

In one line: volatile is a promise about the **compiler**: every read and write in the source becomes a real load/store, never cached in a register or removed. It gives **no atomicity**, **no ordering** of ordinary memory and **no CPU barrier**. State shared by a task and an ISR or another core needs **_Atomic** or a **critical section**.

What volatile does

- Each access is *observable behaviour*: emitted, never merged or dropped, not reordered *against other volatile accesses*.
- The ISR-flag bug: without it `while(!flag){}` is read *once* (nothing in the loop writes it) → `if(!flag) for(;;);`.
- For: **memory-mapped registers** (a read may pop a FIFO or clear a status bit), **vars written by an ISR**, volatile sig_atomic_t signal flags, locals live across longjmp.
- volatile uint32_t *r = pointer to volatile (registers);
uint32_t *volatile p = volatile pointer (rarely meant).

What volatile does NOT do

- Atomicity:** v++ = LOAD, ADD, STORE; an ISR fits between any two. 64-bit on a 32-bit CPU = 2 loads → **born read**.
- Order plain memory:** a normal `buf[i]=x` may move across a volatile store to ready.
- CPU barrier:** none emitted — another core may see the writes in a different order.
- Cache bypass:** it stops *register* caching, not the data cache (DMA → invalidate / non-cacheable RAM).
- Thread safety:** a data race (unsynchronised, one side writes) is UB in C11, volatile or not.

C11 atomics — <stdatomic.h>

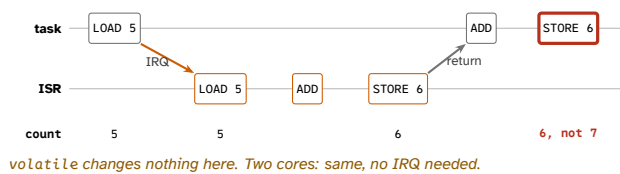
- _Atomic uint32_t n;: plain n++, n+=2, n=0 are atomic, seq_cst. API: atomic_load/store, atomic_fetch_add, atomic_exchange, atomic_compare_exchange_strong/weak, each with an _explicit(..., order) form.
- Hardware: ARMv7-M LDREX/STREX loop · x86 LOCK · ESP32/S3 Xtensa S32C1I (CAS). 64-bit on a 32-bit MCU may not be lock-free (atomic_is_lock_free).
- asm volatile(...) :: "memory" = *compiler-only* barrier; __DMB() / atomic_thread_fence = CPU barrier.

order	guarantees
relaxed	atomic only, no ordering — counters, stats
acquire	load: later accesses can't move <i>above</i> it
release	store: earlier accesses can't move <i>below</i> it; an acquire that reads it sees them all
acq_rel	both — for RMW (fetch_add, CAS)
seq_cst	default: acq/rel + one global order; costliest

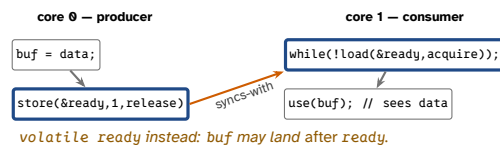
Example — the ISR counter, right

```
static _Atomic uint32_t events; // ISR <-> task
static TaskHandle_t worker;
void IRAM_ATTR gpio_isr(void *arg) {
    atomic_fetch_add_explicit(&events, 1,
                             memory_order_relaxed);
    BaseType_t woken = pdFALSE;
    vTaskNotifyGiveFromISR(worker, &woken); // wake
    portYIELD_FROM_ISR(woken);
}
void worker_task(void *arg) {
    for (;;) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY); // sleep
        handle(atomic_exchange(&events, 0)); // read+reset=1 op
    }
}
```

Picture — the lost update



Picture — publish with release / acquire



Critical sections, FreeRTOS, ESP32

- Use one for **several fields**, a 64-bit value, or check-then-act. Single core: taskENTER_CRITICAL() masks interrupts. A few instructions only; never block inside.
- ESP32 is dual-core.** Masking interrupts stops only *this* core, so ESP-IDF adds a spinlock: portENTER_CRITICAL(&mux) (_ISR variant in ISRs), portMUX_TYPE mux = portMUX_INITIALIZER_UNLOCKED. Unpinned tasks (tskNO_AFFINITY) run truly in parallel.
- A mutex **cannot be taken in an ISR** (may block). A semaphore / notification wakes a task — it does not make access atomic.

Interview traps

- Your Q1:** name the missing volatile *first* (correctness), then the busy-wait (efficiency).
- Your Q2:** volatile counter++ “looks fine” — 3 instructions; the task's check-then-reset is a 2nd race.
- Your Q16:** “volatile skips the cache” — no, the *register* copy; CPU caches are untouched.
- Your Q29:** semaphore ≠ atomicity; handle(c); c=0; drops ISR increments → atomic_exchange.
- No “Mars rover + volatile” story: Pathfinder 1997 was priority inversion.

Remember

volatile: the compiler won't cache it. **_Atomic:** nobody can split it. **Release publishes, acquire subscribes.** Register → volatile · one word → atomic · several → critical section · wait → FromISR signal.

Likely questions

- Is volatile thread-safe? — No: no atomicity, no ordering.
- _Atomic plus volatile? — atomic suffices for sharing; volatile is for MMIO.
- Aligned 32-bit read on ESP32 atomic? — in hardware yes; in C still a race → atomic_load_explicit(relaxed), same instruction.
- Why portMUX? — interrupts-off does not stop the other core.