- **Clarification**, page 190-191
  - `TaskA` and `TaskC` use functions to set the red LED on and off (`RedLed.On()`, `RedLed.Off()`). For these functions to work properly here, the functions must be atomic operations. However, the requirement for atomic operations isn't discussed.
    - For the board and software used here, those functions do appear to be atomic operations. However, if those functions weren't atomic operations, the way they are used here would probably be incorrect.
    - And, the code is not portable to systems in which those functions are not atomic.
  - Why the functions must be atomic operations
    - Functions that turn an LED on and off are likely to have shared data. If two tasks can operate on that shared-data concurrently, those operations must be thread-safe.
    - In the code here, the critical-sections turn the red LED off, but the LED is turned-on outside of the critical sections. So, one task could be turning the LED off while another is turning it on.
  - From investigating the system's source code, it appears those functions are atomic operations.
    - The functions are implemented here:
      - `C:/projects/packtBookRTOS/BSP/Nucleo_F767ZI_GPIO.c`
    - They call `HAL_GPIO_WritePin()`, which is implemented here:
      - `C:/projects/packtBookRTOS/Drivers/STM32F7xx_HAL_Driver/Src/stm32f7xx_hal_`
    - The comments for `HAL_GPIO_WritePin()` state it is an atomic operation:
      - "*This function uses GPIOx_BSRR register to allow atomic read/modify...*"

- **Bug** in the book, page 193
  - The figure has errors similar to those described for the figure on page 186.
  - The events shown in the Terminal window are not those shown in the Events-List and Timeline windows:
    - The differing timestamps reveal that.
    - Also, in the Terminal window, there's a message from TaskB at 30.385, but it's not in the Events List window.

2 Section: Using Mutexes (pg 193-197)

- **Bug** in the book and code (`mainMutexExample.c`), page 194-195
  - The code has the same bug described earlier for `mainSemPriorityInversion.c`.
    - The spin-loop does not run long enough to potentially cause priority inversion.
  - `mainMutexExample.c` was tested, and it produced results very similar to those from `mainSemPriorityInversion.c`. TaskA's semaphore-request did not timeout, however, it wasn't due to the mutexes being used.
  - `mainMutexExample.c` was fixed by making TaskB's spin-loop run longer, as with the fix in `SemPriorityInversion.c`
    - It was changed to run 18 to 28ms: `lookBusy(StmRand(1692000, 2632000))`
    - Testing results for 7.5 minutes are what is expected:
      - TaskA: failed to get semaphore: 0 times; semaphore taken: 1700 times
      - TaskC: failed to get semaphore: 608 times; semaphore taken: 1092 times
      - TaskB: iterations: 7600

- **Bug** in the book, page 195
  - The program discussed is `mainMutexExample.c`. Its mutex-use and task-scheduling are described, but the description is not fully correct. The incorrect parts are underlined:
    - *TaskA returns the mutex, but it is immediately taken again. This is caused by a variable amount of delay in TaskA between calls to the mutex. When there is no delay, TaskC isn't allowed to run between when TaskA returns the mutex and attempts to take it again.*
  - "*TaskA returns the mutex, but it is immediately taken again.*"
    - The sentence's second phrase is saying that `TaskA` immediately takes the mutex again, at (2) in the example. This is the case referred to as "*When there is no delay,*"
    - However, after `TaskA` returns the mutex there is always a delay of at least 5 ticks:
      - `vTaskDelay(StmRand(5,30));` // In the .c file (different than book)
      - `vTaskDelay(StmRand(10,30));` // In the book, pg 190
    - Also, that description is inconsistent with the Timeline, as the Timeline does not show TaskA returning the mutex and immediately taking it