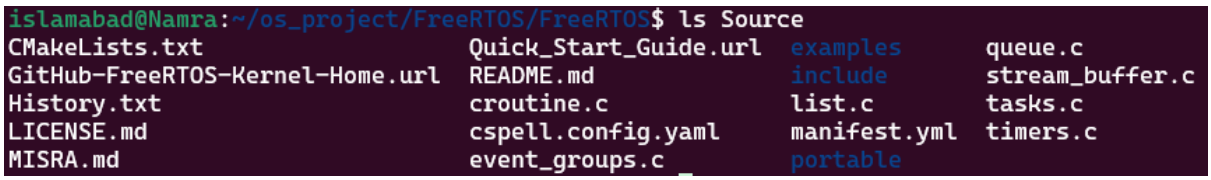
To restore the complete source tree, the following command was executed in the FreeRTOS directory:

```
$ git submodule update --init --recursive
```

This command initializes and recursively populates all nested submodules used by FreeRTOS, ensuring that:

- The kernel sources,
- Portable layer sources (RISC-V-specific),
- Demo files,
- FreeRTOS+ components

are all correctly downloaded.



6.2 Navigating to the RISC-V QEMU Demo Build Directory

FreeRTOS provides multiple architecture-specific demos. The project used the following path: `FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/build/gcc`

This directory contains:

- the architecture-specific Makefile,
- build rules for the RISC-V GCC toolchain,
- and output paths for 'RTOSDemo.elf'.

6.3 Makefile Modifications

TOOLCHAIN PREFIX Adjustment

- The FreeRTOS demo expects a 64-bit compiler, but the installed toolchain is RV32 (riscv-none-elf-gcc). The project uses a self-contained xPack compiler so using the default values results in:
  - compiler not found
  - incorrect architecture flags
  - build failures
- Specifying `TOOLCHAIN_PREFIX` ensures the Makefile uses the correct xPack RV32 toolchain.
- Linking must use the same compiler family. If CC and LD differ, ABI mismatches and linker errors occur.

Original:

```
CC = riscv64-unknown-elf-gcc
LD = riscv64-unknown-elf-gcc
SIZE = riscv64-unknown-elf-size
```

Modified:

```
TOOLCHAIN_PREFIX = /home/islamabad/os_project/xpack-riscv-none-elf-
    ↪ gcc-14.2.0-2/bin
CC = $(TOOLCHAIN_PREFIX)/riscv-none-elf-gcc
LD = $(TOOLCHAIN_PREFIX)/riscv-none-elf-gcc
SIZE = $(TOOLCHAIN_PREFIX)/riscv-none-elf-size
```

Fixing Invalid Variable Syntax

Original Form:

```
SOURCE_FILES += (COMMON_DEMO_FILES)/AbortDelay.c
```

Correct Form:

```
SOURCE_FILES += $(COMMON_DEMO_FILES)/AbortDelay.c
```

Applied Fix:

```
sed -i 's/+= (/+= $(/g' Makefile
```

GNU Make requires `$(VAR)`. Without this fix, the Makefile tries to compile literal paths like: `(COMMON_DEMO_FILES)/AbortDelay.c` which causes "file not found" errors. Correcting the syntax restores proper file expansion.

Fixing GCC Version Extraction

xPack GCC does not print compiler information in the format expected by FreeRTOS, causing:

- empty `GCC_VERSION`
- incorrect selection of architecture flags (zicsr / non-zicsr)

`-dumpversion` always returns a clean numeric version, making version comparison reliable.

Original Form:

A long pipeline using `grep^\$(CC)`:

```
GCC_VERSION = $(shell $(CC) --version | grep ^$(CC) | sed 's/^.* //g
   ↪ ' | awk -F. '{ printf("%d%02d%02d"), $$1, $$2, $$3 }')
```

Modified

```
GCC_VERSION = $(shell $(CC) -dumpversion | awk -F. '{ printf("%d%02d
   ↪ %02d"), $$1, $$2, $$3 }')
```

Shell Compatibility

Ubuntu uses dash for `/bin/sh`.

Several FreeRTOS Makefile expressions assume bash-like behavior. By simplifying pipelines (e.g., using `-dumpversion`) and correcting variable syntax, the Makefile becomes compatible with both dash and bash.

Rebuilding the FreeRTOS Demo

After all Makefile corrections, the project was cleaned and rebuilt:

```
$ make clean
$ make
```

This step regenerates all object files and produces the final executable: `output/RTOSDemo.elf`

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/build
/gcc$ make clean
Using RV32 build
rm -f ./output/RTOSDemo.elf ./output/*.o ./output/*.d ./output/*.map
```

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/build
/gcc$ make
Using RV32 build
/home/islamabad/os_project/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc
 -I./../../../../Source/portable/GCC/RISC-V/chip_specific_extensions/RV32I_CLINT_no
_extensions -I./../../../../Source/include -I./../../../../Source/portable/GCC/RISC
-V -I./../../../../Demo/Common/include -I./../../../../Demo/RISC-V_RV32_QEMU_VIRT_G
CC -fmessage-length=0 -march=rv32imac_zicsr -mabi=ilp32 -mcmodel=medlow -ffunction-
sections -fdata-sections -Wno-unused-parameter -nostartfiles -g3 -Os --specs=nano.s
pecs -fno-builtin-printf -MMD -MP -c ./../../../../Source/tasks.c -o output/tasks.o
/home/islamabad/os_project/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc
 -I./../../../../Source/portable/GCC/RISC-V/chip_specific_extensions/RV32I_CLINT_no
_extensions -I./../../../../Source/include -I./../../../../Source/portable/GCC/RISC
-V -I./../../../../Demo/Common/include -I./../../../../Demo/RISC-V_RV32_QEMU_VIRT_G
CC -fmessage-length=0 -march=rv32imac_zicsr -mabi=ilp32 -mcmodel=medlow -ffunction-
sections -fdata-sections -Wno-unused-parameter -nostartfiles -g3 -Os --specs=nano.s
pecs -fno-builtin-printf -MMD -MP -c ./../../../../Source/list.c -o output/list.o
/home/islamabad/os_project/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc
 -I./../../../../Source/portable/GCC/RISC-V/chip_specific_extensions/RV32I_CLINT_no
_extensions -I./../../../../Source/include -I./../../../../Source/portable/GCC/RISC
-V -I./../../../../Demo/Common/include -I./../../../../Demo/RISC-V_RV32_QEMU_VIRT_G
CC -fmessage-length=0 -march=rv32imac_zicsr -mabi=ilp32 -mcmodel=medlow -ffunction-
sections -fdata-sections -Wno-unused-parameter -nostartfiles -g3 -Os --specs=nano.s
pecs -fno-builtin-printf -MMD -MP -c ./../../../../Source/queue.c -o output/queue.o
/home/islamabad/os_project/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc
```

6.4 Running the Demo on QEMU

The final binary was executed using QEMU with the following command:

```
$ qemu-system-riscv32 -nographic -machine virt -net none -chardev
   ↪ stdio,id=con,mux=on -serial chardev:con -mon chardev=con,mode=
   ↪ readline -bios none -s --kernel ./output/RTOSDemo.elf
```

Purpose of the parameters

- `-machine virt`: creates a generic RISC-V virtual board compatible with FreeRTOS.
- `-nographic`: disables GUI; everything runs on terminal.
- `-bios none`: instructs QEMU to directly load the ELF file instead of a firmware stub.
- `-s`: enables GDB server for debugging (optional).
- `--kernel ...`: loads the compiled FreeRTOS demo.

This launches the system and begins execution of tasks, timers, and any user-inserted test functions.

6.5 FreeRTOS Blinky Demo Modes (Mode 0 & Mode 1)

FreeRTOS provides two demonstration modes within the RISC-V QEMU project. These modes are selected through the macro `mainCREATE_SIMPLE_BLINKY_DEMO_ONLY` in `main.c`:

- **0** : Full Demo Mode (`main_full()`)

- **1** : Minimal Blinky Mode (`main_blinky()`)

Each mode activates its own set of source files and program flow, enabling comparison between a feature-rich FreeRTOS demonstration and a minimal task-and-timer example.

6.5.1 Full Demo Mode (Mode 0)

Characteristics:

- Contains the complete FreeRTOS test suite:
  - Queue tests
  - Semaphore tests
  - Timer tests
  - Interrupt nesting tests
  - Memory allocation tests
- Much more CPU-intensive and produces verbose output.
- More tasks run concurrently.
- Runs a larger number of tasks with varying priorities.
- Excellent for stress testing the RTOS setup and environment, as it shows all FreeRTOS features and allows to catch stack, heap, or interrupt nesting issues early.

Primary Output Messages:

```
FreeRTOS Demo Start # Beginning of execution
FreeRTOS Demo SUCCESS: : <tickcount> # When all internal checks pass
   ↪ and the demo tasks execute correctly
```

These messages are generated by the Check Task (`prvCheckTask`). Its purpose is to:

- Verify that all standard demo tasks are executing without error.
- Periodically report system health.

Behavior: The check task executes continuously, monitoring internal demo sub-tasks and producing a success message at fixed intervals when no inconsistency is detected.

If priorities align, user tasks may execute first; otherwise the demo initialization message prints first.

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/build/gcc$ qemu-system-ris
cv32 -nographic -machine virt -net none -chardev stdio,id=con,mux=on     -serial chardev:con -mon charde
v=con,mode=readline -bios none   -s --kernel ./output/RTOSDemo.elf
FreeRTOS Demo Start
FreeRTOS Demo SUCCESS: : 5034
FreeRTOS Demo SUCCESS: : 10034
FreeRTOS Demo SUCCESS: : 15034
FreeRTOS Demo SUCCESS: : 20034
FreeRTOS Demo SUCCESS: : 25034
FreeRTOS Demo SUCCESS: : 30034
FreeRTOS Demo SUCCESS: : 35034
FreeRTOS Demo SUCCESS: : 40034
FreeRTOS Demo SUCCESS: : 45034
FreeRTOS Demo SUCCESS: : 50034
FreeRTOS Demo SUCCESS: : 55034
FreeRTOS Demo SUCCESS: : 60034
FreeRTOS Demo SUCCESS: : 65034
FreeRTOS Demo SUCCESS: : 70034
FreeRTOS Demo SUCCESS: : 75034
FreeRTOS Demo SUCCESS: : 80034
FreeRTOS Demo SUCCESS: : 85034
FreeRTOS Demo SUCCESS: : 90034
QEMU: Terminated
```

6.5.2 Blinky Demo Mode (Mode 1)

Characteristics:

- Contains very small tasks:
  - A “blink” task that prints a message in QEMU.
  - A check/timer task.
  - Optional user-defined tasks
- Extremely lightweight.
- Intended only to verify scheduler correctness and ensure the RTOS boots correctly, context switches, and runs periodic timers.
- Less chance of complex test errors.

Primary Output Messages:

```
Message received from task
Message received from software timer
```

These messages originate from:

- A queue receive task
- A periodic software timer callback

Behavior:

Starts almost immediately because no heavy initialization is required. This minimal mode illustrates core RTOS primitives using only a small number of tasks. Output is simpler and ideal for verifying correct FreeRTOS configuration under QEMU.

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/build/gcc$ qemu-system-ris
cv32 -nographic -machine virt -net none -chardev stdio,id=con,mux=on     -serial chardev:con -mon charde
v=con,mode=readline -bios none   -s --kernel ./output/RTOSDemo.elf
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from software timer
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from software timer
```

7 Multiplication Task Implementation

A custom multiplication task is introduced to extend the demonstration by adding periodic numeric computation and console output. It introduces a periodic arithmetic workload that prints multiplication results every second. Its design follows standard FreeRTOS task-creation principles and integrates cleanly with both `main_full.c` and `main_blinky.c`.

Both `mult_task.h` and `mult_task.h` are implemented in the same directory as the `main.c`: `FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/`

7.1 Public Interface: `mult_task.h`

Purpose: define the interface, configuration macros, and task creation prototype.

```
#ifndef MULT_TASK_H // include guard to prevent multiple inclusion
#define MULT_TASK_H

// Function prototype for your multiplication task
void createMultiplicationTask(void);

#endif /* MULT_TASK_H */
```

1. **Include Header guard:**

   Prevents multiple inclusions of the header during compilation, avoiding redefinition errors.

2. **Public API Function**:

   `void createMultiplicationTask(void);` This is the only function exposed to the rest of the program. It allows the main application files to create the multiplication

task without needing access to internal implementation details.

7.2 Task Implementation: `mult_task.c`

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

- `<stdio.h>`is used for printing output to the console via `printf()`.
- `FreeRTOS.h` and `task.h` are used to provide FreeRTOS API functions such as `xTaskCreate()` and `vTaskDelay()`.
- `mult_task.h` provides the task-creation prototype.

7.2.1 Task Priority

```
/* Priority for multiplication task */
#define MULT_TASK_PRIORITY ( tskIDLE_PRIORITY + 1 )
```

- Ensures the multiplication task runs above the idle task, but below more important system tasks.
- Priority can be adjusted depending on when you want its output to appear relative to other demo tasks.

7.2.2 Task Creation Function

```
/* Helper function to create the task, call this from main_blinky()
   ↪ */
void createMultiplicationTask( void )
{
   xTaskCreate(
      vMultiplicationTask, // Task function
      "Multiplier", // Task name
      configMINIMAL_STACK_SIZE, // Stack size
      NULL, // Parameters
      MULT_TASK_PRIORITY, // Priority
      NULL // Task handle not needed
   );
}
```

- `xTaskCreate()` registers the multiplication task with the scheduler.
- ”Multiplier” is the task name used for debugging.

- `configMINIMAL_STACK_SIZE` is sufficient because the task performs simple integer operations.
- NULL parameter and task handle indicate no external parameters and no need to store task handles.
-
- The scheduler (`vTaskStartScheduler()`) will automatically start this task once invoked in `main()`.

7.2.3 Task Function Logic

```
/* Function prototype */
void vMultiplicationTask( void * pvParameters );

/* Task function */
void vMultiplicationTask( void * pvParameters )
{
   (void) pvParameters;

   uint32_t a = 2;
   uint32_t b = 3;
   uint32_t result;

   for( ;; )
   {
      result = a * b;
      printf("Multiplication: %u * %u = %u\r\n", a, b, result);

      /* Update values (optional: simple demo) */
      a++;
      b++;

      vTaskDelay(pdMS_TO_TICKS(1000)); // Delay 1 second
   }
}
```

1. Parameters `(void) pvParameters;` :

   The task does not use incoming parameters, so they are safely ignored.

2. Internal Variables

   - a and b are initialized to 2 and 3.
   - They increment after each iteration.
   - `result` stores the multiplication output.

3. Infinite Loop (`for( ;; )`)

   All FreeRTOS tasks run indefinitely unless explicitly deleted.

4. Console Output

   Prints the current operand values and their product.

   This output appears interleaved with the standard FreeRTOS demo messages depending on scheduling.

5. Delay

   - Blocks the task for 1000 ms, allowing other tasks to run.
   - Demonstrates cooperative multitasking and CPU sharing

8 Integration of the Multiplication Task into the FreeRTOS Demo

Integrating a custom task into FreeRTOS requires ensuring that the scheduler is aware of the task, that the build system compiles it, and that the application includes it during system startup. FreeRTOS follows a modular architecture in which every task is implemented as a separate C file and registered in the main application before the scheduler begins execution. The multiplication task follows the same model.

This requires three integration steps:

- Adding the new source file to the build system (Makefile).
- Including its header into the demo main files.
- Registering the task before the scheduler starts.

8.1 Integrating into Makefile

FreeRTOS demos are controlled by a central Makefile that lists all C and assembly files used during the build. This Makefile does not automatically detect new files, so every custom module must be explicitly added.

The multiplication task is placed in the demo project directory (`$(DEMO_PROJECT)`), so the Makefile requires:

```
SOURCE_FILES += $(DEMO_PROJECT)/mult_task.c
```

This step is required because:

- FreeRTOS does not auto-discover files.

  If a file is not listed under `SOURCE_FILES`, it is completely invisible to the build system.

- The compiler must translate the task's C code into an object file. Without adding this entry, the file is never compiled, and no `.o` object is generated.
- Both `main_blinky.c` and `main_full.c` will call `createMultiplicationTask()`. So if `mult_task.c` is not part of the build, the linker produces an "undefined reference" error.
- Adding the multiplication task here ensures it behaves like any other FreeRTOS demo component.

8.2 Integrating into `main_blinky.c`

To ensure that the multiplication task is created when running the Blinky Demo:

1. **Include the header**

   The multiplication task's header (`mult_task.h`) contains the prototype and this function must be visible to any file that intends to create the task. Therefore, both demo entry points need to include this header.

   Add the following near the top of the file:

   ```
   #include "mult_task.h"
   ```

   This step guarantees that the main application can correctly reference and invoke the task.

2. **Create the task**

   Inside `main_blinky()`, the task-creation function must be called at the point where other tasks are created and before the scheduler starts.

   ```
   createMultiplicationTask();
   ```

   This function must appear before:

   `vTaskStartScheduler();`

   Once the scheduler starts, task creation is still technically permitted, but:

   - Startup sequence becomes inconsistent,
   - Ordering of initial execution changes,
   - FreeRTOS demos assume all tasks exist at start.

8.3 Integrating into `main_full.c`

The integration steps for the Full Demo mirror those in Blinky mode.

1. **Include the header**

   ```
   #include "mult_task.h"
   ```

2. **Create the task**

   Inside `main_blinky()`, before the scheduler is started, add:

```
createMultiplicationTask();
```

   Placement rule:

   This function must appear before `vTaskStartScheduler();` and after standard demo initializations.

8.4 Scheduling Behavior

Integrating the multiplication task adds a custom workload to the comprehensive FreeRTOS demo suite. The task behaves like any other FreeRTOS task:

- It has its own priority
- It uses `vTaskDelay()` for periodic execution
- It is managed by the same scheduler as all test tasks

8.5 Conceptual Role of the Task within the Demo Environment

Adding a custom task demonstrates that:

1. FreeRTOS is modular: New tasks integrate cleanly without modifying kernel code.
2. Application extensions are isolated: Only Makefile and main files need modifications.
3. Real-time scheduling principles remain unchanged: Priorities, stack sizes, and delays work exactly like standard demo tasks.
4. The system remains deterministic: The task follows predictable periodic execution.

This mirrors typical embedded development, where custom functionality is added to a base RTOS framework.

9 Build Output and Memory Footprint Analysis

After successful compilation of the FreeRTOS demo and integration of the multiplication task, the build process concludes with a size analysis of the generated ELF binary using the `riscv-none-elf-size` utility. This step is automatically executed at the end of the `make` process and provides insight into the program's memory usage.

The size report is particularly important in embedded and RTOS-based systems, where memory resources are limited and must be carefully managed.

9.1 Full Demo Mode (Mode 0)

```
/home/islamabad/os_project/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-siz
e ./output/RTOSDemo.elf
   text    data     bss     dec     hex filename
  65406      84   89232  154722   25c62 ./output/RTOSDemo.elf
```

9.1.1 Segment Explanation

Text: 65,406 bytes

Represents executable instructions, including:

- FreeRTOS kernel code
- RISC-V porting layer
- All demo test tasks (queues, semaphores, timers, memory tests)
- User-defined multiplication task

The large text size reflects the comprehensive nature of the Full Demo, which includes extensive test logic and validation routines.

Data: 84 bytes

Contains global and static variables that are initialized at compile time. This value is relatively small, indicating limited use of pre-initialized global data.

BSS: 89,232 bytes

Represents uninitialized global and static variables, including:

- Task stacks
- RTOS control structures
- Queue and semaphore storage
- Timer objects

The large BSS segment is expected in Full Demo mode due to the high number of concurrently created tasks and test objects.

Total: 154,722 bytes / 0x25C62

Reflects the complete memory footprint of the system image loaded into QEMU.

9.2 Blinky Demo Mode (Mode 1)

```
_exit -Wl,--wrap=_puts -o ./output/RTOSDemo.elf
/home/islamabad/os_project/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-siz
e ./output/RTOSDemo.elf
   text    data     bss     dec     hex filename
  11156      28   86064   97248   17be0 ./output/RTOSDemo.elf
```

9.2.1 Segment Explanation

Text: 11,156 bytes

Significantly smaller than Full Demo mode because:

- Only minimal demo tasks are compiled
- No extensive test suites are included
- The kernel is exercised with a small number of tasks and timers

Data: 28 bytes

Minimal initialized data, consistent with the lightweight nature of Blinky mode.

BSS: 86,064 bytes

Still relatively large because:

- Task stacks dominate memory usage
- RTOS internal structures are still required
- The multiplication task allocates its own stack

Total: 97,248 bytes / 0x17BE0

Confirms that Blinky mode has a substantially smaller memory footprint, making it suitable for basic validation and low-resource systems.

9.3 Comparative Analysis

Mode	Text Size	Data Size	BSS Size	Total Size
Full Demo	High	Low	High	Highest
Blinky	Very Low	Very Low	High	Lower

The comparison demonstrates that:

- Text size scales with feature complexity
- BSS size is dominated by task stacks and RTOS objects
- Blinky mode is optimized for verification, while Full Demo mode is optimized for stress testing and validation

10 Output and Discussion

This section analyzes the runtime behavior of the FreeRTOS RISC-V demo executed on QEMU after successful compilation, Makefile correction, demo-mode selection, and integration of the custom multiplication task. The discussion focuses on task execution order, scheduling behavior, priority interactions and observable console output, without interpreting numerical results.

10.1 Execution Overview

After launching the compiled executable ('RTOSDemo.elf') on the RISC-V virt machine in QEMU, the FreeRTOS scheduler is started. At this point, all tasks that were created during system initialization-including standard demo tasks and the user-defined multiplication task-become eligible for scheduling.

The output is a consequence of priorities, initial ready states, blocking behavior ('vTaskDelay', timers, queues) & FreeRTOS preemptive scheduler.

The observed console output reflects the interaction between:

- Internal FreeRTOS demo tasks
- Periodic check and timer tasks
- Queue-based communication tasks
- The user-defined multiplication task

Because FreeRTOS is a preemptive, priority-based RTOS, the order in which messages appear on the console directly reflects scheduler decisions rather than source-code order.

10.2 Output Behavior in Full Demo Mode (Mode 0)

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/build/gcc
$ qemu-system-riscv32 -nographic -machine virt -net none -chardev stdio,id=con,mux=on
   -serial chardev:con -mon chardev=con,mode=readline -bios none   -s --kernel ./output
/RTOSDemo.elf
Multiplication: 2 * 3 = 6
FreeRTOS Demo Start
Multiplication: 3 * 4 = 12
Multiplication: 4 * 5 = 20
Multiplication: 5 * 6 = 30
Multiplication: 6 * 7 = 42
Multiplication: 7 * 8 = 56
FreeRTOS Demo SUCCESS: : 5054
Multiplication: 8 * 9 = 72
QEMU: Terminated
```

10.2.1 Why the Multiplication Task Runs Before "FreeRTOS Demo Start"

In the FreeRTOS RISC-V Full Demo, multiple tasks are created inside `main_full()` before the scheduler is started. These include internal test tasks as well as the check task (`prvCheckTask`), which is responsible for printing:

```
FreeRTOS Demo Start
```

and later:

```
FreeRTOS Demo SUCCESS
```

The key points are:

- The multiplication task is created before `vTaskStartScheduler()`.

- When the scheduler starts, it immediately selects the highest-priority ready task.
- The check task (`prvCheckTask`) is periodic and typically begins by sleeping on a timing function such as `vTaskDelayUntil()`.

As a result:

- At tick 0, the multiplication task is ready immediately.
- The check task may not yet be runnable.
- The scheduler therefore runs the multiplication task first.

Even if the multiplication task has equal or lower priority, it can still execute first if the check task is blocked during its initial timing interval.

Note: *This behavior is normal and expected in FreeRTOS and indicates correct scheduler operation.*

10.2.2 Role of the Check Task Messages

The messages:

```
FreeRTOS Demo Start
```

```
FreeRTOS Demo SUCCESS: <tick count>
```

are generated by the check task, whose purpose is to:

- Monitor all internal demo tasks
- Verify that queues, semaphores, timers, and interrupts operate correctly
- Periodically report system health

The appearance of the SUCCESS message confirms that:

- All FreeRTOS demo tasks are running correctly
- The system remains stable after adding the multiplication task
- No scheduling, stack, or memory errors were introduced

10.2.3 Why Other FreeRTOS Demo Tasks Do Not Appear in the Output

Although many demo tasks are created with equal or higher priority than the multiplication task, they do not produce visible output at startup.

This is because:

- Most official FreeRTOS demo tasks perform internal self-tests only.
- They update shared status variables rather than printing messages.
- Output is intentionally centralized in the check task (`prvCheckTask`).

Therefore:

- These tasks may execute before both the multiplication task and the check task.
- They simply produce no console output.
- Their activity is only reflected indirectly through the SUCCESS message.

This explains why the first visible output is either the multiplication task or the check task message, even though other tasks are running in the background.

10.3 Output Behavior in Blinky Demo Mode (Mode 1)

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/build/gcc
$ qemu-system-riscv32 -nographic -machine virt -net none -chardev stdio,id=con,mux=on
   -serial chardev:con -mon chardev=con,mode=readline -bios none   -s --kernel ./output
/RTOSDemo.elf
Multiplication: 2 * 3 = 6
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Multiplication: 3 * 4 = 12
Message received from task
Message received from task
Message received from task
Message received from task
Message received from software timer
Message received from task
Multiplication: 4 * 5 = 20
Message received from task
Message received from task
QEMU: Terminated
```

10.3.1 Interpretation of Blinky Output

Blinky mode is a minimal FreeRTOS demonstration intended to verify:

- Scheduler startup
- Task switching
- Software timer execution
- Queue communication

The messages:

`Message received from task`

`Message received from software timer`

originate from:

- A queue-receiving task.
- A periodic software timer callback.

Because Blinky mode has very little initialization overhead, these tasks begin executing almost immediately, producing frequent and repetitive output.

10.3.2 Scheduling Interaction with the Multiplication Task

The multiplication task executes periodically and blocks itself using `vTaskDelay(pdMS_TO_TICKS(100`

During this blocked interval:

- Queue tasks and timer callbacks repeatedly execute.
- The scheduler alternates between ready tasks of equal priority.
- Multiple Blinky messages appear between multiplication outputs.

This behavior demonstrates:

- Correct preemptive scheduling.
- Proper task blocking and unblocking.
- Cooperative CPU sharing among tasks.

10.4 Key Observations

1. The multiplication task integrates cleanly into both demo modes.
2. Task priorities are respected by the FreeRTOS scheduler.
3. Blocking delays allow fair CPU time distribution.
4. Internal FreeRTOS demo tasks continue to function correctly.
5. Output interleaving confirms correct multitasking behavior.

11 Demonstration of Synchronization and Deadlock Concepts

To reinforce the practical understanding of multitasking, concurrency control, and scheduling in FreeRTOS, two additional demonstrations were implemented in **Blinky Demo Mode (Mode 1)**:

- Mutual Exclusion using a mutex
- Intentional deadlock creation using multiple mutexes

These demonstrations go beyond theoretical discussion and explicitly show how synchronization primitives behave at runtime under a preemptive, priority-based RTOS scheduler.

11.1 Mutex Demonstration (Mutual Exclusion)

11.1.1 Objective

The mutex demonstration illustrates how FreeRTOS prevents race conditions when multiple concurrent tasks attempt to access a shared resource. It shows that:

- Only one task may enter a critical section at a time
- Other tasks are blocked until the mutex is released
- Task execution becomes serialized around shared resources

This directly demonstrates the concept of **mutual exclusion** in a multitasking environment.

11.1.2 Design and Integration

Two tasks are created in Blinky mode. Both tasks attempt to print messages to the console, which is treated as a shared resource. A mutex is used to protect the printing operation.

The mutex and tasks are implemented in a separate module and integrated into the Blinky demo using the same modular approach as other custom tasks in this project.

11.1.3 Mutex Code Implementation

Code: **mutex_task.h**

```
#ifndef MUTEX_TASK_H
#define MUTEX_TASK_H

void createMutexDemoTasks(void);

#endif // MUTEX_TASK_H
```

Code: **mutex_task.h**

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

static SemaphoreHandle_t xMutex;
static int sharedCounter = 0;

static void vMutexTask1(void *pvParameters)
{
   for (;;)
   {
```

```
      xSemaphoreTake(xMutex, portMAX_DELAY);
      printf("Task 1 entered critical section\n");

      sharedCounter++;
      printf("Task 1 counter = %d\n", sharedCounter);

      vTaskDelay(pdMS_TO_TICKS(200));

      printf("Task 1 leaving critical section\n");
      xSemaphoreGive(xMutex);

      vTaskDelay(pdMS_TO_TICKS(1000));
   }
}

static void vMutexTask2(void *pvParameters)
{
   for (;;)
   {
      xSemaphoreTake(xMutex, portMAX_DELAY);
      printf("Task 2 entered critical section\n");

      sharedCounter++;
      printf("Task 2 counter = %d\n", sharedCounter);

      vTaskDelay(pdMS_TO_TICKS(200));

      printf("Task 2 leaving critical section\n");
      xSemaphoreGive(xMutex);

      vTaskDelay(pdMS_TO_TICKS(1000));
   }
}

void createMutexDemoTasks(void)
{
   xMutex = xSemaphoreCreateMutex();
   configASSERT(xMutex != NULL);

   xTaskCreate(
      vMutexTask1,
      "MutexT1",
      configMINIMAL_STACK_SIZE,
      NULL,
      tskIDLE_PRIORITY + 2,
```

```
        NULL
    );

    xTaskCreate(
        vMutexTask2,
        "MutexT2",
        configMINIMAL_STACK_SIZE,
        NULL,
        tskIDLE_PRIORITY + 1,
        NULL
    );
}
```

Key characteristics of the implementation:

- A single mutex is created using `xSemaphoreCreateMutex()`.
- Two tasks attempt to take the mutex before accessing the shared resource.
- Each task releases the mutex after completing its operation.
- A delay is introduced to clearly observe task alternation.

11.1.4 Observed Output

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GC
C/build/gcc$ qemu-system-riscv32 -nographic -machine virt -net none -chardev
 stdio,id=con,mux=on     -serial chardev:con -mon chardev=con,mode=readline
-bios none   -s --kernel ./output/RTOSDemo.elf
Task 1 entered critical section
Task 1 counter = 1
Task 1 leaving critical section
Message received from task
Task 2 entered critical section
Task 2 counter = 2
Message received from task
Task 2 leaving critical section
Message received from task
Message received from task
Message received from task
Task 1 entered critical section
Task 1 counter = 3
Message received from task
QEMU: Terminated
```

11.1.5 Discussion

The output confirms that:

- Tasks do not interleave within the critical section
- Mutual exclusion is enforced correctly by the FreeRTOS kernel

- Blocking occurs automatically when a mutex is unavailable

This behavior validates correct synchronization, demonstrates safe shared-resource access, and confirms correct scheduler interaction with mutex primitives.

11.2 Deadlock Demonstration

11.2.1 Objective

The deadlock demonstration intentionally creates a circular wait condition to show how improper synchronization design can halt system progress. The goal is to demonstrate:

- Hold-and-wait behavior
- Circular resource dependency
- Permanent task blocking without kernel error

This confirms that while FreeRTOS provides synchronization mechanisms, deadlock prevention remains the responsibility of the application developer.

11.2.2 Design and Integration

Two mutexes are created, representing two independent shared resources. Two tasks are then designed as follows:

- Task 1 locks Mutex A, then attempts to lock Mutex B
- Task 2 locks Mutex B, then attempts to lock Mutex A

Both tasks run at the same priority in Blinky mode, maximizing the likelihood of circular wait.

Note: *Independent* `blinky_mode` *tasks are not involved in the deadlock.*

11.2.3 Deadlock Code Implementation

Code: **deadlock_task.h**

```
#ifndef DEADLOCK_TASK_H
#define DEADLOCK_TASK_H

void createDeadlockDemoTasks(void);

#endif /* DEADLOCK_TASK_H */
```

Code: **deadlock_task.h**

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

```
#include "deadlock_task.h"

static SemaphoreHandle_t xMutexA;
static SemaphoreHandle_t xMutexB;

static void vDeadlockTask1(void *pvParameters)
{
   for (;;)
   {
      printf("Task 1 trying to lock Mutex A\n");
      xSemaphoreTake(xMutexA, portMAX_DELAY);
      printf("Task 1 locked Mutex A\n");

      vTaskDelay(pdMS_TO_TICKS(500)); // force overlap

      printf("Task 1 trying to lock Mutex B\n");
      xSemaphoreTake(xMutexB, portMAX_DELAY);
      printf("Task 1 locked Mutex B\n");

      /* Never reached in deadlock */
      xSemaphoreGive(xMutexB);
      xSemaphoreGive(xMutexA);
   }
}

static void vDeadlockTask2(void *pvParameters)
{
   for (;;)
   {
      printf("Task 2 trying to lock Mutex B\n");
      xSemaphoreTake(xMutexB, portMAX_DELAY);
      printf("Task 2 locked Mutex B\n");

      vTaskDelay(pdMS_TO_TICKS(500)); // force overlap

      printf("Task 2 trying to lock Mutex A\n");
      xSemaphoreTake(xMutexA, portMAX_DELAY);
      printf("Task 2 locked Mutex A\n");

      /* Never reached in deadlock */
      xSemaphoreGive(xMutexA);
      xSemaphoreGive(xMutexB);
   }
}
```

```
void createDeadlockDemoTasks(void)
{
   xMutexA = xSemaphoreCreateMutex();
   xMutexB = xSemaphoreCreateMutex();

   configASSERT(xMutexA);
   configASSERT(xMutexB);

   xTaskCreate(
      vDeadlockTask1,
      "DeadlockT1",
      configMINIMAL_STACK_SIZE,
      NULL,
      tskIDLE_PRIORITY + 2,
      NULL
   );

   xTaskCreate(
      vDeadlockTask2,
      "DeadlockT2",
      configMINIMAL_STACK_SIZE,
      NULL,
      tskIDLE_PRIORITY + 2,
      NULL
   );
}
```

Key design elements:

- Two mutexes created using `xSemaphoreCreateMutex()`.
- Opposite locking order between tasks.
- Blocking calls with `portMAX_DELAY`.
- No timeout or recovery logic.

11.2.4 Observed Output

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_
VIRT_GCC/build/gcc$ qemu-system-riscv32 -nographic -machine virt -net
 none -chardev stdio,id=con,mux=on     -serial chardev:con -mon chard
ev=con,mode=readline -bios none   -s --kernel ./output/RTOSDemo.elf
Task 1 trying to lock Mutex A
Task 2 trying to lock Mutex B
Task 2 Task 1 locked Mutex A
locked Mutex B
Message received from task
Message received from task
Task 1 trying to lock Mutex B
Task 2 trying to lock Mutex A
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from software timer
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
Message received from task
QEMU: Terminated
```

After this point, no further output is produced.

- Blinky demo messages continue to appear due to independent tasks (queue/timer tasks), showing the system is still running even though the deadlock prevents progress in the two mutex tasks.

11.2.5 Deadlock Analysis

The system enters a deadlock state because all four Coffman conditions are satisfied:

1. **Mutual Exclusion**: Mutexes allow exclusive access
2. **Hold and Wait**: Each task holds one mutex while waiting for another
3. **No Preemption**: FreeRTOS does not forcibly release mutexes
4. **Circular Wait**: Tasks wait on resources held by each other

The scheduler continues running, but no task can make progress, resulting in a system-level stall.

11.2.6 Discussion

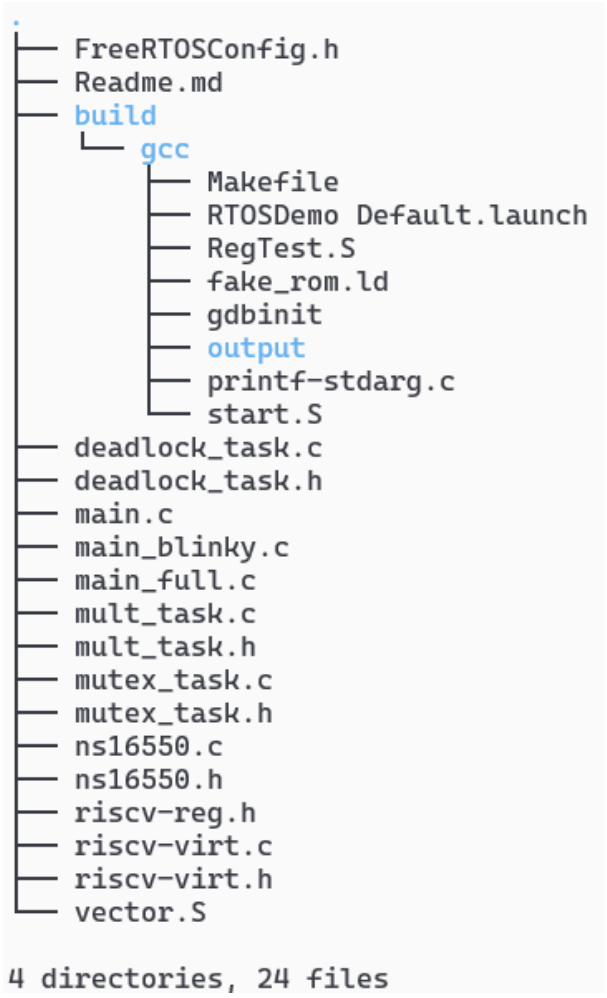
This demonstration confirms that:

- Deadlocks are application-level design errors
- FreeRTOS does not automatically detect or resolve deadlocks
- Proper locking discipline is essential in real-time systems

The deadlock scenario provides a concrete illustration of concurrency hazards and reinforces the importance of careful synchronization design in RTOS-based applications.

12 Project Directory Structure

The following directory tree shows the layout of the FreeRTOS RISC-V demo project:



This provides a high-level overview of the organization of source files, demos, and configuration files used in the project.

13 Limitations

Although the FreeRTOS demonstration and custom task integration were successfully implemented and executed, several limitations were identified. These limitations primarily arise from the use of an emulated execution environment and the scope of the demonstration itself, rather than from deficiencies in the FreeRTOS kernel or application design.

13.1 Timing-Sensitive Test Failures in QEMU

During execution of the Full Demo mode, the following error message may be observed:

```
islamabad@Namra:~/os_project/FreeRTOS/FreeRTOS/Demo/RISC-V_RV32_QEMU_VIRT_GCC/build
/gcc$ qemu-system-riscv32 -nographic -machine virt -net none -chardev stdio,id=con,
mux=on     -serial chardev:con -mon chardev=con,mode=readline -bios none   -s --ker
nel ./output/RTOSDemo.elf
Multiplication: 2 * 3 = 6
FreeRTOS Demo Start
Multiplication: 3 * 4 = 12
Multiplication: 4 * 5 = 20
Multiplication: 5 * 6 = 30
Multiplication: 6 * 7 = 42
Multiplication: 7 * 8 = 56
FreeRTOS Demo ERROR: xAreAbortDelayTestTasksStillRunning() returned false : 5038
Multiplication: 8 * 9 = 72
Multiplication: 9 * 10 = 90
Multiplication: 10 * 11 = 110
Multiplication: 11 * 12 = 132
QEMU: Terminated
```

This error is generated by the Abort Delay Test Task, which verifies the correctness of the FreeRTOS API function:

`xTaskAbortDelay()`

This test is highly sensitive to precise tick timing, interrupt latency, and scheduler behavior.

Root Cause

QEMU functions as a high-level functional emulator rather than a cycle-accurate hardware simulator. As a result:

- Tick interrupt timing does not perfectly reflect real hardware behavior.
- Interrupt latency and scheduling granularity may vary.
- Certain race-condition–based timing assumptions made by the test do not hold.

Consequently, timing-dependent validation tests may fail even when the RTOS port and configuration are correct.

Important Clarification

This limitation does not indicate a fault in:

- The FreeRTOS kernel implementation

- The RISC-V RV32 port
- The integrated multiplication task

Such behavior has been documented across multiple FreeRTOS ports when executed under simulation or emulation environments.

13.2 Emulation vs. Real Hardware

While QEMU provides a convenient and portable testing platform, it cannot fully replicate real embedded hardware behavior. The following constraints were observed:

- Absence of true peripheral timing characteristics.
- Simplified UART and timer models.
- Lack of hardware-induced interrupt jitter.
- No measurement of worst-case execution time (WCET).

As a result, strict real-time guarantees and hard deadline validation cannot be conclusively verified in this environment.

14 Conclusion

This project successfully demonstrated the setup, configuration, and extension of the FreeRTOS operating system on a RISC-V RV32 platform using QEMU. Both Full Demo and Blinky modes executed correctly, validating task creation, priority-based preemptive scheduling, and inter-task communication. The integration of a custom multiplication task confirmed FreeRTOS's modularity and predictable scheduling behavior, while the mutex and deadlock experiments illustrated key concurrency concepts and synchronization mechanisms. Memory footprint analysis highlighted the scalability differences between minimal and full-featured demo configurations. Although timing-sensitive tests exhibited minor limitations under emulation, these are inherent to QEMU and do not compromise the correctness of the FreeRTOS implementation, providing a strong foundation for future deployment on real RISC-V hardware.

15 References

- **Original FreeRTOS Repository:** FreeRTOS/FreeRTOS. (n.d.). *Classic FreeRTOS distribution*. https://github.com/FreeRTOS/FreeRTOS/tree/main
- **Project Repository:** Faiqa Zarar. (n.d.). *FreeRTOS RTOS Lab Project*. https://github.com/FaiqaZarar/FreeRTOS-RTOS-Lab-Project.git