What Volatile Means (and Doesn't Mean)

BEN SAKS

Cppcon
The C++ Conference

2024 September 15 - 20

Saks & Associates

Saks & Associates
393 Leander Dr.
Springfield, OH 45504-4906 USA
+1-412-521-4117 (voice)
service@saksandassociates.com
saksandassociates.com

2

Introduction

- The `volatile` qualifier is a vital tool for preventing compilers from performing certain harmful optimizations.
- Unfortunately, many C++ programmers aren't clear on exactly what protections `volatile` provides.
- As such, many programmers apply the `volatile` qualifier incorrectly.
- A misapplied `volatile` might:
  - prevent optimizations unnecessarily, or worse
  - fail to provide the expected protection, leading to subtle run-time bugs.

7

Introduction

- This session examines:
  - Why `volatile` is necessary
  - How to place `volatile` in object declarations
  - What protections `volatile` does and doesn't provide
  - Workarounds for compiler issues regarding `volatile`

8

Why `volatile` is Necessary

- Many device drivers contain code that clearly illustrates the need for `volatile`.
- In this section, we'll look at code from a simple UART (serial port) driver for the ARM Evaluator-7T (E7T)…

9

Device Registers

- A ***device driver*** is a software subsystem that controls an "external" device attached to a computer.
- Here, "external" means "outside the CPU"…
- …even if it's on the same chip.
- CPUs typically communicate with external devices via device registers.
- A ***device register*** is circuitry that provides an interface to a device…

10

Device Registers

- A single device may use different registers for different functions:
  - A ***control*** register:
    - configures the device, or
    - initiates an operation.
  - A ***status*** register:
    - provides information about the device's state.
  - A ***transmit*** register:
    - sends a data value to the device.
  - A ***receive*** register:
    - receives a data value from the device.

11

Sample Device Registers

- Many hardware devices have multiple registers, often located at contiguous addresses.
- For example, the E7T has two serial ports (UARTs), each with the same layout:

Offset	Register	Description
0x00 (0)	ULCON	line ***control***
0x04 (4)	UCON	***control***
0x08 (8)	USTAT	***status***
0x0C (12)	UTXBUF	***transmit*** buffer
0x10 (16)	URXBUF	***receive*** buffer
0x14 (20)	UBRDIV	baud rate divisor (***control***)

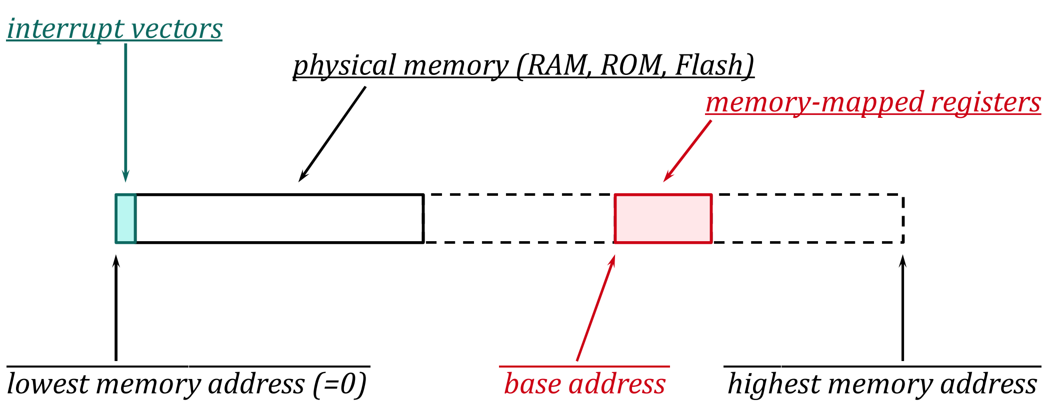
12

Memory-Mapped Registers

- Most modern computer architectures use ***memory-mapped addressing***.
- That is, a ***memory-mapped [device] register***:
  - connects to the CPU's (address and data) bus structure, and
  - responds to bus signals almost as if it were ordinary memory.
- In short, memory-mapped addressing disguises the device registers to be addressable like "ordinary" memory...

13

"Typical" Address Space



- On the E7T, the memory-mapped registers for UART 0 are located at 0x3FFD000.

14

UART Output

- To output a character via the UART, you must access both the USTAT and UTXBUF registers.
- The TBE bit (Transmit Buffer Empty) in USTAT is set to 1 when UTXBUF is ready for use.
- You shouldn't store a character into UTXBUF until the TBE bit is set to 1.
- Storing a character into UTXBUF initiates output to the port and clears the TBE bit in USTAT.
- The hardware automatically sets the TBE bit back to 1 when it completes the output operation.

15

UART Output

- There are several ways that the program could create C++ objects to communicate with UART 0's device registers.
- This example code accesses the USTAT and UTXBUF registers for UART 0 through the following references:

```
std::uint32_t &USTAT0 =
    *reinterpret_cast<special_register *>(0x03FFD008);
std::uint32_t &UTXBUF0 =
    *reinterpret_cast<special_register *>(0x03FFD00C);
```

- `volatile` is intentionally omitted here to demonstrate why it's necessary.

16

UART Output

- This code waits for the UTXBUF to become available, then writes to it:

```
std::uint32_t &USTAT0 =
    *reinterpret_cast<special_register *>(0x03FFD008);
std::uint32_t &UTXBUF0 =
    *reinterpret_cast<special_register *>(0x03FFD00C);
~~~
while ((USTAT0 & TBE) == 0) {
}
UTXBUF0 = c;
```

- This looks like it should work.
- However, an optimizer might cause this code to fail.

17

Overly-Aggressive Optimization

- To see why optimization might be a problem, consider this variation on the code, which sends a '\r' and then a '\n' to UART 0:

```
while ((USTAT0 & TBE) == 0) {
}
UTXBUF0 = '\r';
while ((USTAT0 & TBE) == 0) {
}
UTXBUF0 = '\n';
```

18

Overly-Aggressive Optimization

- Although they're mapped to memory locations, device registers aren't ordinary memory.
- Device register accesses (reads and writes) may have side effects. For example:
  - Writing to a control register may initiate an operation.
  - Reading from a receive buffer may set or clear bits in a status register.
- Compiler optimizations might change the number of register accesses.
  - Eliminating an access eliminates its side effects.
  - Eliminating those side effects might cause device drivers to fail.

19

Overly-Aggressive Optimization

- Unfortunately, to the compiler, `USTAT0` looks like an ordinary object.
  - Its state should change only when the program acts on it.
- Thus, the compiler's optimizer might conclude that `USTAT0`'s value never changes.

The compiler thinks this condition never changes

```
while ((USTAT0 & TBE) == 0) {
}
```

- The `TBE` bit in `USTAT0` is either always 1 or it's always 0.

20

Overly-Aggressive Optimization

- If the condition never changes, there's no need to test the loop condition repeatedly.
- The program can simply test the condition once:

```
if ((USTAT0 & TBE) == 0) {
    for (;;) {
    }
    ~~~
}
```

the program never leaves this loop

- The program either:
  - loops forever, or
  - skips the loop entirely.

21

Overly-Aggressive Optimization

- After this optimization, the code looks like:

```
if ((USTAT0 & TBE) == 0) {
    for (;;) {
    }
}
UTXBUF0 = '\r';
if ((USTAT0 & TBE) == 0) {
    for (;;) {
    }
}
UTXBUF0 = '\n';
```

- Again, the compiler deduces that the TBE bit never changes:
  - If the TBE bit is always off, the code enters the first loop and never escapes.
  - If the TBE bit is always on, the code bypasses both loops.
- In either case, execution never reaches the second loop…

22

Overly-Aggressive Optimization

- The optimizer can eliminate the second if-statement entirely:

```
if ((USTAT0 & TBE) == 0) {
    for (;;) {
    }
}
UTXBUF0 = '\r';
UTXBUF0 = '\n';
```

second if-statement removed

- It then becomes "evident" that the first assignment has no effect…

23

Overly-Aggressive Optimization

- The first assignment stores a value into `UTXBUF0` that's immediately overwritten by the second assignment:

```
if ((USTAT0 & TBE) == 0) {
    for (;;) {
    }
}
UTXBUF0 = '\r';
UTXBUF0 = '\n';
```

this *overwrites* ***this***

- The optimizer can eliminate the first assignment…

24

Overly-Aggressive Optimization

- The optimized code looks like:

```
if ((USTAT0 & TBE) == 0) {
    for (;;) {
    }
}
UTXBUF0 = '\n';
```

- It does the wrong thing, but more efficiently!

25

Overly-Aggressive Optimization

- Many toolchains allow you to disable all optimizations for a region of code, which is one way to avoid this problem.
- Disabling *all* optimizations for a region of code can be overkill.
  - You'll probably also lose some beneficial optimizations that you'd prefer to keep.
- `volatile` can provide a more precise solution…

26

The volatile Qualifier

- To prevent these unwanted optimizations, declare `USTAT0` and `UTXBUF0` as "reference to volatile", as in:

```
std::uint32_t volatile &USTAT0 =
    *reinterpret_cast<std::uint32_t *>(0x03FFD008);
std::uint32_t volatile &UTXBUF0 =
    *reinterpret_cast<std::uint32_t *>(0x03FFD00C);
```

27

The volatile Qualifier

- Conceptually, `volatile` informs the compiler that the object may change state even though the program didn't change it.
- More mechanically, the compiler must assume that any access to a volatile object (reading or writing) may have a side effect.
- Thus, the compiler mustn't "optimize away" an access to a volatile object, even when it seems safe to do so.

28

CV-Qualifiers

- In C++, the ***cv-qualifiers*** `const` and `volatile` are closely related.
- Almost all of the rules regarding the placement and meaning of `const` also apply to `volatile`.
- Understanding cv-qualifiers starts with understanding the structure of declarations…

29

The Structure of Declarations

Insight: Every object and function declaration has two main parts:
- *a sequence of one or more **declaration specifiers***
- *a **declarator** (or a sequence thereof, separated by commas)*

- For example:

```
static unsigned long int *x[N]
```

declaration specifiers *declarator*

- The name declared in a declarator is the ***declarator-id***.

30

Declaration Specifiers and Declarators

- A ***declaration specifier*** can be:
  - a ***type specifier***:
    - a keyword such as `int`, `unsigned`, `long`, or `double`
    - a user-defined type, such as `ostream` or `string`
  - a ***non-type specifier***:
    - a keyword such as `extern`, `static`, `inline`, or `typedef`

- A ***declarator*** is the name being declared, possibly surrounded by operators:
  - `*` means "pointer"
  - `&` means "reference"
  - `[]` mean "array"
  - `()` mean "function"

31

Type vs. Non-type Specifiers

Insight: Type specifiers modify other type specifiers.

Insight: Non-type specifiers apply directly to the declarator-id.

```
static unsigned long int *x[N]
```

- Here, `unsigned`, `long`, and `int` are type specifiers.
  - They form the type to which the pointers in array `x` point.

- `static` is a non-type specifier that applies directly to `x`.

32

volatile is a Type-Specifier

Insight: The order of the declaration specifiers doesn't matter to the compiler.

- These two declarations mean the same thing:

```
unsigned long ul;          // unsigned long
long unsigned ul;          // same thing
```

- So do these three:

```
const unsigned long cul;    // const unsigned long
long unsigned const cul;    // same thing
unsigned const long cul;    // same, and we're not amused
```

33

volatile is a Type-Specifier

`volatile` is a type specifier, much like `long` or `unsigned`.

*`volatile` modifies the other **type** specifier(s) in the same declaration.*

right interpretation	***wrong interpretation***
`volatile int *v[N]`	`volatile int *v[N]`
`int volatile *v[N]`	`int volatile *v[N]`

- v is an object of type "array of N pointers to volatile int".

34

Placing `volatile` in Declarations

Insight: `const` and `volatile` are the only symbols (in C++) that can appear either as declaration specifiers or in declarators.

- In both of these, `volatile` is a type specifier:

```
             |
volatile int | *v[N]    // volatile modifies int
int volatile | *v[N]    // same thing
             |
```

- Here, `volatile` appears in the declarator:

```
    |
int | *volatile v[N]    // volatile modifies the * (the pointer)
    |
```

35

Placing `volatile` in Declarations

- Although `volatile` *can* appear as in a declarator, it rarely does.
- A hardware register typically holds an integer value, a collection of bitmasks, or maybe character data.
  - It probably doesn't hold a memory address.
- However, there's a simple way to ensure that you're placing `const` (or `volatile`) where you want it in a declaration…

36

Placing `volatile` in Declarations

- First, write the declaration as it would be without `const` or `volatile`.
- Then…

✓ *Place `const` or `volatile` to the immediate right of the type specifier or operator that you want it to modify.*

- This is the "East Const" (or "East Volatile") style…

37

Placing `volatile` in Declarations

- For example, suppose we want `x` to be:
  - "array of N ***const pointers*** to ***volatile uint32_t***".
- Start by writing the declaration for:
  - "array of N ***pointers*** to ***uint32_t***"…

```
uint32_t *x[N];
```

38

Placing volatile in Declarations

- Here it is again, with room for the cv-qualifiers:

```
uint32_t          *      x[N];
```

- Next, add const to the immediate right of the *:

```
uint32_t          *const x[N];
```

- Finally, add volatile to the immediate right of uint32_t:

```
uint32_t volatile *const x[N];
```

- Bob's your uncle!
  - x is an "array of N const pointers to volatile uint32_t".

39

Ordering of Volatile Operations

- Again, the compiler must assume that any access to a volatile object (reading or writing) may have a side effect.
- Moreover, the compiler must assume that the side effects from accessing volatile objects could be related.
- Thus, the compiler can't change the ordering of two volatile accesses, even if the accesses are for different objects.
- Consider the code for transmitting a character over a UART again…

40

Ordering of Volatile Operations

- The TBE bit in USTAT0 determines if it's currently safe to write to UTXBUF0.
- However, the declarations for USTAT0 and UTXBUF0 don't establish any obvious connection between them:

```
std::uint32_t volatile &USTAT0 =
    reinterpret_cast<std::uint32_t *>(0x03FFD008);
std::uint32_t volatile &UTXBUF0 =
    reinterpret_cast<std::uint32_t *>(0x03FFD00C);
```

- The compiler's only indication that USTAT0 and UTXBUF0 are somehow related is that they're both declared volatile.

41

Ordering of Volatile Operations

- Thus, the compiler must conclude that it can't reorder this code simply from the fact that USTAT0 and UTXBUF0 are both volatile:

```
while ((USTAT0 & TBE) == 0) {
}
UTXBUF0 = '\r';
```

42

Ordering of Volatile and Non-Volatile Operations

- `volatile` disables optimizations for a specific object rather than for a region of code.
- The compiler can still optimize accesses to non-volatile objects in the surrounding code.
- Notably, the compiler can also *reorder* accesses to non-volatile objects with respect to an access to a volatile object.
- As the following example illustrates, this is one reason that `volatile` is not a reliable tool for managing inter-thread communication…

43

Ordering of Volatile and Non-Volatile Operations

- This slightly-modified code from Eide and Regehr [2008] appears to clear the buffer, then set the volatile flag `buffer_ready` to notify another thread:

```
bool volatile buffer_ready;
char buffer[BUF_SIZE];

void buffer_init() {

    for (int i = 0; i < BUF_SIZE; ++i) {
        buffer[i] = 0;
    }
    buffer_ready = true;    // looks like a reliable signal, but...
}
```

44

Ordering of Volatile and Non-Volatile Operations

- Because the loop contains no accesses to volatile objects or other side effects, the compiler can move the assignment to `buffer_ready` around it:

```
bool volatile buffer_ready;
char buffer[BUF_SIZE];

void buffer_init() {
    buffer_ready = true;    // uh-oh, signal is too early
    for (int i = 0; i < BUF_SIZE; ++i) {
        buffer[i] = 0;
    }

}
```

- Now the flag is set before the buffer is ready.

45

Ordering of Volatile and Non-Volatile Operations

- This code *would* work as intended if `buffer` were also volatile, but then the compiler couldn't optimize any use of `buffer`:

```
bool volatile buffer_ready;
char volatile buffer[BUF_SIZE];

void buffer_init() {

    for (int i = 0; i < BUF_SIZE; ++i) {
        buffer[i] = 0;
    }
    buffer_ready = true;
}
```

46

Multithreading — The Wrong Tool for the Job

- The Standard Library and other threading libraries provide synchronization tools designed for inter-thread communication, such as:
  - mutexes
  - semaphores
  - condition variables

✓ *For inter-thread communication, use synchronization tools such as mutexes and semaphores.*

✓ *Don't use volatile objects for inter-thread communication.*

47

Atomicity Not Guaranteed

- Accesses to volatile objects are ***not guaranteed*** to be atomic.
- In other words, an operation on a volatile object can potentially result in a ***data race***.
- As an example, consider the following code…

48

Atomicity Not Guaranteed

- Suppose that this code is executing on thread A on a platform where a `double` is not naturally atomic:

```
double volatile v = 0.0;

void modify_value() {
    v = 8.67;           // #1
    v = 53.09;          // #2
}
```

- Another thread B could potentially access `v` while thread A is executing the code marked #2.
- If it does, B might observe `v` as having a value that is neither `8.67` nor `53.09`.

49

Atomicity Not Guaranteed

- Of particular note, increment, decrement, and compound assignment expressions such as these are not guaranteed to be atomic for volatile objects:

```
double volatile v = 0.0;
v++;
--v;
v += 3;
v <<= 2;
```

50

Miscompiled `volatiles`?

- Eide and Regehr [2008] tested 13 versions of 5 distinct C compilers to see if they consistently generated correct code for accessing volatile objects.
- At the time, all 13 generated incorrect code for at least one usage of `volatile` in their randomly-generated test programs.
- That is, at some optimization level, each compiler optimized an access to a volatile object in a way that it shouldn't have.
- In a safety-critical system, an incorrect optimization could have serious consequences.

51

Miscompiled `volatiles`?

- While the study was some time ago and resulted in several fixes, we should probably expect new `volatile` bugs to appear occasionally.
- Compiler writers are constantly working to improve their optimizers, which are naturally in conflict with `volatile`.
- If you suspect that your compiler is mishandling a `volatile` object, you have a few options…

52

Miscompiled volatiles — Option #1

- Option #1: You could turn off optimizations for the affected code.
- For example, GCC provides an attribute that you can use to change the optimization level for a function:

```
void [[gnu::optimize("O0")]] void foo() f() { ~~~ }
```

- For the GNU compilers, `O0` (optimization level 0) does the least optimization.

53

Miscompiled volatiles — Option #1

- Here's another way to disable optimizations for a function in GCC using #pragmas:

```
#pragma GCC push_options
#pragma GCC optimize ("O0")

void f() { ~~~ }

#pragma GCC pop_options
```

54

Miscompiled `volatiles` — Option #2

- Option #2: You could try using a different version of your compiler (or a different compiler altogether).
- Eide and Regehr found significant differences in `volatile` bugs between versions of GCC.
- Caution: Eide and Regehr found that a more recent version doesn't necessarily have fewer volatile bugs.
- For example, they found more volatile bugs in GCC 4.2.4 than in GCC 4.0.4.

55

Miscompiled `volatiles` — Option #3

- Option #3: Use a workaround suggested by Eide and Regehr.
- Eide and Regehr found that accessing the volatile object through ***non-inline*** functions corrected ~96% of the bugs that they observed.
- Eide and Regehr created two functions for each type of volatile object that they tested — one used for reading and one used for writing.

56

Working Around Miscompiled volatiles

- This code reads from a “volatile int” in the usual way:

```
int volatile v_int;
~~~
int value = v_int;
```

57

Working Around Miscompiled volatiles

- Here’s a version that reads the int using a C++ version of their workaround technique:

```
int vol_read_int(int volatile &vp) {
    return vp;
}

int volatile v_int;
~~~
int value = vol_read_int(v_int);
```

58

Working Around Miscompiled volatiles

- This code writes to a “volatile int” in the usual way:

```
int volatile v_int;
~~~
v_int = 256;
```

59

Working Around Miscompiled volatiles

- Here’s a version that writes to the int using a C++ version of their workaround technique:

```
int volatile &vol_id_int(int volatile &v) {
    return v;
}

int volatile v_int;
~~~
vol_id_int(v_int) = 256;
```

60

Working Around Miscompiled `volatile`s

- Why do these workaround functions help?
- When compiling a call to a non-inline function, the compiler doesn't know what the function will do or what side effects it might have.
- To ensure correctness, the compiler must be careful to call the function exactly as many times as the user expects.
- This closely mirrors the intended behavior for accessing a volatile object.
- Thus, the workaround serves as a kind of "redundant backup" for the protections that volatile is supposed to provide.

61

Working Around Miscompiled `volatile`s

- If we apply these workarounds to the UART code from earlier, we get:

```
std::uint32_t vol_read_u32(std::uint32_t volatile &v) {
    return v;
}

std::uint32_t volatile &vol_id_u32(std::uint32_t volatile &v) {
    return v;
}

// continued on the next slide...
```

62

Working Around Miscompiled volatiles

- If we apply these workarounds to the UART code from earlier, we get:

```
//...continued from the previous slide

std::uint32_t volatile &USTAT0 =
    reinterpret_cast<std::uint32_t *>(0x03FFD008);
std::uint32_t volatile &UTXBUF0 =
    reinterpret_cast<std::uint32_t *>(0x03FFD00C);
~~~
while ((vol_read_u32(USTAT0) & TBE) == 0) {
}
vol_id_u32(UTXBUF0) = '\r';
```

63

Cautionary Note Regarding Templates

- I experimented with template versions of these workaround functions using the GNU ARM Embedded Toolchain v10.2.1:

```
template <typename T>
T vol_read(T &v) {
    return v;
}

template <typename T>
T &vol_id_sr(T &v) {
    return v;
}
```

- In my tests, GCC automatically inlined them at `-O1` and above, even without an explicit `inline` qualifier.

64

Cautionary Note Regarding Templates

- In my case, I found that I was able to prevent this automatic inlining by tagging the functions with the `[[gnu::noinline]]` attribute:

```
template <typename T>
[[gnu::noinline]] T vol_read_sr(T &v) {
    return v;
}

template <typename T>
[[gnu::noinline]] T &vol_id_sr(T &v) {
    return v;
}
```

- If you use template versions of these functions, make sure they aren't inlined automatically.

65

Takeaways

- `volatile` tells the compiler that accessing an object may have side effects that mustn't be optimized away.
- The compiler must keep accesses to volatile objects in order, but may reorder accesses to non-volatile objects around them.
- Use synchronization tools (e.g., mutexes and semaphores) rather than volatile objects to manage inter-thread communication.
- Accesses to volatile objects are ***not*** guaranteed to be atomic.

66

Takeaways

- If you find that your compiler is mishandling `volatile`, try these remedies:
  - Disable optimizations for that code.
  - Use a different version of the compiler.
  - Use Eide and Regehr's workaround.
- If you do use Eide and Regehr's workaround, make sure that the functions aren't inlined.

67

Bibliography

- Eide, Eric & Regehr, John. [2008]. *Volatiles are miscompiled, and what to do about it*. Proceedings of the 8th ACM International Conference on Embedded Software, EMSOFT'08. 255-264. 10.1145/1450058.1450093.

68

