Bit-Banding GPIO HAL on STM32

Ashok Ashwin

February 15, 2026

Contents

I What is Bit-Banding? 2

1 The Bit-Band Memory Model 2
1.1 The Alias Address Formula . . . 2
1.2 Why Bit-Banding Matters . . . 2

II The GPIO Bit-Band HAL 3

2 Project Structure 3

3 The Core Bit-Band Macro 3

4 Clock Enable via Bit-Banding 3

5 Pin Configuration 4

6 Pin Set, Clear, Toggle, and Read 4

III Application: Blinking PC13 5

7 The main.c Source 5

8 Startup and Linker 5

9 Makefile 6

IV Bit-Banding in Depth 8

10 Traditional Register Access vs Bit-Banding 8
10.1 Traditional: Read-Modify-Write on ODR . . . 8
10.2 Bit-Band: Single Store . . . 8

11 When to Use Bit-Banding 8

12 Limitations 8

Part I
What is Bit-Banding?

ARM Cortex-M3 processors provide a hardware feature called bit-banding that allows atomic, single-instruction access to individual bits in memory. Instead of the conventional read-modify-write pattern required to manipulate a single bit in a register, bit-banding maps each bit in a 1 MB region to a full 32-bit word in a 32 MB alias region. Writing to the alias word sets or clears exactly one bit in the original register — no masking, no shifting, no race conditions.

This project demonstrates bit-banding by building a small GPIO HAL for the STM32F103C8T6 (Blue Pill). The HAL uses bit-band alias addresses to enable clocks, set pins, clear pins, and toggle pins without read-modify-write sequences.

1 The Bit-Band Memory Model

The Cortex-M3 defines two bit-band regions:

- Peripheral bit-band region: 0x40000000 – 0x400FFFFF (1 MB)
- Peripheral bit-band alias: 0x42000000 – 0x43FFFFFF (32 MB)
- SRAM bit-band region: 0x20000000 – 0x200FFFFF (1 MB)
- SRAM bit-band alias: 0x22000000 – 0x23FFFFFF (32 MB)

Every bit in the 1 MB region is mapped to a word in the 32 MB alias region. Since 1 MB = 8 million bits and each alias word is 4 bytes, the alias region occupies 32 MB.

1.1 The Alias Address Formula

For a peripheral register at address A and bit number n (0–31):

$$\text{alias\_addr} = 0\text{x}42000000 + (A - 0\text{x}40000000) \times 32 + n \times 4$$

The multiplication by 32 converts byte offset to bit offset (8 bits per byte × 4 bytes per alias word), and the multiplication by 4 selects the correct word within that byte's 8 alias words.

1.2 Why Bit-Banding Matters

Consider enabling the GPIOC clock. The traditional approach uses a read-modify-write pattern:

```
// Traditional read-modify-write: 3 bus transactions
RCC_APB2ENR |= (1 << 4);   // READ register, OR with mask, WRITE back
```

This compiles to a load, an OR, and a store — three bus transactions. If an interrupt fires between the load and the store, it could modify the same register, and the write-back would overwrite that change.

With bit-banding:

```
// Bit-band: 1 bus transaction, atomic
BITBAND_PERIPH(RCC_APB2ENR_ADDR, 4) = 1;  // Single STORE to alias
```

This is a single store instruction. The bus hardware translates it into an atomic bit-set on the actual register. No read, no mask, no race.

Key advantages:

- Atomic: A single store cannot be interrupted mid-operation
- No masking: No shift or OR/AND operations needed in software
- Interrupt-safe: No window for data corruption between read and write
- Smaller code: One instruction instead of three

Part II
The GPIO Bit-Band HAL

2 Project Structure

- gpio_bitband.h
- main.c
- startup.s
- linker.inv
- Makefile

The HAL is implemented as a single header file (gpio_bitband.h) with static inline functions. This ensures zero call overhead — the compiler inlines every function, producing the same machine code as hand-written register access.

3 The Core Bit-Band Macro

```
#define PERIPH_BASE        0x40000000U
#define PERIPH_BB_BASE     0x42000000U

#define BITBAND_PERIPH(reg_addr, bit)
    (*(volatile unsigned int *)(PERIPH_BB_BASE +
        (((unsigned int)(reg_addr) - PERIPH_BASE) * 32U) +
        ((bit) * 4U)))
```

This macro takes a peripheral register address and a bit number, then computes the 32-bit alias address. The result is a volatile dereference, so it can be used on both sides of an assignment:

```
BITBAND_PERIPH(addr, bit) = 1;   // Set bit
BITBAND_PERIPH(addr, bit) = 0;   // Clear bit
x = BITBAND_PERIPH(addr, bit);   // Read bit (returns 0 or 1)
```

4 Clock Enable via Bit-Banding

```
#define RCC_APB2ENR_ADDR   (RCC_BASE + 0x18U)

/* RCC_APB2ENR bit positions */
#define RCC_IOPAEN_BIT     2U
#define RCC_IOPBEN_BIT     3U
#define RCC_IOPCEN_BIT     4U
#define RCC_IOPDEN_BIT     5U

static inline void gpio_clock_enable(gpio_port_t port) {
    BITBAND_PERIPH(RCC_APB2ENR_ADDR, _gpio_rcc_bit(port)) = 1U;
}
```

For GPIOC (bit 4 in RCC_APB2ENR at 0x40021018):

$$\text{alias} = 0\text{x}42000000 + (0\text{x}40021018 - 0\text{x}40000000) \times 32 + 4 \times 4$$

$$= 0\text{x}42000000 + 0\text{x}21018 \times 32 + 16$$

$$= 0\text{x}42000000 + 0\text{x}420300 + 0\text{x}10$$

$$= 0\text{x}42420310$$

Writing 1 to address 0x42420310 atomically sets bit 4 in RCC_APB2ENR, enabling the GPIOC clock.

5 Pin Configuration

Pin configuration involves a 4-bit field (MODE[1:0] + CNF[1:0]) and cannot be done with a single bit-band write. This function uses the traditional read-modify-write pattern:

```
static inline void gpio_pin_config(gpio_port_t port, unsigned int pin,
                                   gpio_mode_t mode, gpio_cnf_t cnf) {
    unsigned int base = _gpio_base(port);
    unsigned int offset = (pin < 8U) ? GPIO_CRL_OFF : GPIO_CRH_OFF;
    unsigned int pos = (pin % 8U) * 4U;
    volatile unsigned int *cr = (volatile unsigned int *)(base + offset);

    *cr &= ~(0xFU << pos);
    *cr |= (((unsigned int)cnf << 2U) | (unsigned int)mode) << pos;
}
```

This is a deliberate design choice: bit-banding is used where it provides a real benefit (single-bit operations), while multi-bit fields still require the mask-and-set approach.

6 Pin Set, Clear, Toggle, and Read

These operations target single bits in ODR or IDR, making them ideal for bit-banding:

```
/* Set pin HIGH: write 1 to ODR bit alias */
static inline void gpio_pin_set(gpio_port_t port, unsigned int pin) {
    BITBAND_PERIPH(_gpio_base(port) + GPIO_ODR_OFF, pin) = 1U;
}

/* Clear pin LOW: write 0 to ODR bit alias */
static inline void gpio_pin_clear(gpio_port_t port, unsigned int pin) {
    BITBAND_PERIPH(_gpio_base(port) + GPIO_ODR_OFF, pin) = 0U;
}

/* Toggle: read ODR bit via alias, XOR, write back */
static inline void gpio_pin_toggle(gpio_port_t port, unsigned int pin) {
    volatile unsigned int *alias =
        (volatile unsigned int *)(PERIPH_BB_BASE +
            ((_gpio_base(port) + GPIO_ODR_OFF - PERIPH_BASE) * 32U) +
            (pin * 4U));
    *alias ^= 1U;
}

/* Read pin: returns 0 or 1 from IDR bit alias */
static inline unsigned int gpio_pin_read(gpio_port_t port, unsigned int pin) {
    return BITBAND_PERIPH(_gpio_base(port) + GPIO_IDR_OFF, pin);
}
```

Note that gpio_pin_toggle still requires a read-XOR-write on the alias word, but this operates on a single word containing only 0 or 1, which is simpler than masking a 32-bit register. For truly atomic set/clear without any read, use gpio_pin_set and gpio_pin_clear directly.

Part III
Application: Blinking PC13

7 The main.c Source

```c
/* main.c */
#include "gpio_bitband.h"

int main(void) {

    // Enable GPIOC clock via bit-banding (atomic)
    gpio_clock_enable(GPIO_PORT_C);

    // Configure PC13 as push-pull output, 50 MHz
    gpio_pin_config(GPIO_PORT_C, 13, GPIO_MODE_OUTPUT_50MHZ, GPIO_CNF_OUTPUT_PP);

    // Blink loop using bit-band toggle
    while (1) {
        gpio_pin_toggle(GPIO_PORT_C, 13);
        for (volatile int i = 0; i < 100000; i++);
    }

    return 0;
}
```

Compare this with the original register-level version:

```c
/* Original: raw register access */
#define RCC_APB2ENR   (*(volatile unsigned int *)0x40021018)
#define GPIOC_CRH     (*(volatile unsigned int *)0x40011004)
#define GPIOC_ODR     (*(volatile unsigned int *)0x4001100C)

int main(void) {
    RCC_APB2ENR |= (1 << 4);
    GPIOC_CRH &= ~(0xF << 20);
    GPIOC_CRH |= (0x3 << 20);
    while(1) {
        GPIOC_ODR ^= (1 << 13);
        for(volatile int i = 0; i < 100000; i++);
    }
    return 0;
}
```

The HAL version is more readable, port-independent, and the clock enable is genuinely atomic thanks to bit-banding.

8 Startup and Linker

The startup file and linker script are unchanged from the base project. The startup initializes the stack pointer and calls main():

```
.syntax unified
.cpu cortex-m3
.fpu softvfp
.thumb

.global vtable
.global reset_handler

.section .isr_vector
.align 2
vtable:
    .word _estack
    .word reset_handler + 1

.section .text
.align 2
reset_handler:
    ldr r0, =_estack
    mov sp, r0
    bl main
    b .
    .ltorg
```

Listing 1: startup.s

The linker script defines Flash and RAM regions and computes the stack top:

```
MEMORY
{
    FLASH ( rx )      : ORIGIN = 0x08000000, LENGTH = 64K
    RAM ( rxw )       : ORIGIN = 0x20000000, LENGTH = 20K
}

_estack = ORIGIN(RAM) + LENGTH(RAM);

SECTIONS
{
    .text : { *(.isr_vector) *(.text) *(.text*) *(.rodata) *(.rodata*) } > FLASH
    .data : { *(.data) *(.data*) } > RAM AT> FLASH
    .bss  : { *(.bss) *(.bss*) *(COMMON) } > RAM
}
```

9 Makefile

The HAL is header-only, so the Makefile requires no additional source files:

```
# Toolchain
CC = arm-none-eabi-gcc
OBJCOPY = arm-none-eabi-objcopy
SIZE = arm-none-eabi-size
GDB = arm-none-eabi-gdb

TARGET = program
ASM_SOURCES = startup.s
C_SOURCES = main.c
OBJECTS = $(ASM_SOURCES:.s=.o) $(C_SOURCES:.c=.o)

CFLAGS = -Og -g3 -mthumb -mcpu=cortex-m3 -Wall --specs=nosys.specs -nostdlib
LDFLAGS = -T linker.inv -Wl,-Map=$(TARGET).map

all: $(TARGET).elf $(TARGET).bin

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
%.o: %.s
    $(CC) $(CFLAGS) -c $< -o $@
$(TARGET).elf: $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) $(LDFLAGS) -o $@
    $(SIZE) $@
$(TARGET).bin: $(TARGET).elf
    $(OBJCOPY) -O binary $< $@

flash: $(TARGET).elf
    openocd -c "set CPUTAPID 0x1ba01477" -f interface/stlink.cfg -f target/stm32f1x.cfg
        -c "program $(TARGET).elf verify reset exit"

clean:
    rm -f $(TARGET).elf $(TARGET).bin $(TARGET).map $(OBJECTS)

.PHONY: all flash clean
```

Part IV
Bit-Banding in Depth

10 Traditional Register Access vs Bit-Banding

To understand the concrete improvement, consider what happens at the instruction level when setting a single GPIO pin.

10.1 Traditional: Read-Modify-Write on ODR

```
// C code
GPIOC_ODR |= (1 << 13);
```

Compiles to approximately:

```
    ldr  r0, =0x4001100C      ; Load ODR address
    ldr  r1, [r0]             ; READ current ODR value
    orr  r1, r1, #(1 << 13) ; MODIFY: set bit 13
    str  r1, [r0]             ; WRITE back
```

Four instructions, two bus transactions (load + store). An interrupt between the ldr and str could corrupt the register.

10.2 Bit-Band: Single Store

```
// C code
BITBAND_PERIPH(0x4001100C, 13) = 1;
```

Compiles to approximately:

```
    ldr  r0, =0x42220034      ; Load alias address for ODR bit 13
    movs r1, #1               ; Value to write
    str  r1, [r0]             ; Single STORE - hardware sets bit 13
```

Three instructions, one bus write. The store is atomic — no interrupt can interfere.

11 When to Use Bit-Banding

- Single-bit operations on peripheral registers (clock enable, pin set/clear, flag checks)
- Interrupt-safe bit manipulation without disabling interrupts
- SRAM flags shared between main code and ISR handlers
- Any scenario where read-modify-write races are a concern

12 Limitations

- Only available on Cortex-M3 and Cortex-M4 (not M0, not M7 in most implementations)
- Only the first 1 MB of each region is bit-addressable
- Multi-bit fields (like GPIO mode configuration) still require read-modify-write
- The alias region consumes 32 MB of address space (no physical memory cost)
- Toggle still needs a read-XOR-write on the alias word