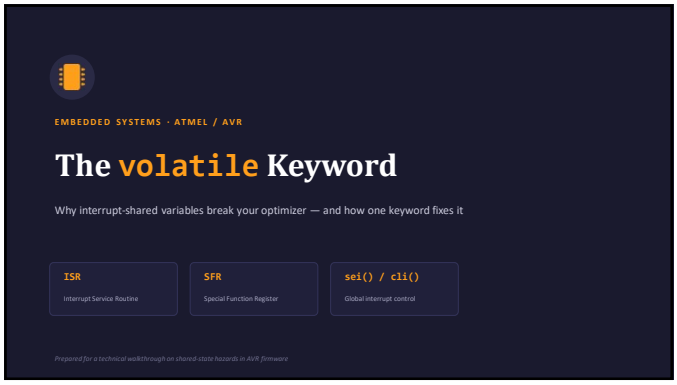
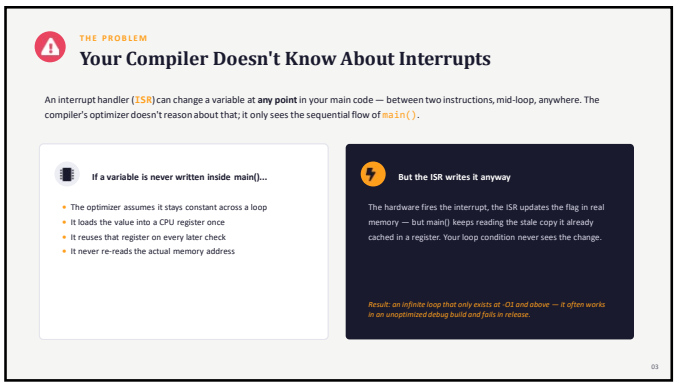

EMBEDDED SYSTEMS · ATMEL / AVR
The volatile Keyword
Why interrupt-shared variables break your optimizer — and how one keyword fixes it
ISR
Interrupt Service Routine
SFR
Special Function Register
sei() / cli()
Global interrupt control
Prepared for a technical walkthrough on shared-state hazards in AVR firmware

1


OVERVIEW
What We'll Cover
01
The Problem
How compiler optimization silently breaks ISR-to-main communication
02
AVR Example
A polling loop that never exits — and the one-word fix
03
What volatile Really Does
Forcing every read and write back to real memory
04
When You Need It
Flags, buffers, and hardware registers shared with an ISR
05
Beyond volatile
Atomicity, multi-byte variables, and ATOMIC_BLOCK
02

2


THE PROBLEM
Your Compiler Doesn't Know About Interrupts
An interrupt handler (ISR) can change a variable at any point in your main code — between two instructions, mid-loop, anywhere. The compiler's optimizer doesn't reason about that; it only sees the sequential flow of main().
If a variable is never written inside main()...
The optimizer assumes it stays constant across a loop
It loads the value into a CPU register once
It reuses that register on every later check
It never re-reads the actual memory address
But the ISR writes it anyway
The hardware fires the interrupt, the ISR updates the flag in real memory — but main() keeps reading the stale copy it already cached in a register. Your loop condition never sees the change.
Result: an infinite loop that only exists at -O1 and above — it often works in an unoptimized debug build and fails in release.
03

3

AVR EXAMPLE

Without volatile — A Loop That Never Ends

BUG Timer ISR sets a flag; main() polls it

main.c — compiled with -O2

```
uint8_t flag = 0;

ISR(TIMER1_OVF_vect) {
    flag = 1;   // hardware sets this asynchronously
}

int main(void) {
    sei();                  // enable global interrupts
    while (!flag) { }       // ⚠ optimized to while(true)
    // → compiler loaded 'flag' into a register once
    //   and never re-reads memory here
    run_task();
}
```

04

4

THE FIX

What volatile Actually Does

```
volatile uint8_t flag = 0;
```

One keyword, three guarantees the compiler must now honor:

Real reads

Every access re-reads the value from its actual memory address — never from a cached register.

No reordering

The compiler can't reorder, merge, or eliminate reads/writes to this variable around it.

No caching

It may change between any two statements — from an ISR, a signal handler, or hardware.

05

5

AVR EXAMPLE

Before vs. After

BEFORE

hangs at -O2

```
uint8_t flag = 0;

ISR(TIMER1_OVF_vect) {
  flag = 1;
}

int main(void) {
  sei();
  while (!flag) { }
  run_task();
}
```

AFTER

exits correctly

```
volatile uint8_t flag = 0;

ISR(TIMER1_OVF_vect) {
  flag = 1;
}

int main(void) {
  sei();
  while (!flag) { }  // re-reads memory
  run_task();
}
```

06

6

USAGE GUIDE

When You Need volatile on AVR

Flags shared with an ISR	Any variable an ISR sets and main() (or another ISR) reads — status bits, "data ready" flags, error codes.
Circular buffers / ring indices	Head/tail indices of a buffer filled by a UART or SPI ISR and drained in main().
Memory-mapped hardware registers	SFRs like PORTB, TCCR1A, UDR0. avr-libc's <avr/io.h> already declares these volatile for you.
Values touched by more than one ISR	E.g. a millisecond counter incremented by a timer ISR and read inside another handler.

07

7

BEYOND VOLATILE

volatile Doesn't Mean Atomic

AVR is an **8-bit** architecture. Reading or writing a 16-bit or 32-bit `volatile` variable takes multiple instructions — an ISR can fire **between** them and leave main() with a torn, half-updated value.

the hazard — 16-bit counter

```
volatile uint16_t ticks = 0;

ISR(TIMER0_OVF_vect) {
  ticks++;  // 2 instructions: lo, hi
}

// in main():
uint16_t t = ticks;  // ⚠ 2 reads
// ISR can fire between the
// low-byte and high-byte read
// → t may never have existed
```

the fix — atomic block

```
#include <util/atomic.h>

uint16_t t;
ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
  t = ticks;  // interrupts held off
}
// interrupt state auto-restored

// manual equivalent:
cli(); t = ticks; sei();
```

08

8

SUMMARY

Best Practices

DO

- Mark every variable shared between an ISR and main() volatile
- Wrap multi-byte volatile access in ATOMIC_BLOCK (or cli()/sei())
- Keep ISRs short — set a flag or copy data, do the real work in main()

DON'T

- Don't mark everything volatile — it blocks optimization on data that doesn't need it
- Don't assume volatile makes multi-byte access atomic on 8-bit AVR
- Don't do slow work (printf, long math) inside an ISR

09

9