

Exercise - 02c

Atomic Bit Manipulation for GPIO Access

Andreas Habegger | Adrian Steiner
BTS3230 | Version 1.0.0 of 15.09.2026

Please be aware that the content is subject to change at any time. For the latest version, please check the website.

In [Ex-02a: GPIO Introduction](#), you changed individual bits of a GPIO register using ordinary bit-wise operators. This approach works well as long as nothing else can access the same register at the same time. As soon as more than one part of a program can modify the same register, a plain read-modify-write sequence is no longer safe. In this exercise, you will compare two techniques for manipulating GPIO bits **atomically**, i.e. in a way that cannot be corrupted by something else happening in between.

Objectives

- ▶ Understand why a read-modify-write sequence on a shared register can fail
- ▶ Compare different techniques for atomic bit manipulation
- ▶ Understand the STM32 GPIO port bit set/reset register (BSRR)
- ▶ Weigh the advantages and disadvantages of each technique

Outcomes

- ▶ Explain why a read-modify-write sequence can be unsafe
- ▶ Use the GPIOx_BSRR register for atomic pin control
- ▶ Select an appropriate bit-manipulation technique for a given situation

Description

The bit-wise operators you used in the previous exercise (`|=` , `&=` , `^=`) all follow the same pattern at the register level: the current register value is **read**, modified in a CPU register, and **written** back. This sequence is called a **read-modify-write (RMW)** operation.

A RMW operation is not indivisible. Between the read and the write, several CPU cycles pass during which something else could, in principle, access the same register. If that other activity also modifies the register, the result of one of the two changes can be silently lost, because the later write overwrites the register with a value computed from now-outdated data. This class of problem is called a **lost update**, and it is one of the most common sources of hard-to-reproduce bugs in embedded software.

Your superloop application from [Ex-02a: GPIO Introduction](#) and [Ex-02b: Flash GPIO with Blocking Delay](#) runs everything sequentially, so this specific hazard cannot occur in the current application. Later exercises will introduce ways for more than one part of your program to access the same register, at which point the technique you use to change a bit will start to matter. Understanding atomic register access now will save you a difficult debugging session later.

Note

Only the Nucleo-64 board is used in this task.

Tasks

The Read-Modify-Write Hazard

Consider the following, familiar way of setting a GPIO output pin:

```
GPIOx->ODR |= (1 << LED_PIN);
```

 Question

- a. Into how many separate steps does this single line of C code decompose at the register-access level?
- b. Suppose that, after the microcontroller has read `ODR` but before it has written the modified value back, something else changes a *different* bit of the same `ODR` register. What ends up in `ODR` once your code resumes and completes its write?
- c. Why is this dangerous even though your own application code never touches the same *bit* from two different places?

 Solution

- a. Three steps: read the current value of `ODR` into a CPU register, modify that value (the `OR` with the bitmask), and write the result back to `ODR`.
- b. The other change is lost. Your code's write is based on the value of `ODR` from *before* that other change happened, so it overwrites it with stale data.
- c. Because the hazard operates on the whole register, not on individual bits. Any two pieces of code that perform a RMW on the *same register*, even on different bits, can interfere with each other. A GPIO port bundles up to 16 unrelated pins into one register, so code controlling one pin can silently corrupt a pin your main loop controls.

Atomic Set/Reset Register (BSRR)

STM32 GPIO ports provide a dedicated register that was designed specifically to avoid the RMW hazard: the GPIO port bit set/reset register (GPIOx_BSRR). Writing a `1` to bit `n` of this register sets pin `n` of the port; writing a `1` to bit `n + 16` resets pin `n`. Writing a `0` to any bit of BSRR has no effect at all.

 Question

- d. Look up the GPIO port bit set/reset register (GPIOx_BSRR) in section 7.4 GPIO registers of the [reference manual RM0390](#) reference manual. What is its offset from the GPIO port base address, and what do the two 16-bit halves of the register do?
- e. Why is a single write to BSRR inherently atomic, while a write to ODR is not?
- f. Can BSRR be used to toggle a pin directly? Why or why not?

 Solution

- d. The offset and exact bit layout are given in the reference manual; the lower 16 bits (BSy) set the corresponding ODR bit, the upper 16 bits (BRy) reset it.
- e. Writing to BSRR does not require reading the current register value first: the hardware itself only ever changes the bits you asked it to change, and a single store instruction cannot be partially completed. There is no window in which stale data could be written back, because no data is read in the first place.
- f. No. Toggling requires knowing the pin's *current* state, which means you still have to read it from somewhere (ODR or IDR) before deciding whether to set or reset it. That read-then-decide step reintroduces the same kind of race described earlier, even though the final write to BSRR is itself atomic.

Implementation

 Implement the Variants

1. Implement a function (or macro) that sets, clears, and toggles the LED pin using the plain read-modify-write approach on ODR, as in [Ex-02a: GPIO Introduction](#).
2. Implement a second version that sets and clears the LED pin using GPIOx_BSRR.

 Hint

BSRR has no dedicated toggle functionality. You can still implement “toggle” by reading the current pin state from ODR (or IDR) and writing the opposite value to BSRR, but remember that this specific read-then-decide step is not itself atomic.

3. Use both variants, one at a time, to blink the LED in your superloop and confirm that they behave identically from a functional point of view.

Results

Even though both variants produce the same visible behaviour, they are not equivalent in terms of safety or efficiency.



Compare the Variants

4. Open the disassembly / instruction-step view in your debugger and count the number of instructions generated for the set/reset operation in each of the two variants.
5. Based on the instruction count and the CPU clock frequency, estimate the execution time of each variant.



Discussion

The table below summarises the trade-offs between the techniques from this exercise.

Technique	Atomic by itself	Relative cost	Can toggle directly
Read-modify-write on ODR	No	Lowest	Yes
BSRR set/reset	Yes	Low (single store)	No (see hint above)

With your classmates, discuss:

- ▶ Does atomicity matter in code that runs strictly sequentially, with nothing else ever accessing the same register? Why might it still be good practice to prefer the safer variant anyway?
- ▶ If more than one part of your program could someday access this register at the same time, which variant would you prefer, and why?
- ▶ Why does the `BSRR` variant require fewer instructions than the read-modify-write variant?
- ▶ Does atomic access to a single register guarantee that your overall application logic is race-free? What else might still need protecting?