GeMRTOS: Multiprocessor RTOS

User and Development Manual — Chapter 14: Windows Emulation Layer

gemrtos.com

August 2026

Revision: 0.9a-2026-06-01

CHAPTER 14

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.

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.

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?").

14.2.4 Processor Reset

Hardware: Can reset individual processors via controller.

Windows: No equivalent. Processor reset is not emulated.

Impact: Tests requiring processor reset must be validated on hardware.

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:

```
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:

```
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 (`GNU_GRTOS_controller.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:

```
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:

```
cd software/gemrtos_windows
gcc -g ../hellogemrtos/hellogemrtos.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_kernel.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_core.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_list.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_sem.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_mq.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_trigger.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_task.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_monitor.c \
    ../hellogemrtos_bsp/drivers/src/gemrtos_hal.c \
    GNU_GRTOS_controller.c grtos_fiber.c \
    -I ../hellogemrtos_bsp/ -I ../hellogemrtos_bsp/HAL/inc \
    -I ../hellogemrtos_bsp/drivers/inc -I ../hellogemrtos -I . \
    -I ../hellogemrtos_bsp/HAL/apps/http \
    -lgcov -o hellogemrtos.exe
```

This produces `software/gemrtos_windows/hellogemrtos.exe`.

14.4.5 Run Commands

To execute the Windows application:

```
cd software/gemrtos_windows
./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