STM32CubeMX and STM32CubeIDE thread-safe solution

Introduction

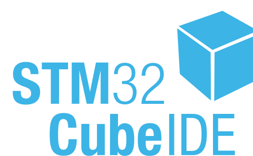
This application note describes the thread-safe solution, implemented in STM32CubeMX and STM32CubeIDE, to ensure safe and reliable operation with respect to concurrent calls to C library functions:

- The thread-safe issue and the solution described in this application note is only related to functions containing a critical section, operating on one single instance of global data, like `malloc()` or `free()`.
- The application note does not address reentrance as there is no locking mechanism in the thread-safe solution protecting the user against reentrance-related issues. It is instead the user's responsibility to make sure that each C library function, like `strtok()`, is called with its own thread context.

The C libraries, bundled with the toolchains promoted in the STM32Cube ecosystem, are by default not protected against concurrency. This means that the usage of `malloc()`, `free()`, or any other function implicitly calling these functions is not safe in multi-threaded systems. These multi-threaded systems can be either bare-metal or RTOS-based systems.

This application note is a guideline to get started with the thread-safe solution, create, and build projects with the suitable thread-safe strategy to ensure that the supported C libraries are protected against concurrency. The proposed RTOS strategies and their implementations apply to the FreeRTOS™ real-time operating system only.

Section 2 presents the thread-safe issue and the proposed solution, while Section 3 and Section 4 describe the implementation of thread-safe setting in STM32CubeMX and the various files generated, using the compatible toolchains. Section 5 shows common bare-metal and FreeRTOS™ examples illustrating how to ensure safe application by adopting the thread-safe solution.



1 General information

1.1 References

- STM32CubeMX for STM32 configuration and initialization C code generation (UM1718)
- STM32CubeIDE user guide (UM2609)[1]

1. The current application note complements the thread-safe wizard section in STM32CubeIDE user guide.

1.2 Compatible toolchains

The STM32CubeMX and STM32CubeIDE thread-safe solution presented in this application note works with the minimum versions of the following toolchains and software tools:

- STMicroelectronics - STM32CubeMX: V6.3.0
- STMicroelectronics - STM32CubeIDE: v1.7.0
- IAR Systems® - IAR Embedded Workbench® (EWARM): V8.50
- Keil® - Microcontroller Development Kit for Arm®-based microcontrollers (MDK-ARM): V4.73

Note: IAR Systems is a registered trademark owned by IAR Systems AB.
Arm and Keil are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 Thread-safe solution background

Making sure that applications are thread-safe is a struggle in the whole embedded industry. The problems become more frequent with the introduction of larger-memory embedded systems in combination with increased RTOS usage. The various software toolchains considered along with STM32CubeMX – MDK-ARM, EWARM and STM32CubeIDE – face similar problems with respect to thread safety. Some user-contributed solutions exist today, which are not fully compatible with the STM32Cube ecosystem. The thread-safe solution presented in this application note is developped by STMicroelectronics to comply with the STM32Cube ecosystem.

2.1 Description of the thread-safe issue

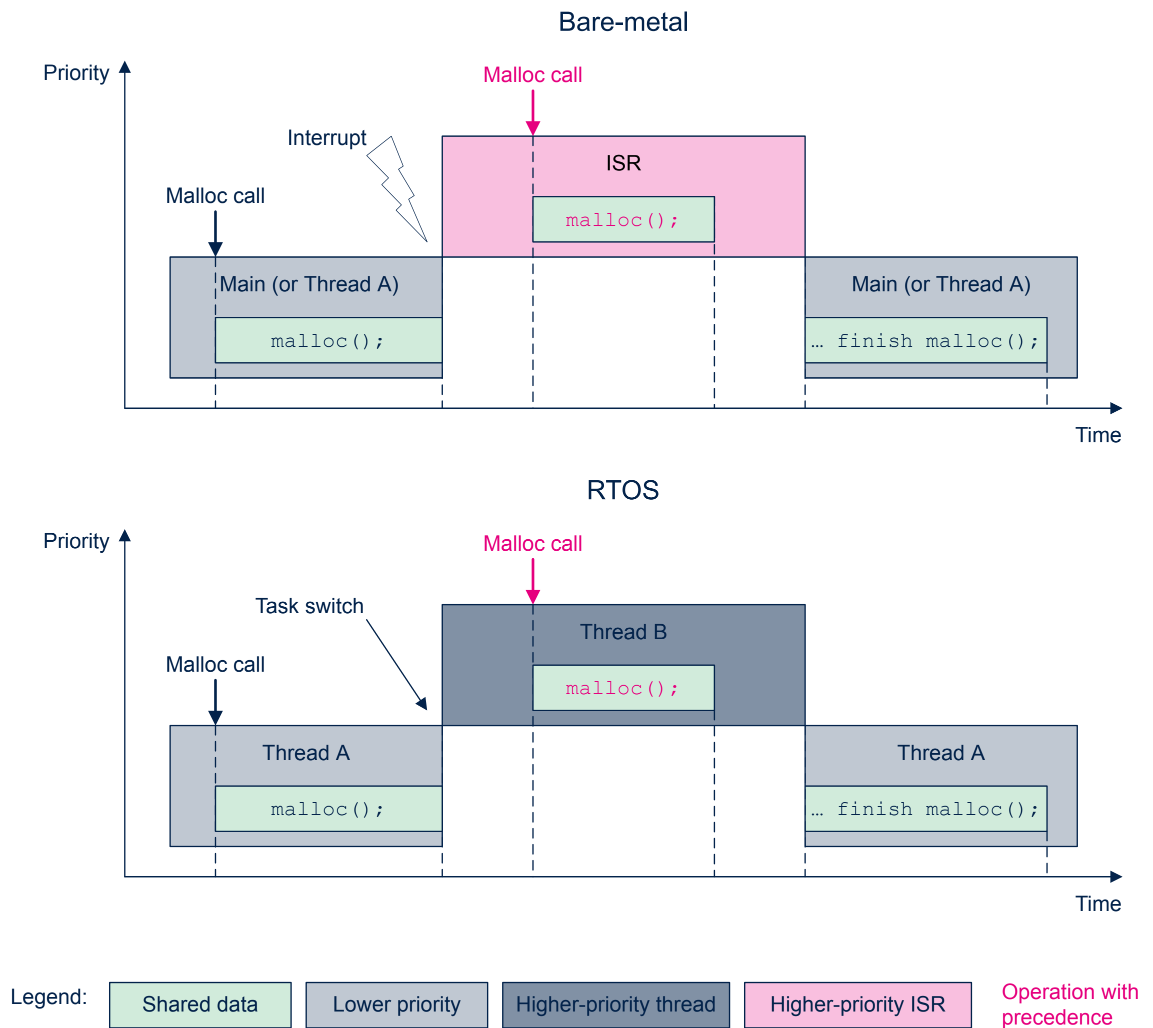
The terms thread and multi-threaded apply to both bare-metal and RTOS-based application:

- RTOS application: two threads or tasks in an RTOS or an ISR (interrupt sevice routine)
- Bare-metal application: a main thread is interrupted by an ISR, which is also seen as a second execution thread

It is important to note that thread-safe issues are not exclusive to RTOS-based applications; The problem exists also in bare-metal applications, where interrupt sevice routines are allowed calls to C library functions. The thread-safe issue can arise in a multi-threaded application where two threads try to manipulate one instance of shared global data like `malloc()` or `free()`. To illustrate the problem of thread safety, two common use cases are considered in Figure 1, in which the problem can arise in a bare-metal or RTOS application. The application code (main or thread) makes a call to `malloc()` to allocate some memory. `malloc()` contains a critical section, where the memory pointer is updated to point to the next free memory area. If the CPU is allowed to switch the context while `malloc()` is executing the critical section where this memory pointer is updated, there is a risk that `malloc()` returns the same memory pointer to both threads. Both threads are then writing data to the same memory block, which means that the memory contents is corrupted.

It is also worth to highlight that the C library can make less obvious calls (implicit calls) leading to similar issues. As an example `printf()` can make a call to `malloc()`. For the end user, the thread-safe issue described above is very difficult to debug. The memory contents is corrupted, but the application may not crash until much later in the execution flow. At this point, it is quite difficult to identify that the root cause is a memory corruption caused long time ago in a completely different module of the application.

Figure 1. Use case examples

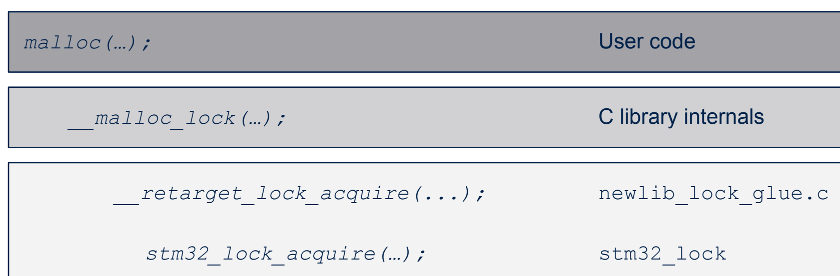


2.2 Technical solution

> ***Tip***: *To proceed further, review the* `stm32_lock.h` *in parallel with this application note for a better understanding. This file can be generated by STM32CubeMX as shown in Section 4 .*

The C library is built with retargetable locking. This means that functions that are at risk of introducing thread-safe issues contain lock and unlock hooks. The lock hook is called at entry of the critical section, and the unlock hook is called at the exit of the critical section. The thread-safe solution implements the required lock-related functions. Figure 2 indicates a simplified interaction between the user code, the C library and the thread-safe solution upon a call to `malloc()`.

Figure 2. Thread-safe solution



The thread-safe solution is a flexible solution, offering different thread-safe strategies to the users based on their requirements. Each strategy represents a unique implementation of the lock-related functions.

All lock-related functions are implemented in the `stm32_lock.h` file. This file is generic and applies regardless of whether the toolchain used is EWARM, MDK-ARM or STM32CubeIDE. For each supported toolchain (actually C library), a dedicated glue file is provided. For STM32CubeIDE, this corresponds to `newlib_lock_glue.c`. This file acts as the glue between the hooks in the C library and the actual lock implementation in file `stm32_lock.h`. Each strategy has advantages and disadvantages, which is why the user must make an active choice among the possible strategies.

The thread-safe solution supports five strategies for handling thread-safe locks. The strategies are divided into generic strategies and RTOS-dependent strategies.

- Generic strategies
  - Strategy #1: user-defined solution for handling thread safety.
  - Strategy #2: allows lock usage from interrupts.

    This implementation ensures thread safety by disabling all interrupts during, for instance, calls to `malloc()`.
  - Strategy #3: denies lock usage from interrupts.

    This implementation assumes single-thread execution and denies any attempt to take a lock from ISR context.
- FreeRTOS™-based strategies
  - Strategy #4: allows lock usage from interrupts.

    Implemented using FreeRTOS™ locks. This implementation ensures thread safety by entering RTOS ISR capable critical sections during, for instance, calls to `malloc()`. This implies that thread safety is achieved by disabling low-priority interrupts and task switching. High-priority interrupts are however not safe.
  - Strategy #5: denies lock usage from interrupts.

    Implemented using FreeRTOS™ locks. This implementation ensures thread safety by suspending all tasks during, for instance, calls to `malloc()`.

3 STM32CubeMX thread-safe settings

Thread-safe setting menu

- Under [**Project Manager Settings tab**], a [**Thread-safe Settings**] section is available as shown in Figure 3. It is applicable to all STM32 microcontrollers and the supported IDEs are: EWARM, MDK-ARM, STM32CubeIDE.

  Note that the thread-safe section is disabled (grayed) when another IDE is selected.
- Enable checkbox [**Enable multi-threaded support**] so that a list of five different thread-safe strategies is proposed in [**Thread-safe locking strategy**]:
  - *Default*: mapping suitable strategy depending on RTOS selection.
  - *Generic Strategy #1*: custom implementation.
  - *Generic Strategy #2*: allows lock usage from interrupts.
  - *Generic Strategy #3*: denies lock usage from interrupts.
  - *FreeRTOS Strategy #4*: allows lock usage from interrupts: Depends on FreeRTOS™.
  - *FreeRTOS Strategy #5*: denies lock usage from interrupts.

Figure 3. **Thread-safe settings**

Thread-safe strategy selection

- Select a strategy as shown in Figure 3.
- The use of the *Default* setting maps the suitable strategy depending on RTOS selection. This helps the user to automatically set up the recommended strategy proposed by the tool depending on the use of an RTOS or not. All strategies are always available (whatever the configuration, the selected microcontroller, or the FreeRTOS™ availability).
  - If FreeRTOS™ is disabled in STM32CubeMX, strategy #2 is automatically selected.
  - If FreeRTOS™ is enabled in STM32CubeMX, strategy #4 is automatically selected.

Multi-core projects

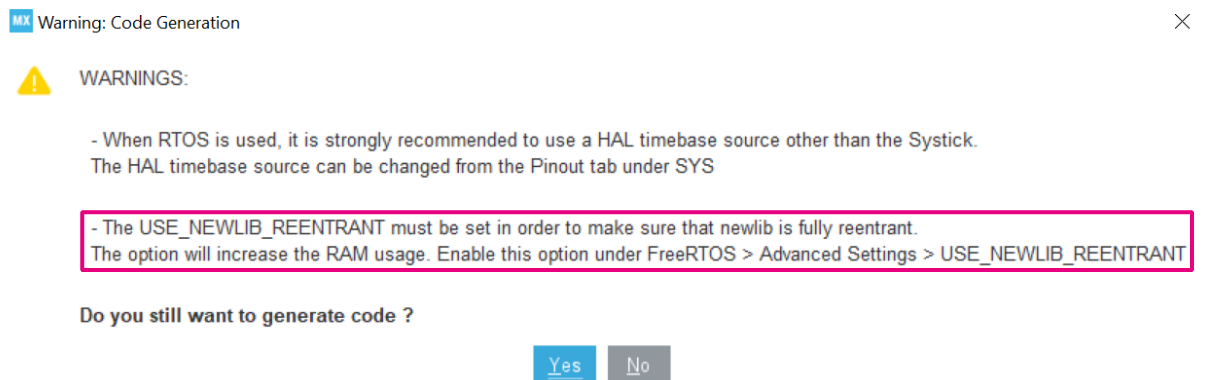
In the case of multi-core projects, a thread-safe section is available for each core project.

FreeRTOS™ applications

In the case of FreeRTOS™ applications, FreeRTOS™ manages reentrance support for the C library functions depending on `_reent struct`, like `strtok()`.

If the project is configured using STM32CubeMX while used in STM32CubeIDE, the `configUSE_NEWLIB_REENTRANT` parameter must be equal to 1. STM32CubeMX checks this constraint and displays a warning message at the code generation step if this is not respected (refer to Figure 4). This flag allows a `newlib _reent struct` allocation for each created task; The scheduler updates the global impure pointer to point at the `_reent struct` of the activated task. This setting protects only against reentrance upon task switches and does not cover reentrant functions called from the ISR. Note also that reentrance support increases the RAM usage.

Figure 4. Generation warning



4 Project configuration and file structure using software toolchains

Three files must be added into the project structure:

- `stm32_lock.h`: defines the different lock strategies.
- `xxx_lock_glue.c` (depends on the toolchain): implements the necessary locking glue to protect C library functions and the initialization of local static objects in C++.
- `stm32_lock_user.h`: user file to be used to implement a custom strategy.

The user can update manually the `stm32_lock_user.h`, which is kept by STM32CubeMX during code re-generation.

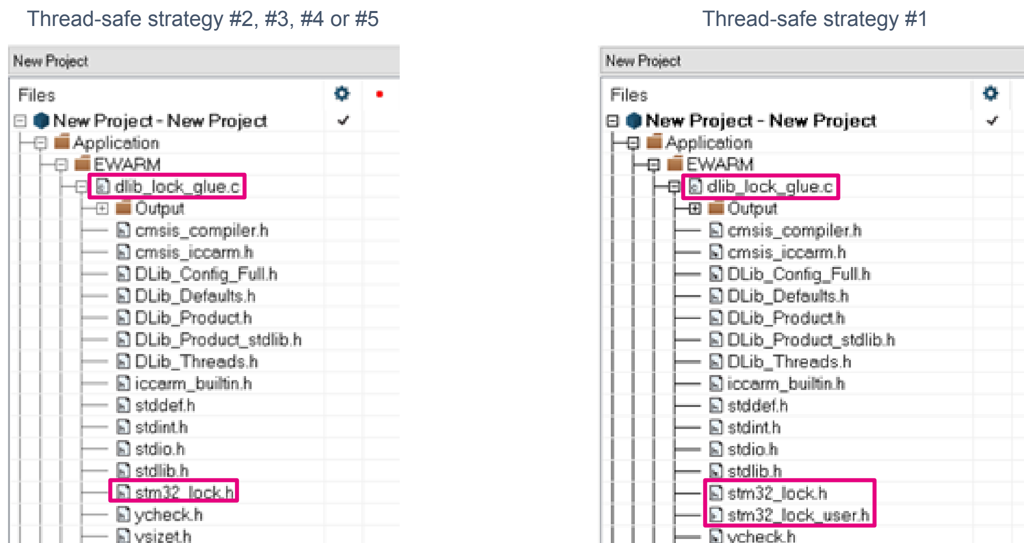
4.1 EWARM toolchain

Single-core project

In the case of a single-core project:

- If the user selects strategy #2, #3, #4 or #5, two files are generated: `stm32_lock.h` and `dlib_lock_glue.c` (refer to Figure 5)
- If the user selects strategy #1, three files are generated: `stm32_lock.h`, `dlib_lock_glue.c` and `stm32_lock_user.h` (refer to Figure 5)

Figure 5. EWARM single-core project configuration



Multi-core project

In the case of a multi-core project, the same files (`stm32_lock.h` and `dlib_lock_glue.c`) are referenced in each core project, and a separate file (`stm32_lock_user.h`) is used per core project.

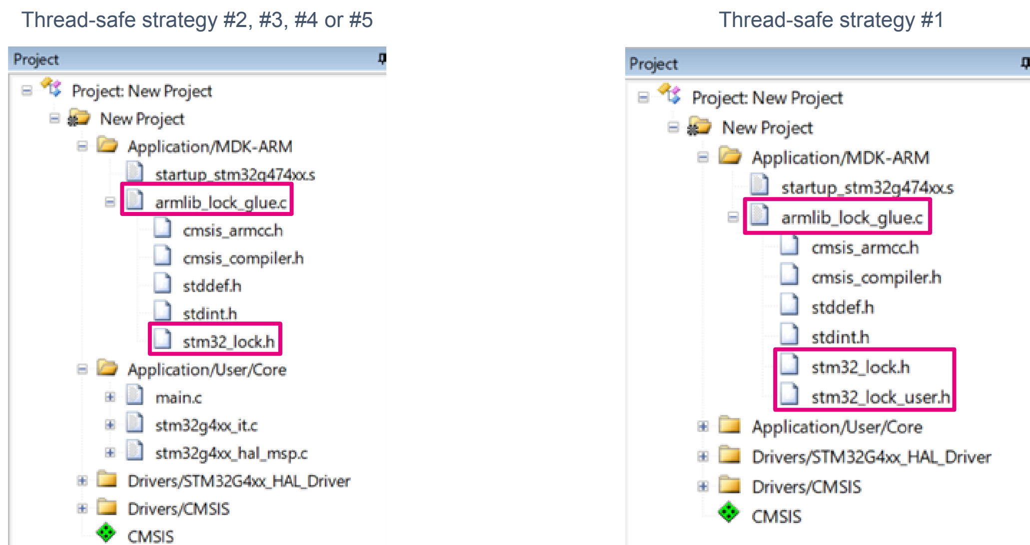
4.2 MDK-ARM toolchain

Single-core project

In the case of a single-core project:

- If the user selects strategy #2, #3, #4 or #5, two files are generated: `stm32_lock.h` and `armlib_lock_glue.c` (refer to Figure 6)
- If the user selects strategy #1, three files are generated: `stm32_lock.h`, `armlib_lock_glue.c` and `stm32_lock_user.h` (refer to Figure 6)

Figure 6. **MDK-ARM single-core project configuration**



Multi-core project

In the case of a multi-core project, the same files (`stm32_lock.h` and `armlib_lock_glue.c`) are referenced in each core project, and a separate file (`stm32_lock_user.h`) is used per core project.

4.3 STM32CubeIDE toolchain

Single-core project

In the case of a single-core project:

- If the user selects strategy #2, #3, #4 or #5, two files are generated: `stm32_lock.h` and `newlib_lock_glue.c` (refer to Figure 7)
- If the user selects strategy #1, three files are generated: `stm32_lock.h`, `newlib_lock_glue.c` and `stm32_lock_user.h` (refer to Figure 7)

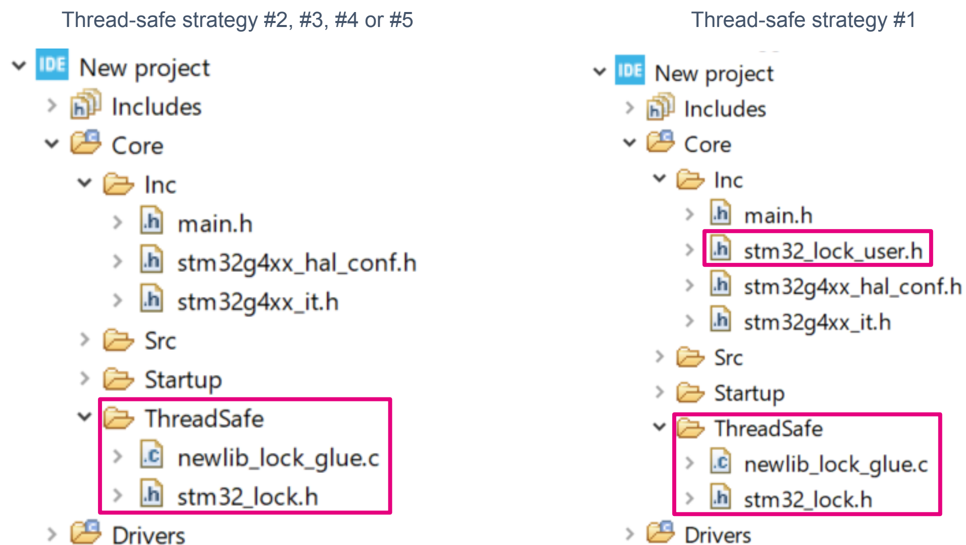


Figure 7. **STM32CubeIDE single-core project configuration**

Multi-core project

In the case of a multi-core project, the same files (`stm32_lock.h` and `newlib_lock_glue.c`) are referenced in each core-project, and a separate file (`stm32_lock_user.h`) is used per core project.

Empty project

In the case of STM32CubeIDE empty projects, it is possible to generate a thread-safe solution. For more details, refer to the section *Thread-safe wizard for empty projects and CDT™ projects* of the STM32CubeIDE user guide (UM2609), which is available on *www.st.com*.

5 Use case examples

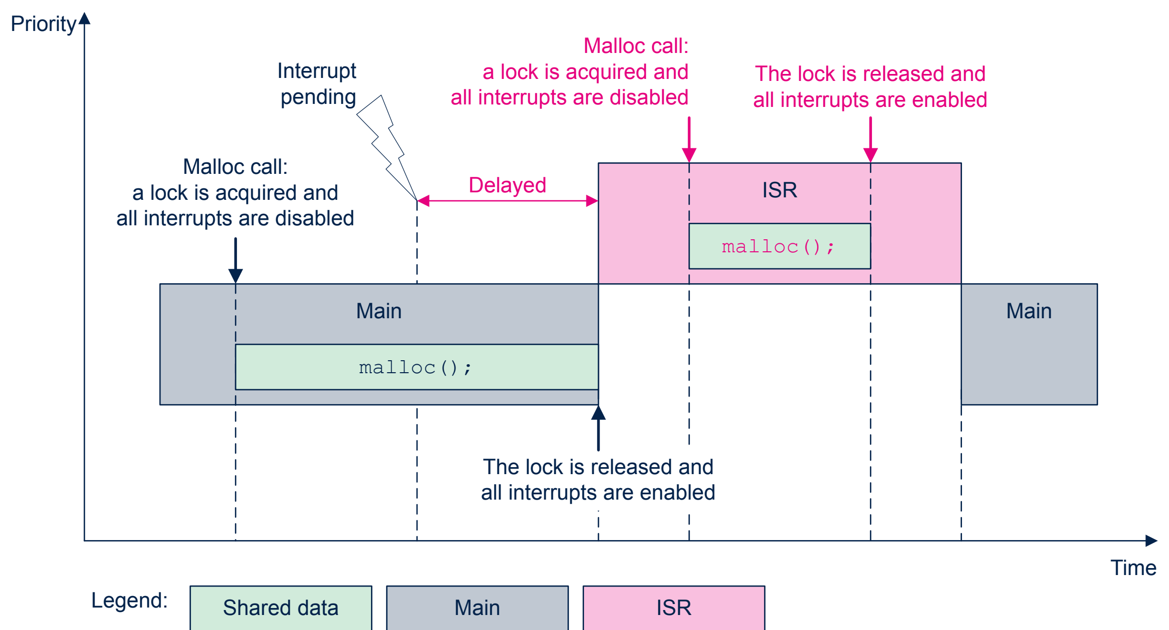
To support the use cases described in Figure 1. Use case examples, this chapter details some examples showing the deployment of both bare-metal and FreeRTOS™ strategies. Each strategy provides lock implementation with its own advantages and disadvantages.

5.1 Bare-metal application

5.1.1 Use case with strategy #2

This strategy ensures thread safety by temporarily disabling all interrupts when the lock is acquired (refer to Figure 8). Interrupts are left pending, and the currently active execution context cannot be preempted. When the execution of `malloc()` is completed, the lock is released, which re-enables interrupts. Hence, the CPU is able again to switch context and execute the interrupt. If the ISR makes a call to `malloc()`, it takes a lock, finishes its execution, and releases the lock. The application is safe in terms of reentrance from the ISR and the shared data is not corrupted. However, the side-effect of this strategy is that interrupts are delayed.

Figure 8. Use case with strategy #2



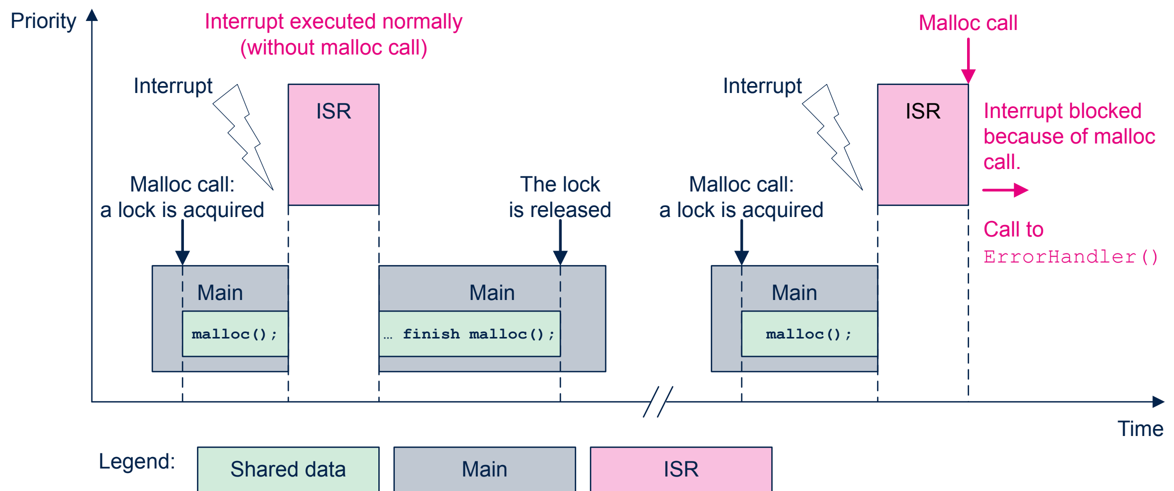
5.1.2 Use case with strategy #3

This strategy allows context switches from the ISR when the lock is acquired (refer to Figure 9) and interrupts are not delayed. But with this strategy, it is not possible to obtain a lock from inside the ISR context. Consequently, when a C library function tries to acquire a lock from inside the ISR context, the attempt is denied and the CPU gets stuck in the `ErrorHandler()`.

On the left in Figure 9, main() calls malloc(). The malloc() execution is interrupted, but the ISR does not call malloc(); Consequently, there is no attempt to acquire a lock from the ISR context. No data is corrupted since malloc() can finish the critical section after returning from the ISR context.

On the right in Figure 9, main() calls malloc(). The malloc() execution is interrupted and the ISR calls malloc(); Consequently, there is an attempt to acquire a lock from the ISR context. Acquiring locks from the ISR is not allowed and the application hangs in the ErrorHandler(). The intention is to send a clear signal to the developer that the C libraries must not be used in this way. The application is considered safe by making the developer aware about dangerous usage of C library functions inside the ISR context.

Figure 9. Use case with strategy #3



5.2 FreeRTOS™ application

In an RTOS-based application, there are three possible sources with respect to concurrent calls to C library functions:

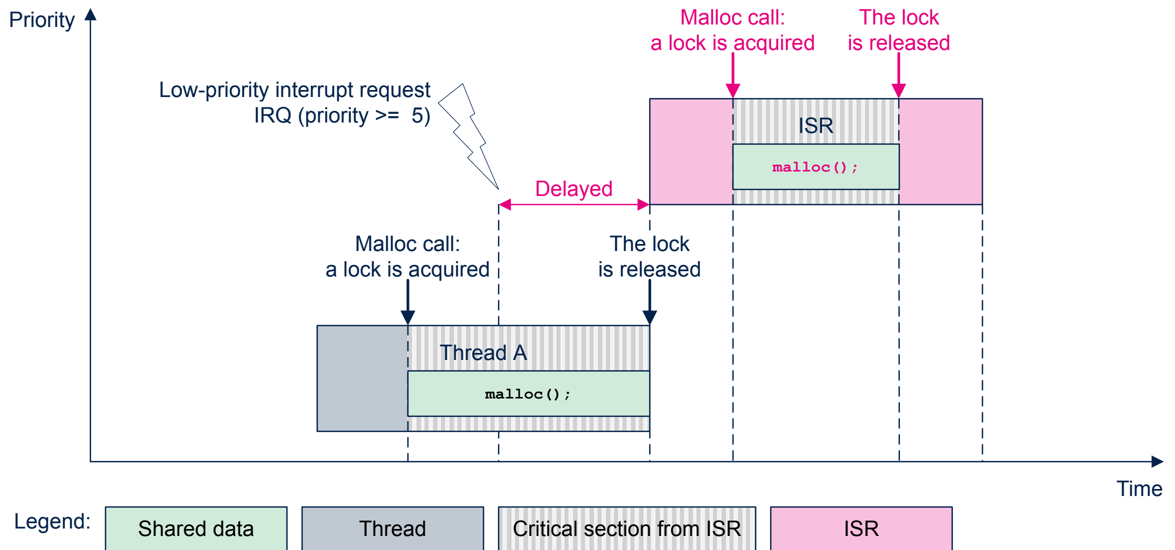
- Low-priority interrupt
  - Used for less time-sensitive operation
  - Used to tick the RTOS
  - Used to perform switching between FreeRTOS™ tasks
- High-priority interrupt: may be required in the application, for example to perform time-critical operations
- Task switch

5.2.1 Use case with strategy #4

This strategy ensures thread safety by entering RTOS ISR capable critical sections during call to malloc() via macro taskENTER_CRITICAL_FROM_ISR. The taskENTER_CRITICAL_FROM_ISR macro is implemented slightly differently depending on which Cortex®-M core the project is targeting. When the lock is acquired, malloc() enters a critical section and therefore low-priority interrupts and task switches are disabled. However, high-priority interrupt can still happen at the cost of not being safe from concurrent C library function calls.
This implementation, by default, supports two levels of recursive locking. The number of recursive levels is configurable.

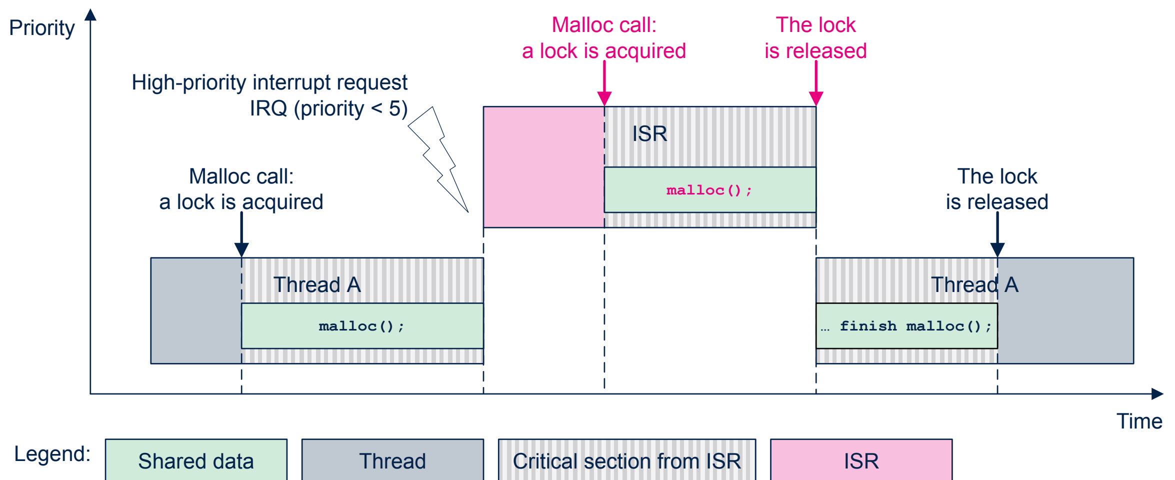
As illustrated in Figure 10, low-priority interrupts (high priority value) are delayed until `taskEXIT_CRITICAL_FROM_ISR` finishes and the lock is released. Hence, `malloc()` is protected from other concurrent calls and the shared data is not corrupted.

Figure 10. Use case with strategy #4 (low-priority interrupt)



In the case of high-priority interrupts (refer to Figure 11), the `malloc()` call is executed and therefore the shared data can be corrupted.

Figure 11. Use case with strategy #4 (high-priority interrupt)

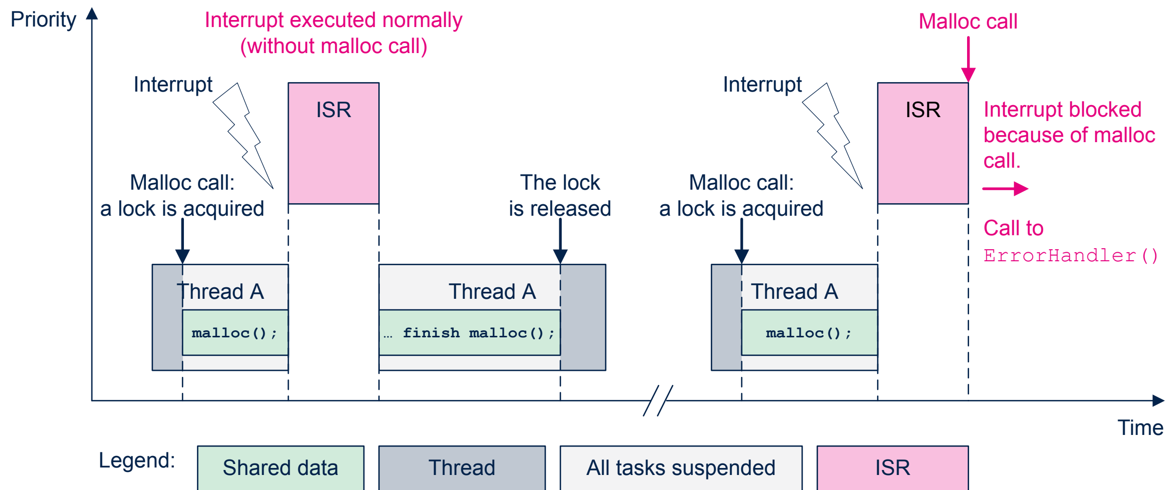


Note: The IRQ priority threshold at 5 is only an example. It is the user's responsibility to configure the priority masking.

5.2.2 Use case with strategy #5

This strategy ensures thread safety by suspending all tasks during a `malloc()` call but leaves interrupts enabled (refer to Figure 12). When the lock is acquired by `malloc()`, task switches cannot happen until the lock is released. However, interrupts are allowed to switch the execution. But if the ISR attempts to acquire a lock from the ISR context, the application gets trapped in the `ErrorHandler()` and hangs. The application is thus considered safe also from the ISR context by making the developer aware of dangerous usage of C library functions. This approach is similar to strategy #3.

Figure 12. Use case with strategy #5



6 Key takeaways

- Using C library functions, which may need synchronization, at the same time from both the main application (or thread) and from the ISR, is not recommended.
- The majority of the support requests relating to thread-safety issues concern the usage of `malloc()` in FreeRTOS™ applications. The thread-safe solution protects `malloc()` and other C library functions with respect to concurrent calls while executing critical sections. Additionally, enabling the `configUSE_NEWLIB_REENTRANT` flag ensures that any function with reentrance like `strtok()` can be used safely across multiple threads since each task gets a unique `_reent struct`. But, the user is not protected if `strtok()` is called from an ISR. Therefore, the user must be careful with C library calls from ISRs in general and manage `_reent struct` instances manually.
- The user must be careful about implicit call to thread-safety function. For example, `printf()` can under certain circumstance make a call to `malloc()`.
- The implementation of locks for `newlib` function temporarily disables interrupts (strategies #2 and #4) and thereby affects the real-time behavior of the application. Consider the implications of disabling interrupts in the application carefully before adopting the lock mechanisms.

Revision history

Table 1. **Document revision history**

Date	Revision	Changes
30-Nov-2021	1	Initial release.

Contents

1 General information 2
1.1 References 2
1.2 Compatible toolchains 2
2 Thread-safe solution background 3
2.1 Description of the thread-safe issue 3
2.2 Technical solution 5
3 STM32CubeMX thread-safe settings 6
4 Project configuration and file structure using software toolchains 8
4.1 EWARM toolchain 8
4.2 MDK-ARM toolchain 9
4.3 STM32CubeIDE toolchain 10
5 Use case examples 11
5.1 Bare-metal application 11
5.1.1 Use case with strategy #2 11
5.1.2 Use case with strategy #3 11
5.2 FreeRTOS™ application 12
5.2.1 Use case with strategy #4 12
5.2.2 Use case with strategy #5 14
6 Key takeaways 15
Revision history 16
List of tables 18
List of figures 19

List of tables

Table 1. Document revision history 16

List of figures

Figure 1. Use case examples . . . 4
Figure 2. Thread-safe solution . . . 5
Figure 3. Thread-safe settings . . . 6
Figure 4. Generation warning . . . 7
Figure 5. EWARM single-core project configuration . . . 8
Figure 6. MDK-ARM single-core project configuration . . . 9
Figure 7. STM32CubeIDE single-core project configuration . . . 10
Figure 8. Use case with strategy #2 . . . 11
Figure 9. Use case with strategy #3 . . . 12
Figure 10. Use case with strategy #4 (low-priority interrupt) . . . 13
Figure 11. Use case with strategy #4 (high-priority interrupt) . . . 13
Figure 12. Use case with strategy #5 . . . 14

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, please refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2021 STMicroelectronics – All rights reserved