QEMU Quasi-Hosting on TriCore
Emulate Infineon TriCore Microcontrollers with C Test Programs

Roy Ratcliffe[1,*]

Abstract

This document explores the use of QEMU's capabilities to emulate Infineon TriCore microcontrollers. It provides a comprehensive guide on setting up the environment, writing test programs that leverage quasi-hosting features, and debugging techniques to streamline embedded systems development.

Keywords: Infineon TriCore, C Programming, Embedded Systems

Infineon AURIX TriCore [1, 2]! It's a fascinating piece of silicon: DSP meets RISC in a high-speed, multi-core system. Ideal for automotive applications.

1. Introduction

Is it possible to run TriCore TC3xx tests without a development board using only an emulator in semi-hosting configuration, where the tests can write to the emulator's standard output for printing test results; also, where tests can interact with the host's file system by reading text fixtures or writing test result tabulations?

The short answer is *no*. Not yet. Not with the current version of QEMU, not completely. You *can* run TriCore code with QEMU, but it cannot straightforwardly exit the firmware or output results, which makes testing difficult—but not impossible. Tests need to signal success or failure at a minimum, and ideally output some results for comparison with test expectations.

This article explores the current state of TriCore emulation with QEMU and quasi-hosting; I'm calling it "quasi" for an exit code with basic output—not quite full semihosting but close enough to be useful as a testing platform. Exiting with a code allows for test assertions; an exit code of 0 indicates success, non-0 indicates failure. You can see the resulting exemplar project on GitHub. The example *only* works with a patched QEMU toolchain also available from GitHub.

2. Patching the Emulator

The solution is relatively simple. It involves patching the emulator. This is the basic idea: repurpose the "test device," giving it the ability to write to the emulator's standard output.

2.1. Test Device

QEMU's Tricore emulation equips a test board with a test device. It emulates a memory-mapped peripheral at F000 0000_{16} in address space. A memory read operation loads DEAD BEEF_{16}. A memory write operation, however, exits the emulator.

[*]Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1]See more hackery at GitHub.

The device only exits the emulator. It could do more. A write operation's data could select one of a number of different useful test-oriented tasks:

- exit the emulator with some given exit code;
- output an ASCII character to standard output;
- flush standard output; or
- any number of other things in future work.

2.2. Patching QEMU

The patch is straightforward. It modifies the test device's write handler to check the written value. If the value is less than 256, it is treated as an ASCII character and written to standard output. If the value is 256 or greater, it is treated as an exit code to terminate the emulator. The modified write handler looks like this.

The test device can easily be applied to the other TriCore emulations, not just the test board. It involves adding the test device to the emulated system's device tree; details follow.

2.3. TriCore Toolchain

The QEMU patches need a TriCore toolchain to build the test firmware. Fortunately, Domenico Iezzi [3] provides prebuilt GNU toolchains for TriCore targets. The toolchains are available for Linux and Windows hosts. The toolchains include the GNU compiler, linker, assembler, and other necessary tools for building TriCore applications.

These toolchains can be patched to include the necessary QEMU changes. My fork at GitHub of Domenico's repository includes the QEMU patches. Release 1.0.1 includes patches for the QEMU test board and TC277 emulations; access to the enhanced test device works for both.

Unzip the release package and add the binary folder to the build host's search path. Ensure the binary path appears *before* any other TriCore toolchain paths in the search path. This ensures that the patched tools are used when building the test firmware.

3. System Start-Up

When you build a project in AURIX Development Studio, you receive libraries and configuration sources along with linker scripts for the alternative toolchains (TASKING and GNU). These are part of the Infineon Low-Level Driver (iLLD) library. The iLLD provides drivers and low-level access to the microcontroller's peripherals and features. It is essential for developing applications that interact with the hardware at a low level. The start-up software configures the basic multi-core environment, and away we fly.

System start-up on TriCore has special requirements.

3.1. Startup Software

However, the emulator supports none of the peripherals and only one core.

This is not so much a problem. It means that the standard start-up will not work. We will need to provide our own minimal start-up code that initialises the system sufficiently for the emulated application to run. This includes setting up the LCX and FCX registers correctly and preparing the CSA.

3.1.1. Setting up the context save area (CSA)

LCX is the Link Context register, and FCX is the Free Context register. They are used to manage context switching in the TriCore architecture. LCX points to the previous context save area (CSA), while FCX points to the next available CSA. Properly configuring these registers is crucial for the correct operation of function calls and returns, as they rely on the context switching mechanism.

No CSA, no calling or returning. Calls and returns do not work without CSA set-up.

3.1.2. Initialise the CSA

The following function adapts the standard SSW function to initialise the Context Save Area (CSA) linked list. It is an always-inline function to avoid stack usage before the CSA is initialised.

```c
/*!
 * \brief Initialise the Context Save Area (CSA) linked list.
 * \details This function initialises the CSA linked list for the CPU on which
 * it is called. It links all CSAs in a linked list fashion and sets the FCX and
 * LCX registers accordingly.
 * \param csa_begin Pointer to the start of the CSA area.
 * \param csa_end Pointer to the end of the CSA area.
 * \note Each CSA consists of 16 words (64 bytes). The number of CSAs is
 * calculated based on the provided begin and end pointers.
 * \note The function ensures that all memory operations are completed before
 * returning by executing a DSYNC instruction.
 * \note This function \e must be an inline function to avoid stack usage before
 * the CSA is initialised.
 * \note Manifest constant \c IFX_SSW_INLINE is not just static and inline. It
 * declares an always_inline function. This is important to avoid function call
 * overhead and stack usage before CSA initialisation.
 */
IFX_SSW_INLINE void Ssw_initCSA(unsigned int *csa_begin, unsigned int *csa_end) {
  /*
   * Calculate the number of CSAs and the LCX index. Each CSA consists of
   * IFX_SSW_CSA_SIZE words (16 words to be exact). The number of CSAs can be
   * calculated as:
   *
   *    (((unsigned int)csa_end - (unsigned int)csa_begin) / (IFX_SSW_CSA_SIZE << 2U))
   */
  const unsigned int num_of_csa = (csa_end - csa_begin) / IFX_SSW_CSA_SIZE;
  const unsigned int lcx_idx = num_of_csa - 3U;
  unsigned int *prv_csa = csa_begin;
  unsigned int *nxt_csa = csa_begin;

  /*
   * Iterate over all CSAs and link them in a linked list.
   */
  for (unsigned int csa_idx = 0U; csa_idx < num_of_csa; csa_idx++) {
    unsigned int cxi_adr = (unsigned int)nxt_csa;
    unsigned int cxi_val =
        ((cxi_adr & ((unsigned int)0xfU << 28U)) >> 12U) |
        ((cxi_adr & ((unsigned int)0xffffU << 6U)) >> 6U);

    /*
     * Link the previous CSA to the current one. For the first CSA, set FCX
     * register.
     */
    if (csa_idx == 0U) {
      Ifx_Ssw_MTCR(CPU_FCX, cxi_val);
    } else {
      *prv_csa = cxi_val;
    }
    /*
     * For the last CSA, link it to LCX register.
     */
    if (csa_idx == lcx_idx) {
      Ifx_Ssw_MTCR(CPU_LCX, cxi_val);
    }

    /*
     * Move to the next CSA.
     */
    prv_csa = nxt_csa;
    nxt_csa += IFX_SSW_CSA_SIZE;
  }

  /*
   * Terminate the linked list. Apply a data memory barrier to ensure all memory
   * operations are completed.
   */
  *prv_csa = 0U;
  Ifx_Ssw_DSYNC();
}
```

This implementation uses unsigned integer pointer arithmetic for clarity and correctness, and reuses the `IFX_SSW_CSA_SIZE` manifest constant that defines the size of a single CSA in words (not bytes).

3.1.3. Essential optimisation

The start-up software (SSW) for TriCore requires at least `-O2` optimisation to function correctly. Without this, the function calls may fail or behave unexpectedly. The following pragma can be used to enforce this optimisation level in specific source files. Do not fiddle them away.

```
#pragma GCC optimize "O2"
```

4. Non-Standard Library

With a pseudo-test device controlling the emulator's exit and access to the emulator's standard output, the standard library needs adjusting. The normal toolchain comes with Red Hat's Newlib [4] C library.

4.1. Exit and Abort

Standard library assertions invoke the C standard library's `abort` function which runs the `at_exit` functions then sits there in an infinite loop. This is no good for testing. Abort needs to terminate the emulator.

```
#include <stdlib.h>

#include "rwonce.h"
#include "tricore_testdevice.h"

void exit(int status) {
  WRITE_ONCE(*TRICORE_TESTDEVICE, status);
  while (1) {
    ;
  }
```

```c
}

void abort(void) { exit(8); }
```

The first function exits the program by writing the status code to the TriCore test device. This function does not return. `WRITE_ONCE` is a macro to prevent the compiler from optimising away the write operation by casting a given *l*-value[2] to a volatile reference, assigning it to a given *r*-value to effect a mandatory memory write operation.

The second function aborts the program by calling exit with a specific status code. Why exit code 8? It's arbitrary but distinct. Exit codes do have meanings in some environments. See Exit Codes With Special Meanings, for instance. This function does not return either.

5. Adding Quasihosting Tests

Writing a little CMake [6] function can help with this.

```cmake
# CMake function to add TriCore quasihosting tests.
# Parameters:
# TEST_NAME - Name of the test executable.
# TEST_SOURCES - List of source files for the test.
# APP_SOURCES - List of application source files to include in the test.
# SYSTEM_SOURCES - List of system source files, e.g. startup code.
# LINK_LIBRARIES - List of libraries to link against.
# LINK_OPTIONS - List of link options to apply.
# TIMEOUT - Optional timeout for the test.
# Usage:
# add_tricore_quasihosting_test(TEST_NAME my_test
#     TEST_SOURCES
#         test1.c
#         test2.c
#     APP_SOURCES
#         app1.c
#         app2.c
#     SYSTEM_SOURCES
#         system1.c
#         system2.c
#     LINK_LIBRARIES
#         library1
#         library2
# )
function(add_tricore_quasihosting_test)
    set(options)
    set(oneValueArgs TEST_NAME TIMEOUT)
    set(multiValueArgs TEST_SOURCES APP_SOURCES SYSTEM_SOURCES LINK_LIBRARIES LINK_OPTIONS)
    cmake_parse_arguments(ATQT "${options}" "${oneValueArgs}" "${multiValueArgs}" ${ARGN})

    add_executable(${ATQT_TEST_NAME})
    target_sources(${ATQT_TEST_NAME} PRIVATE
        ${TriCoreQuasihostingTestSources}
        ${ATQT_TEST_SOURCES}
```

[2] An *l-value* is an expression with an object type other than void that potentially designates an object [5].

```
        ${TriCoreQuasihostingAppSources}
        ${ATQT_APP_SOURCES}
        ${TriCoreQuasihostingSystemSources}
        ${ATQT_SYSTEM_SOURCES}
    )
    target_include_directories(${ATQT_TEST_NAME} PRIVATE ${TriCoreQuasihostingIncludeDirectories})
    target_compile_definitions(${ATQT_TEST_NAME} PRIVATE ${TriCoreQuasihostingCompileDefinitions})

    # Link against any additional libraries, e.g. TriCore math.
    target_link_libraries(${ATQT_TEST_NAME} PRIVATE
        ${TriCoreQuasihostingLinkLibraries}
        ${ATQT_LINK_LIBRARIES}
    )

    # Link with quasihosting specifications. The test device provides quasihosting
    # support for TriCore targets.
    target_link_options(${ATQT_TEST_NAME} PRIVATE
        ${TriCoreQuasihostingLinkOptions}
        ${ATQT_LINK_OPTIONS}
        -Wl,-Map,$<TARGET_FILE_BASE_NAME:${ATQT_TEST_NAME}>.map
    )

    # Set timeout for the test if specified.
    if(ATQT_TIMEOUT)
        set_target_properties(${ATQT_TEST_NAME} PROPERTIES TIMEOUT ${ATQT_TIMEOUT})
    endif()

    add_test(NAME ${ATQT_TEST_NAME} COMMAND
        ${CMAKE_CROSSCOMPILING_EMULATOR} $<TARGET_FILE:${ATQT_TEST_NAME}>)
endfunction()
```

Adding a CTest [7] then becomes simple.

```
add_tricore_quasihosting_test(TEST_NAME assert_true
    TEST_SOURCES
        ${CMAKE_SOURCE_DIR}/Tests/assert_true.c
)
```

6. Debugging with VS Code

Debugging tests with VS Code would be useful. The following `tasks.json` file launches QEMU with the necessary options to support GDB debugging.

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "Launch QEMU",
            "type": "shell",
            "command": "qemu-system-tricore",
            "args": [
                // Use the TriCore TC277 machine model. It supports a pseudo-test device
                // at address 0xf0000000 for simple quasi-hosting operations.
                "-M", "KIT_AURIX_TC277_TRB",

                // Use the console for serial I/O. QEMU's default is to create a
                // separate window for serial I/O. The QEMU monitor will appear
                // in the launch terminal.
                "-nographic",

                // Create some extra reset output for the problem matcher to catch.
                // This helps to determine when QEMU has started successfully.
                // Add in_asm to get more detailed output.
                "-d", "cpu_reset,guest_errors",

                "-kernel", "${command:cmake.launchTargetPath}",
                "-s",
                "-S"
            ],
            "dependsOn": [
                "CMake: build",
            ],
            "isBackground": true,
            "problemMatcher": {
                "owner": "qemu",
                "pattern": [
                    {
                        "regexp": "^(.*)$",
                        "file": 1,
                        "location": 2,
                        "message": 3
                    }
                ],
                "background": {
                    "activeOnStart": true,
                    "beginsPattern": "qemu-system-tricore",
                    "endsPattern": "CPU Reset"
                }
            },
            "group": "test",
            "detail": "Launch QEMU emulator",
            "presentation": {
                "reveal": "always",
                "panel": "dedicated"
            }
        },
        {
            "label": "Terminate QEMU",
            "command": "${input:terminate}",
            "type": "shell",
            "problemMatcher": []
        }
    ],
    "inputs": [
        {
            "id": "terminate",
            "type": "command",
            "command": "workbench.action.tasks.terminate",
            "args": "terminateAll"
        }
    ]
}
```

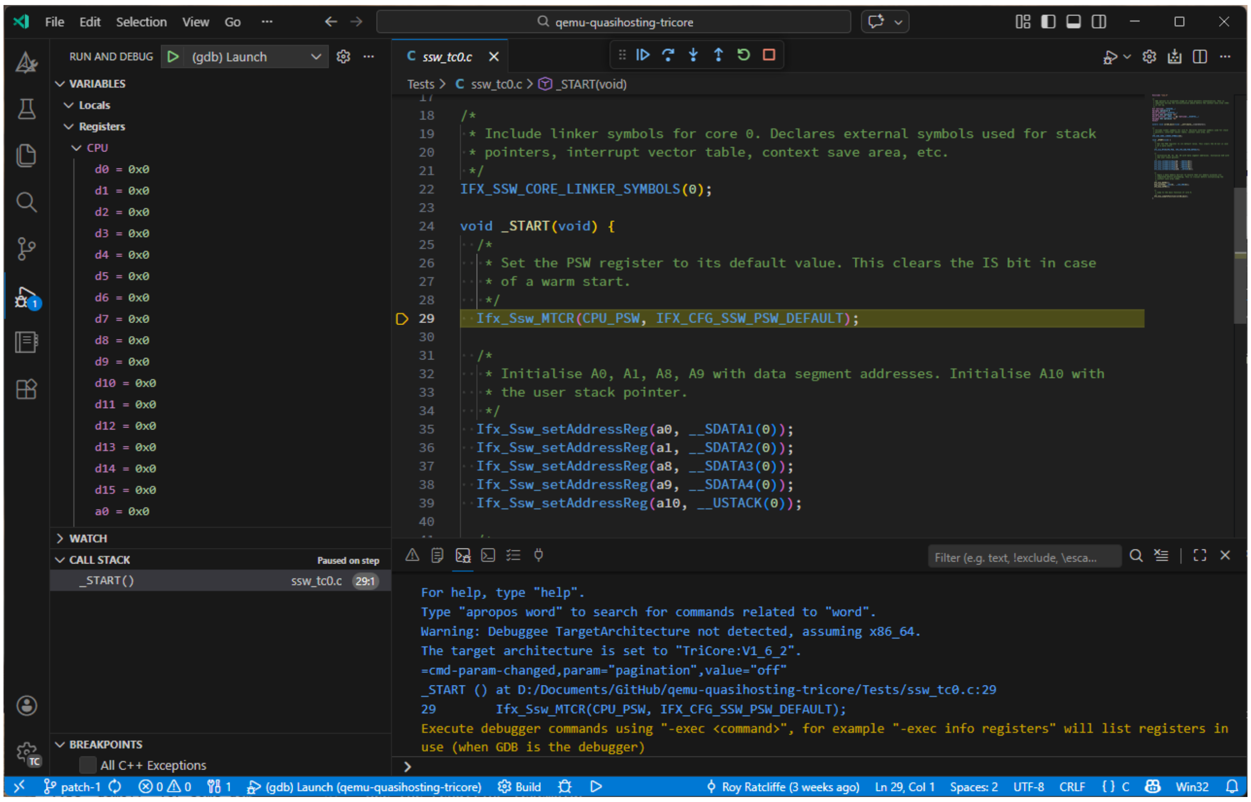
Requires QEMU to be installed and available in `PATH`. You may need to adjust the command to point to `qemu-system-tricore` or `qemu-system-tricore.exe` after installation.

Note that the GDB server options are required to match those specified in launch.json for the debugger to connect. Option `-s` is shorthand for `-gdb tcp::1234` which starts a GDB server listening on localhost port 1234. Option `-S` makes QEMU start in paused mode, waiting for a GDB connection.

The launch configuration in `launch.json` only needs to specify the GDB connection details and the pre- and post-launch tasks.

```
            "preLaunchTask": "Launch QEMU",
            "postDebugTask": "Terminate QEMU",
```

Then just press F5 to start debugging. See Figure 1 for a screenshot of the setup in action.


File Edit Selection View Go ···
qemu-quasihosting-tricore
RUN AND DEBUG (gdb) Launch
ssw_tc0.c
Tests > ssw_tc0.c > _START(void)
VARIABLES
Locals
Registers
CPU
WATCH
CALL STACK Paused on step
_START() ssw_tc0.c 29:1
BREAKPOINTS
All C++ Exceptions
Filter (e.g. text, !exclude, \esca...
patch-1 ⊗ 0 ⚠ 0 1 (gdb) Launch (qemu-quasihosting-tricore) Build
Roy Ratcliffe (3 weeks ago) Ln 29, Col 1 Spaces: 2 UTF-8 CRLF {} C Win32

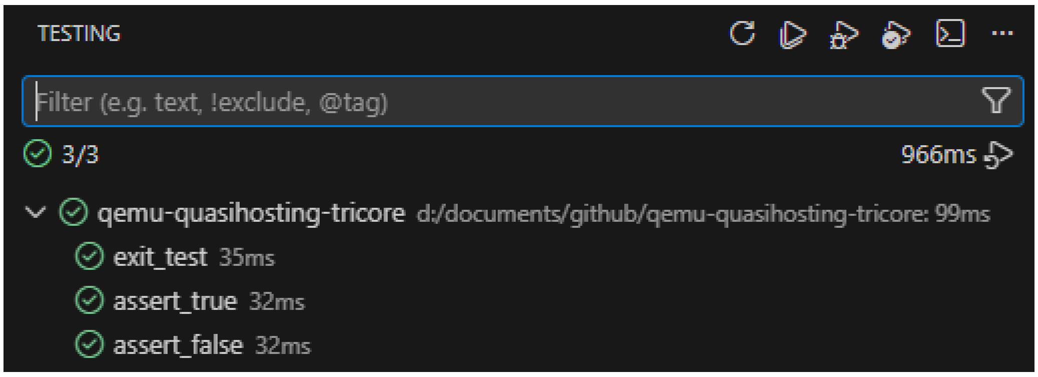
```
d0 = 0x0
d1 = 0x0
d2 = 0x0
d3 = 0x0
d4 = 0x0
d5 = 0x0
d6 = 0x0
d7 = 0x0
d8 = 0x0
d9 = 0x0
d10 = 0x0
d11 = 0x0
d12 = 0x0
d13 = 0x0
d14 = 0x0
d15 = 0x0
a0 = 0x0
```

```
/*
 * Include linker symbols for core 0. Declares external symbols used for stack
 * pointers, interrupt vector table, context save area, etc.
 */
IFX_SSW_CORE_LINKER_SYMBOLS(0);

void _START(void) {
  /*
   * Set the PSW register to its default value. This clears the IS bit in case
   * of a warm start.
   */
  Ifx_Ssw_MTCR(CPU_PSW, IFX_CFG_SSW_PSW_DEFAULT);

  /*
   * Initialise A0, A1, A8, A9 with data segment addresses. Initialise A10 with
   * the user stack pointer.
   */
  Ifx_Ssw_setAddressReg(a0, __SDATA1(0));
  Ifx_Ssw_setAddressReg(a1, __SDATA2(0));
  Ifx_Ssw_setAddressReg(a8, __SDATA3(0));
  Ifx_Ssw_setAddressReg(a9, __SDATA4(0));
  Ifx_Ssw_setAddressReg(a10, __USTACK(0));
```

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
Warning: Debuggee TargetArchitecture not detected, assuming x86_64.
The target architecture is set to "TriCore:V1_6_2".
=cmd-param-changed,param="pagination",value="off"
_START () at D:/Documents/GitHub/qemu-quasihosting-tricore/Tests/ssw_tc0.c:29
29        Ifx_Ssw_MTCR(CPU_PSW, IFX_CFG_SSW_PSW_DEFAULT);
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in
use (when GDB is the debugger)
```

Figure 1: Debugging with VS Code is then a matter of setting breakpoints in the test source files and launching the debugger.

Press “;” then “A” to run all tests. See Figure 2 for a screenshot of the test results in the terminal window.


TESTING

Filter (e.g. text, !exclude, @tag)

3/3 966ms

qemu-quasihosting-tricore d:/documents/github/qemu-quasihosting-tricore: 99ms
exit_test 35ms
assert_true 32ms
assert_false 32ms

Figure 2: Test results appear in the terminal window.

7. Conclusions

Why bother? This approach allows for emulated test programs that can apply exhaustive testing to a firmware's application code.

This approach is not a substitute for testing on real hardware. Some aspects of embedded systems cannot be fully emulated. However, it provides a valuable tool for early-stage testing and development. It does have some limitations, but it also adds some useful capabilities. Application-level testing can be automated without needing physical hardware. Such tests can run quickly and repeatedly, catching regressions early in the development cycle. Tests can also apply exhaustive testing, which may be impractical on real hardware. Safety-critical systems benefit from such thorough testing.

Typically, an embedded firmware's application code is tightly coupled to the hardware. This makes it difficult to test the application code in isolation. By using an emulator with quasi-hosting capabilities, the application code can be tested without needing the actual hardware. The hardware and low-level drivers can be mocked or stubbed out, allowing the application code to be tested in a controlled and repeatable environment.

7.1. Test Board or TC277

The emulator has a very basic "testboard" emulation. It includes the test device by default. Better to use the "KIT_AURIX_TC277_TRB" emulation. The memory map more closely matches real devices. In fact, the test firmware can completely reuse the primary firmware's loader script. Shared loader script simplifies the build system.

References

[1] Infineon Technologies, TriCore^{TM} TC1.6.2 Core Architecture Manual: Volume 1, User Manual Volume 1 (2020).
URL https://www.infineon.com/products/microcontroller/32-bit-tricore

[2] Infineon Technologies, TriCore^{TM} TC1.6.2 Core Architecture Manual: Volume 2, User Manual Volume 2 (2020).
URL https://www.infineon.com/products/microcontroller/32-bit-tricore

[3] D. Iezzi, TriCore GCC Toolchain, GCC-based toolchain for Infineon TriCore microcontrollers.
URL https://nomore201.github.io/tricore-gcc-toolchain/

[4] Red Hat, Inc., Newlib C Library (2024).
URL https://sourceware.org/newlib/

[5] ISO/IEC JTC 1/SC 22/WG 14, Lvalues, arrays, and function designators, ISO, 2018, Ch. 6.3.2.1.

[6] Kitware, Inc., CMake – Open Source Build System (2024).
URL https://cmake.org

[7] Kitware, Inc., CTest – Automated Testing Tool (2024).
URL https://cmake.org/cmake/help/latest/manual/ctest.1.html

[8] The QEMU Project, QEMU (2024).
URL https://www.qemu.org/

[9] QEMU TriCore Platform Documentation (2026).
URL https://wiki.qemu.org/Documentation/Platforms/TriCore