QEMU Semi-Hosting on STM32

Emulate ARM Cortex-M Microcontrollers with C Test Programs

Roy Ratcliffe[1,*]

Abstract

This document explores the use of QEMU's semi-hosting capabilities to emulate ARM Cortex-M microcontrollers, specifically the STM32F4 series. It provides a comprehensive guide on setting up the environment, writing test programs that leverage semi-hosting features, and debugging techniques to streamline embedded systems development.

Keywords: ARM Cortex-M, C Programming, Embedded Systems

1. Introduction

How easy is it to set up a CMake [1] STM32F4 project to include QEMU [2] semi-hosted tests using CTest [3]?

You can find the final result as a working example on GitHub.

2. The Problem

Semi-hosting allows the emulated target to access the host's standard input-output, as well as its file system. This seems a little strange, to say the least. An embedded system does not have a "host" with which it can communicate. It lives in its own little world and communicates with external systems using its bus via integrated peripherals.

For testing the *application* part of the firmware, however, a semi-hosted approach could allow emulated testing on those parts of an application that do not interact with the peripherals: the logical and behavioural parts of the application, call it the application proper. These tests could also run automatically using CTest on some cloud-based testing service, e.g. as a GitHub action.

Here are some considerations.

2.1. Use an Up-to-Date QEMU Version

This is an important starting point. If you install QEMU using Chocolatey on Windows, your development machine will have version 10.1.93 at the time of writing, which reports using:

```
PS C:\Users\me> & 'C:\Program Files\qemu\qemu-system-arm.exe' --version
QEMU emulator version 10.1.93 (v10.2.0-rc3-12102-gd47b0105be)
Copyright (c) 2003-2025 Fabrice Bellard and the QEMU Project developers
```

Linux distributions may provide different versions. Make sure that the version of QEMU matches or exceeds 10.*x* because some versions of Linux, e.g. Bookworm editions, come with 7.*x* series via the standard APT packaging system, where support for the STM32F4 does not include Core-Coupled Memory (CCM) RAM. Install the `qemu` package using Homebrew; it equips the later, more up-to-date versions. Ensure that APT does not install `qemu-system-arm` or remove it[2].

*Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)

[1] See more hackery at GitHub.

[2] Remove the ARM system emulator using `sudo apt remove qemu-system-arm` on the Linux command line.

2.2. *Main Function Has No Arguments*

Since the firmware operates an embedded system, the `main` function has the C prototype: `int main(void)`. In other words, the main function does **not** accept `int argc` nor `char **argv`— the argument counter and vector. Strictly speaking, the application entry-point does not return either; it should perhaps have a `__attribute__((noreturn))` applied to it. The firmware proper's `main` function sets up the hardware abstraction layer (HAL) and starts the operating system kernel, assuming it equips a kernel such as FreeRTOS [4] and never returns because there is nowhere to return to.

This may seem like an obvious point, and it is. However, it has ramifications for testing, particularly when testing with CTest. CTest's testing normally folds multiple tests into the same executable test runner. Multiple tests live in the same binary. Command-line arguments normally select which test to run. The standard test runner asks for standard input for a test selection if none appears on the command line. So, for no-argument `main` functions, CTest cannot pass a test selector. Consequently, every test must live in its own test executable: one test, one binary.

3. The Solution

Start by creating a CTest test that runs QEMU with semi-hosting enabled.

3.1. *CMake Snippet for a Single Test*

The following CMake snippet adds a CTest test called `a_test` that runs a semi-hosting-enabled test binary using QEMU. Notice that it adds include directory paths for the core application as well as the HAL driver and CMSIS headers. It also defines the `STM32F407xx` manifest constant and other necessary defines for the test executable. The test's source should include the test's `main` function, application-specific source, excluding hardware dependencies, and system and start-up sources for the STM32F4.

```
include(CTest)
enable_testing()

# Unit tests setup. Define include paths and symbols for unit tests. These are
# similar to the main project, but can differ as the tests exclude
# hardware-specific sources and middlewares.
set(MX_Test_Include_Dirs
    ${CMAKE_SOURCE_DIR}/Core/Inc
    ${CMAKE_SOURCE_DIR}/Drivers/STM32F4xx_HAL_Driver/Inc
    ${CMAKE_SOURCE_DIR}/Drivers/STM32F4xx_HAL_Driver/Inc/Legacy
    ${CMAKE_SOURCE_DIR}/Drivers/CMSIS/Device/ST/STM32F4xx/Include
    ${CMAKE_SOURCE_DIR}/Drivers/CMSIS/Include
)
set(MX_Test_Defines_Syms
    USE_HAL_DRIVER
    STM32F407xx
    $<$<CONFIG:Debug>:DEBUG>
)

# This test is run using QEMU emulating a Netduino Plus 2 board (STM32F407VG).
# The test uses semihosting to report results back to the host.
# The test has a timeout of 30 seconds.
add_executable(a_test)
target_sources(a_test PRIVATE
    # Add test sources.
    ${CMAKE_SOURCE_DIR}/Tests/a_test.c
    # Add application sources.
```

```
    # Add system and startup sources.
    ${CMAKE_SOURCE_DIR}/Core/Src/system_stm32f4xx.c
    ${CMAKE_SOURCE_DIR}/startup_stm32f407xx.s
)
target_include_directories(a_test PRIVATE ${MX_Test_Include_Dirs})
target_compile_definitions(a_test PRIVATE ${MX_Test_Defines_Syms})
target_link_options(a_test PRIVATE --specs=rdimon.specs -lrdimon)
set_target_properties(a_test PROPERTIES TIMEOUT 30)

# Add the CTest test that runs the QEMU emulation with semihosting enabled.
add_test(NAME a_test COMMAND qemu-system-arm
    -machine netduinoplus2
    -nographic
    -no-reboot
    -semihosting-config enable=on,target=native
    -kernel ${CMAKE_BINARY_DIR}/a_test.elf)
```

Simply add a new executable to add a new test, as many as necessary. The test does not accept command-line arguments. A new test, therefore, requires a new binary.

3.2. Remote Dispatch Interface (RDI) Monitor

Notice the target link options. They carry some essential overrides.

```
target_link_options(a_test PRIVATE --specs=rdimon.specs -lrdimon)
```

`rdimon` is a specification used in the ARM GNU Toolchain [5] that enables semi-hosting for debugging on bare-metal targets. It allows code running on an ARM target to communicate with a host computer. The emulated firmware can use system calls.

The link options do two things:

1. uses the RDI monitor specification;
2. links against the RDI monitor library.

The test target's link options merge with the original toolchain link options. Typically, the toolchain includes link options with `--specs=nano.specs` for newlib [6] nano. The test's new specification overrides the original nano because it comes later in the linker's command line, and `-lrdimon` links with the RDI monitor library. This makes the test ready for running with QEMU equipped to call `printf` for printing to standard output, and `fopen` for reading and writing to the host filesystem.

It is worth noting that not all QEMU system emulator types support semi-hosting. The AURIX TriCore emulator, for example, does not; the TC27x emulator does not accept the `-semihosting-config` option, nor does the compiler toolchain for TriCore equip the RDI monitor specification and library. The ARM variant does, however, and also includes the RDI monitor. So this approach works for STM32xx devices.

3.3. Monitor handles

There is more. There are test run-time considerations. The test's `main` function must call an initialisation function to set up the monitor.

The `main` function needs to call the "initialisation monitor handles" function supplied by the RDI monitor library. File handles, including standard input and output, fail to operate until this function runs.

```c
/* SPDX-License-Identifier: MIT */
/*!
 * \file monitor_handles.h
 * \brief Header file for monitor handles initialisation.
 * \details This file declares the function to initialise monitor handles
 * for semihosting support in the testing environment.
 */

#pragma once

/*!
 * \brief Initialise the monitor handles for semihosting support.
 * \details File handles will not be available until this function is called.
 */
void initialise_monitor_handles(void);
```

The basic `main` implementation then becomes:

```c
#include "monitor_handles.h"

#include <stdio.h>
#include <unistd.h>

int main(void) {
  initialise_monitor_handles();
  (void)printf("Hello, World from a_test!\n");
  _exit(0);
  return 0;
}
```

3.4. Floating-Point Conversion

There is an additional run-time limitation when using floating-point numbers. The nano-standard library does not include `%f` formatting in `printf`. Its build configuration excludes that, providing only integer formatting. This only matters if your tests want to print out floats.

At the time of writing, the STM32 GNU tools provide newlib version 4.4.0. Clone its Git repository using:

```
git clone https://sourceware.org/git/newlib-cygwin.git
```

The 4.4.0 version has a `newlib-4.4.0` tag that snapshots the version in use. Its worth a dig to discover what goes on. The library does include float formatting capabilities in the source, but its compile options make them optional; obviously, this helps to make a bloat-free "nano" version of a C standard library suitable for small memory footprint microprocessors such as an STM32. ARM have therefore built their nano-Newlib with minimal floating-point support, understandably; many of their 32-bit processors do not even have a floating-point co-processor.

To enable floating-point formatting in `printf`, one would need to recompile newlib with the `--enable-newlib-io-float` option given to `configure`. This makes float formatting available in `printf` and related functions. That would be possible, for sure, but long-winded. Better to keep it simple.

3.4.1. Converting a float to a string

Do it the pragmatic way based on the `fcvtf()` legacy function, which the bare-bones nano library does include.

```c
#include <malloc.h>
#include <stdio.h>
```

```c
#include <string.h>

/*
 * The newlib nano C library provides this legacy function.
 * Converts a float to a string.
 */
extern char *fcvtf(float d, int ndigit, int *decpt, int *sign);

/*!
 * \brief Convert a float to string using \c fcvtf() into a user-provided buffer.
 * \details Converts the float number \p f with \p ndigit digits after the
 * decimal point to a string using the legacy function fcvtf(). The result is
 * stored in the user-provided buffer \p buf. If \p buf is \c NULL, a new buffer
 * is allocated using the standard library's \c realloc() function.
 * \param f Float number to convert.
 * \param ndigit Number of digits to convert after the decimal point.
 * \param buf User-provided buffer to store the result. If \c NULL, a new buffer
 * is allocated.
 * \return Pointer to the converted string in the provided or newly-allocated
 * buffer. Returns \c NULL if memory allocation fails.
 * \note The caller is responsible for freeing the returned buffer if it was
 * allocated by the function (i.e., when \p buf is \c NULL).
 */
char *cvtfbuf(float f, int ndigit, char *buf) {
  /*
   * Allocate buffer space from the heap if not provided by the user. As a
   * reasonable estimate, ask for ndigit + 36 characters to accommodate the
   * digits, decimal point, sign, and possible leading "0." and extra zeros.
   * Remember to free the buffer after use if it was allocated here.
   */
  if (buf == NULL) {
    if ((buf = (char *)realloc(buf, ndigit + 36)) == NULL)
      return NULL;
  }
  int decpt, sign;
  const char *cvt = fcvtf(f, ndigit, &decpt, &sign);
  char *ptr = buf;
  if (sign) {
    *ptr++ = '-';
  }
  /*
   * Construct the final part of the string. If the decimal point is before the
   * first digit, insert a leading "0." and the appropriate number of zeros.
   * Otherwise, copy the digits before the decimal point,
   */
  if (decpt <= 0) {
    *ptr++ = '0';
    *ptr++ = '.';
    for (int i = 0; i < -decpt; i++) {
      *ptr++ = '0';
    }
```

```
    (void)strcpy(ptr, cvt);
  } else {
    (void)strncpy(ptr, cvt, decpt);
    ptr += decpt;
    *ptr++ = '.';
    /*
     * Copy the remaining digits after the decimal point.
     * Assumes that fcvtf() null-terminates the string.
     */
    (void)strcpy(ptr, cvt + decpt);
  }
  return buf;
}
```

This workaround allows for float printing using a temporary buffer and the %s format specifier.

3.5. *Cutting a Long Story Short*

Putting it all together—the solution simply requires a CMake function that accepts

- the name of a test target that builds an ELF file compiled with RDI,
- a list of sources to test and
- a selection of application sources that do *not* depend on hardware.

3.5.1. *CMake function for adding a test*

See such a CMake function below.

```
# CMake function to add ARM semihosting 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.
# Usage:
# add_arm_semihosting_test(TEST_NAME my_test
#     TEST_SOURCES
#         test1.c
#         test2.c
#     APP_SOURCES
#         app1.c
#         app2.c
# )
function(add_arm_semihosting_test)
    set(options)
    set(oneValueArgs TEST_NAME)
    set(multiValueArgs TEST_SOURCES APP_SOURCES)
    cmake_parse_arguments(AAST "${options}" "${oneValueArgs}" "${multiValueArgs}" ${ARGN})

    add_executable(${AAST_TEST_NAME})
    target_sources(${AAST_TEST_NAME} PRIVATE
        ${AAST_TEST_SOURCES}
        ${AAST_APP_SOURCES}
        ${CMAKE_SOURCE_DIR}/Core/Src/system_stm32f4xx.c
```

```
        ${CMAKE_SOURCE_DIR}/startup_stm32f407xx.s
    )
    target_include_directories(${AAST_TEST_NAME} PRIVATE ${ARMSemihostingIncludeDirectories})
    target_compile_definitions(${AAST_TEST_NAME} PRIVATE ${ARMSemihostingCompileDefinitions})

    # Link against the ARM Cortex-M4 math library.
    target_link_libraries(${AAST_TEST_NAME} PRIVATE arm_cortexM4lf_math)

    target_link_options(${AAST_TEST_NAME} PRIVATE --specs=rdimon.specs -lrdimon)
    set_target_properties(${AAST_TEST_NAME} PROPERTIES TIMEOUT 30)
    add_test(NAME ${AAST_TEST_NAME} COMMAND ${CMAKE_CROSSCOMPILING_EMULATOR} ${CMAKE_BINARY_DIR}/${AAST_TEST_NAME}
endfunction()
```

This makes it easy to add new tests. Simply call the function with the test name, the test sources, and the application sources to include in the test; CMake will do the rest: build an ELF and launch QEMU to run it, capturing success or failure and any standard output.

3.5.2. Configure the toolchain's cross-compiling emulator

It relies on a refactored cross-compiling emulator configuration added to the toolchain file:

```
# QEMU System ARM emulator configuration for cross-compiling tests.
set(CMAKE_CROSSCOMPILING_EMULATOR qemu-system-arm
    -machine netduinoplus2
    -nographic
    -no-reboot
    -semihosting-config enable=on,target=native
    -kernel
)
```

3.5.3. Define Include Paths and Compile Definitions

It relies on two variables defined elsewhere in the CMakeLists.txt file:

```
set(ARMSemihostingCompileDefinitions
    USE_HAL_DRIVER
    STM32F407xx
    $<$<CONFIG:Debug>:DEBUG>
)
set(ARMSemihostingIncludeDirectories
    ${CMAKE_SOURCE_DIR}/Core/Inc
    ${CMAKE_SOURCE_DIR}/Drivers/STM32F4xx_HAL_Driver/Inc
    ${CMAKE_SOURCE_DIR}/Drivers/STM32F4xx_HAL_Driver/Inc/Legacy
    ${CMAKE_SOURCE_DIR}/Drivers/CMSIS/Device/ST/STM32F4xx/Include
    ${CMAKE_SOURCE_DIR}/Drivers/CMSIS/Include
)
```

These include paths and compile definitions are identical to those used for the main firmware project, but they can differ if necessary as the tests exclude hardware-specific sources and middleware.

4. Worked Example

Take a typical embedded example. Generate and build a piece of firmware for an STM32F4 using an ST Discovery evaluation board and the STM32CubeMX tool version 6.16.1.

Suppose that your embedded firmware project carries a “rolling ring-buffer signal correlation feature” and you want to build a suite of tests to ensure its correct functionality. The firmware runs on an STM32F4xx and equips ARM’s DSP library [7].

Find the complete working example on GitHub.

4.0.1. Rolling correlation using the ARM DSP library

The essential correlation function implementation is as follows. It reuses the correlation math functions from the DSP library to perform correlation on data extracted from a pair of rolling ring buffers.

```
/*!
 * \brief Perform correlation on the data in the correlate_f32 instance.
 * \param correlate Correlate 32-bit float instance.
 * \returns 0 on success, negative error code on failure.
 * \note Updates the correlated_len, expected_len, and actual_len fields.
 * \note Operates on all data currently in the expected and actual ring buffers.
 */
int correlate_f32(struct correlate_f32 *correlate) {
  /*
   * Get used data from the expected and actual ring buffers into the correlate
   * instance's expected and actual data arrays.
   *
   * This is necessary because arm_correlate_f32() operates on contiguous
   * arrays, whereas the ring buffers overate in discontinuous memory space and
   * may have wrapped around the end of the buffer. At most there will be two
   * memory copies per buffer. That makes two, three or four memory copy
   * operations in total depending on whether each buffer is contiguous or not.
   */
  const size_t expected_len = ring_buf_get_used_f32(correlate->buf_expected, correlate->expected);
  const size_t actual_len = ring_buf_get_used_f32(correlate->buf_actual, correlate->actual);
  correlate->expected_len = expected_len;
  correlate->actual_len = actual_len;
  if (actual_len == 0U || expected_len == 0U) {
    correlate->correlated_len = 0U;
    return -EINVAL;
  }
  arm_correlate_f32(correlate->expected, expected_len, correlate->actual, actual_len,
                    correlate->correlated);
  correlate->correlated_len = expected_len + actual_len - 1U;
  return 0;
}
```

See the ARM-optimised correlate function on GitHub.

4.1. Test Source

Here is a fairly straightforward test source file that exercises the correlation function using known input data and expected output data.

```
#include "arm_math.h"
#include "correlate_f32.h"
#include "fcvtf.h"
#include "fepsiloneq.h"
#include "monitor_handles.h"
```

```
#include <assert.h>
#include <stdio.h>
#include <unistd.h>

CORRELATE_F32_DEFINE_STATIC(test_corr, 100);

static const float32_t x[] = {0.0f, 1.0f, 2.0f, 3.0f, 2.0f, 1.0f};
static const float32_t h[] = {0.5f, 0.25f, -0.25f};

int correlate_f32_test(void) {
  /*
   * Buffer for float-to-string conversion. Used by cvtfbuf() to avoid repeated
   * allocations.
   */
  char buf[80];

  /*
   * Load the input signals and run a correlation.
   * Fail the test if the correlation fails.
   */
  for (size_t i = 0; i < sizeof(x) / sizeof(x[0]); i++) {
    assert(correlate_add_expected_f32(&test_corr, x[i]) == 0);
  }
  for (size_t i = 0; i < sizeof(h) / sizeof(h[0]); i++) {
    assert(correlate_add_actual_f32(&test_corr, h[i]) == 0);
  }
  assert(correlate_f32(&test_corr) == 0);

  float_t *correlated;
  size_t correlated_len = correlate_get_correlated_f32(&test_corr, &correlated);
  assert(correlated_len == sizeof(x) / sizeof(x[0]) + sizeof(h) / sizeof(h[0]) - 1);

  /*
   * Print the correlation result.
   */
  printf("Correlation before normalisation:\n");
  for (size_t i = 0; i < correlated_len; i++) {
    /*
     * Note that nano newlib does not support %f format specifier. Nor does it
     * support %zu for size_t; it crashes!
     */
    (void)printf("  correlated[%3d] = %15s\n", (int)i, cvtfbuf(correlated[i], 9, buf));
  }

  /*
   * Get and print the peak correlation and its lag.
   */
  float_t peak;
  int32_t peak_lag = correlate_peak_lag_f32(&test_corr, &peak);
  assert(peak_lag != INT32_MIN);
  (void)printf("Peak correlation value %s at lag %ld\n", cvtfbuf(peak, 9, buf), (long)peak_lag);
```

```
  /*
   * Normalise the correlation result. Fail the test if normalisation fails.
   */
  assert(correlate_normalise_f32(&test_corr) == 0);
  printf("Correlation after normalisation:\n");
  for (size_t i = 0; i < correlated_len; i++) {
    printf("  correlated[%3d] = %15s\n", (int)i, cvtfbuf(correlated[i], 9, buf));
  }
  peak_lag = correlate_peak_lag_f32(&test_corr, &peak);
  assert(peak_lag != INT32_MIN);
  (void)printf("Normalised peak correlation value %s at lag %ld\n", cvtfbuf(peak, 9, buf),
               (long)peak_lag);

  /*
   * Check normalised maximum value. Use epsilon of one (although it succeeds
   * with one epsilon) to allow for accumulated numerical error in the
   * normalisation process and rounding errors when comparing the maximum with
   * the expected value expressed as a floating-point number with only nine
   * decimal places. Do not presume anything about float representation beyond
   * single-precision IEEE 754's precision limits.
   */
  float_t max, min;
  assert(correlated_max_f32(&test_corr, &max) == 7U);
  assert(fepsiloneqf(3, 0.468292892F, max));
  assert(correlated_min_f32(&test_corr, &min) == 4U);
  assert(fepsiloneqf(3, -0.093658581F, min));

  return 0;
}

int main(void) {
  initialise_monitor_handles();
  (void)printf("Hello, World from %s!!!\n", "correlate_f32_test");

  assert(correlate_f32_test() == 0);

  _exit(0);
  return 0;
}
```

The test prescribes two test signals, x and h, expected and actual, respectively. The correlation buffer has space for 100 elements in total. The ring-buffer pair accumulates them dynamically, and the correlator takes all the float added to their used space.

Note, there needs to be a `main()` function, listed at the end. It just prepares the semi-hosting environment, then runs the `correlate_f32_test` function. There is only one test for this binary, but there could be multiple sub-tests per test, limited only by the size and scope of the test.

4.1.1. Results of the test

It runs ARM M4 code in the STM32 emulator—almost the real thing!

```
[ctest] 1: Hello, World from correlate_f32_test!!!
```

```
[ctest] 1: Correlation before normalisation:
[ctest] 1:   correlated[  0] =     0.000000000
[ctest] 1:   correlated[  1] =     0.000000000
[ctest] 1:   correlated[  2] =     0.000000000
[ctest] 1:   correlated[  3] =     0.000000000
[ctest] 1:   correlated[  4] =    -0.250000000
[ctest] 1:   correlated[  5] =    -0.250000000
[ctest] 1:   correlated[  6] =     0.250000000
[ctest] 1:   correlated[  7] =     1.250000000
[ctest] 1: Peak correlation value 1.250000000 at lag 5
[ctest] 1: Correlation after normalisation:
[ctest] 1:   correlated[  0] =     0.000000000
[ctest] 1:   correlated[  1] =     0.000000000
[ctest] 1:   correlated[  2] =     0.000000000
[ctest] 1:   correlated[  3] =     0.000000000
[ctest] 1:   correlated[  4] =    -0.093658581
[ctest] 1:   correlated[  5] =    -0.093658581
[ctest] 1:   correlated[  6] =     0.093658581
[ctest] 1:   correlated[  7] =     0.468292892
[ctest] 1: Normalised peak correlation value 0.468292892 at lag 5
[ctest] 1/1 Test #1: correlate_f32_test ...............   Passed    0.14 sec
[ctest]
[ctest] The following tests passed:
[ctest]     correlate_f32_test
[ctest]
[ctest] 100% tests passed, 0 tests failed out of 1
[ctest]
[ctest] Total Test time (real) =   0.16 sec
[ctest] CTest finished with return code 0
```

5. Conclusions

Semi-hosting QEMU for STM32F4 can seem like black magic: tricky to set up with CTest and tricky to debug. First, it requires an up-to-date version of QEMU for STM32F4. Second, it requires separate binaries for testing, one test, one binary. Multi-test binaries will not work because the `main` function does not accept arguments. Yet the auto-generated CTest test runner expects arguments. This is not a significant issue, as each ELF binary carries a relatively small amount of code and data.

Is semi-hosting a good idea? The embedded system can access the emulator host's file system; it could remove files if it wanted to. Is there a safety issue for the host? Not really; the tests will only do what you ask them to.

Semi-hosted emulation for testing is a compromise. The tests run in an emulated environment—a good thing. The code is the code that runs on the actual target. The emulator does not support all the peripherals, however. Semi-hosting side-steps this issue by separating the application from the target. A layer of application-specific code can exist that transcends the hardware target. The application embedded with the firmware can exist as a separate and distinct test target. The application can run as a test subject independently of its hardware peripherals if the firmware structures the code appropriately.

5.1. Stack Size Considerations

The default stack size on STM32F4 devices is 400_{16} bytes, or 1 024 bytes decimal. Examine the loader script; it appears at the top of the `STM32F407XX_FLASH.ld` file. This may be insufficient for some tests, particularly if a test makes heavy use of the stack; the same goes for the heap. Increase the heap and stack sizes in the linker script used for the test executable. For example, change:

```
/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM); /* end of RAM */
/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0x200; /* required amount of heap  */
_Min_Stack_Size = 0x400; /* required amount of stack */
```

to double up:

```
/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM); /* end of RAM */
/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0x400; /* required amount of heap  */
_Min_Stack_Size = 0x800; /* required amount of stack */
```

Although this is best done using the STM32CubeMX tool since it automatically regenerates the loader script. This could be done for the primary firmware as well, depending on the application requirements and depending on the amount of available RAM. Though this applies only the newlib heap and stack. If the firmware uses FreeRTOS, tasks will allocate their stacks and heaps separately and these "user mode" spaces will become redundant after kernel start.

References

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

[2] QEMU Development Team, QEMU – Open Source Machine Emulator and Virtualiser (2024).
URL https://www.qemu.org

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

[4] FreeRTOS.org, FreeRTOS Real-Time Operating System (2024).
URL https://www.freertos.org

[5] ARM Ltd., GNU ARM Embedded Toolchain (2024).
URL https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain

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

[7] ARM Ltd., ARM CMSIS-DSP Library (2024).
URL https://arm-software.github.io/CMSIS_5/DSP/html/index.html