

Exercise - 02d

Atomic GPIO Access via Bit-Banding

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-02c: Atomic Bit Manipulation](#), you compared a plain read-modify-write on `ODR` with the atomic `GPIOx_BSRR` register, and found that `BSRR` still has a limitation: it can only set or reset a bit, never read one back, so toggling still requires a separate, non-atomic read. The STM32F446 (and many more M4 processors) provides a third technique that closes this gap for individual bits: **bit-banding**. In this exercise, you will use bit-banding to access GPIO pins, and weigh its advantages and disadvantages against the two techniques from ex-02c.



Objectives

- ▶ Understand the concept of memory-mapped bit-banding
- ▶ Compute a bit-band alias address for a GPIO register bit
- ▶ Compare bit-banding with the read-modify-write and `BSRR` techniques from ex-02c
- ▶ Identify realistic situations where bit-banding is the right tool



Outcomes

- ▶ Explain how the bit-band alias region maps onto the bit-band base region
- ▶ Calculate a bit-band alias address for a GPIO register bit
- ▶ Access an individual GPIO pin as if it were a plain boolean variable
- ▶ Judge when bit-banding is, and is not, an appropriate choice

Description

The Cortex-M4 core inside the STM32F446 reserves two special 1 MB regions of its address space, called **bit-band regions**: one inside SRAM, one inside the peripheral address space. Every single bit of every word inside a bit-band region is mirrored onto its own dedicated 32-bit word address inside a separate 32 MB **alias region**. Reading or writing that alias word reads or writes exactly the one corresponding bit, leaving every other bit of the underlying register untouched, using a single load or store instruction.

Because this is a single instruction that neither reads the surrounding bits nor needs a bitmask, a write through the alias region is atomic in exactly the same sense as a write to `BSRR` was in the previous exercise, but unlike `BSRR`, the alias region supports both reads and writes of individual bits.

The alias address for a given bit is calculated as follows:

$$\text{alias_address} = \text{alias_base} + (\text{byte_address} - \text{region_base}) \times 32 + \text{bit_number} \times 4$$

Most STM32F446 GPIO registers, including `ODR` and `IDR`, lie inside the peripheral bit-band region, and can therefore be accessed bit by bit through the peripheral alias region.



Note

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

Tasks

Computing a Bit-Band Alias Address

? Question

- Look up the bit-band regions and their alias regions for your Cortex-M4 core (either in the [reference manual RM0390](#) reference manual or the Cortex-M4 programming manual). What are the base addresses of the peripheral bit-band region and its alias region?
- Is the GPIOA peripheral base address inside the peripheral bit-band region?
- Using the formula above, calculate the bit-band alias address of bit 5 (LD2, PA5) of the `GPIOA_ODR` register.

✓ Solution

- The peripheral bit-band region starts at `0x40000000` and its alias region starts at `0x42000000`; both are documented in the reference manual / programming manual sections on bit-banding.
- Yes. GPIOA is mapped at `0x40020000`, which lies inside the `0x40000000 - 0x400FFFFF` peripheral bit-band region.
- Apply the formula with `byte_address` set to the address of `GPIOA_ODR` and `bit_number` set to `5`. Verify your result against the address ranges given in the reference manual.

Advantages and Limitations

? Question

- d. Unlike `BSRR`, a bit-band alias address can be read as well as written. What does this make possible that `BSRR` alone could not do atomically?
- e. Is *toggling* a pin through its bit-band alias address atomic as a whole operation? Why or why not?
- f. Bit-banding is a feature of the Cortex-M core, not of the GPIO peripheral itself. What does this imply for code that relies on it, if it later needs to run on a different microcontroller?
- g. Each individual bit occupies a full 32-bit word in the alias region. Is this a meaningful disadvantage in practice? Why or why not?

✓ Solution

- d. It allows the *current* value of a single bit, e.g. an input pin, to be read atomically as a clean `0 / 1` value, in one instruction, without masking `IDR`.
- e. No. Reading the current value, deciding on the new one, and writing it back are still three separate steps; something else could still change the same bit in between. Bit-banding makes each individual read or write atomic, but not a read-then-write sequence built out of two of them, for exactly the same reason `BSRR`-based toggling was not atomic in the previous exercise.
- f. Such code becomes tied to the Cortex-M3/M4 architecture and is no longer portable across microcontroller families, or even across every Cortex-M core; some cores (and some vendors) do not implement bit-banding at all. It is worth isolating bit-band access behind a small macro or function, so that porting the code later only requires changing one place.
- g. In practice, rarely. The alias region consumes address space, not physical RAM or flash: the architecture reserves a fixed 32 MB alias region for exactly this purpose. It is a characteristic worth knowing, but not a resource you can run out of the way you can with RAM or flash.

Implementation

Implement Bit-Banded GPIO Access

1. Write a macro or inline function that computes the peripheral bit-band alias address for a given register address and bit number.

 Hint

Cast the resulting alias address to a `volatile uint32_t *` before dereferencing it. The value you read or write through it is always `0` or `1`, never a bitmask.

2. Use it to implement set, clear, and toggle functions for the LED pin (LD2, PA5), treating the bit-band alias address as a plain pointer to a single bit.
3. Use it to read the state of the USER button pin through its bit-band alias address, instead of masking `IDR`.
4. Replace the register-manipulation code from [Ex-02a: GPIO Introduction](#) with your bit-band based version, and confirm that the LED still mirrors the button correctly.

Results

 Compare the Three Techniques

5. Extend the instruction-count / timing comparison from the [Ex-02c: Atomic Bit Manipulation](#) Results section with the bit-band variant. How many instructions does the bit-band set/clear operation need, compared to the read-modify-write and `BSRR` variants?
6. Compare the source code of all three variants side by side. Which one is easiest to read at a glance, and why?

 Discussion

The table below summarises the trade-offs between the three GPIO bit-manipulation techniques covered in ex-02c and this exercise.

Technique	Atomic write	Atomic read	Toggle in one instruction	Portable across MCUs
Read-modify-write on ODR / IDR	No	Yes	No	Yes
BSRR set/reset	Yes	N/A (write-only)	No	No (STM32-specific register)
Bit-band alias	Yes	Yes	No	No (Cortex-M-specific feature)

Some situations where bit-banding is genuinely useful:

- ▶ **Bit-banded protocols:** software-driven communication protocols (for example a one-wire sensor bus, or an addressable-LED protocol) need tightly and predictably timed pulses on a GPIO pin. A single, fixed-latency bit-band store is easier to reason about than a compiler-optimised read-modify-write sequence.
- ▶ **Timing markers for measurement:** toggling a spare GPIO pin around a piece of code you want to profile with the Saleae logic analyser. Bit-banding keeps the marker itself to the smallest and most predictable possible instruction overhead, so it disturbs the measurement as little as possible.
- ▶ **Tight polling loops:** repeatedly checking a single input pin, for example in a debounce routine, reads more clearly as a plain boolean test through its bit-band alias than as a masked access to `IDR`.

With your classmates, discuss:

- ▶ Given that both `BSRR` and bit-banding solve the *write* side of the read-modify-write problem, when would you reach for bit-banding instead of `BSRR`?
- ▶ Why might a hardware abstraction library (such as a HAL) avoid using bit-banding, even where it would be faster or clearer than its own register access functions?
- ▶ If your GPIO driver code had to run unmodified on a Cortex-M0 microcontroller, which has no bit-banding, which of the three techniques would you choose, and why?

 Generalise the Bit-Band Macro

Your macro from the Implementation section most likely only works for the specific register and pin you used it on.

1. Generalise it so that it accepts any GPIO port, any register (`ODR` , `IDR` , ...), and any pin number, and derives the correct alias address in every case.

 Question

- ▶ Does your generalised macro produce the correct alias address for a pin on GPIOD, given that its peripheral base address differs from GPIOA's?