ELECTRICALENGINEERING

Using volatile in embedded C development

Asked 1 year ago Active 8 months ago Viewed 8k times

44

18

I have been reading some articles and Stack Exchange answers about using the `volatile` keyword to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

If I am reading from an ADC (let's call the variable `adcValue`), and I am declaring this variable as global, should I use the keyword `volatile` in this case?

1. Without using `volatile` keyword

```
// Includes
#include "adcDriver.h"

// Global variables
uint16_t adcValue;

// Some code
void readFromADC(void)
{
   adcValue = readADC();
}
```

2. Using the `volatile` keyword

```
// Includes
#include "adcDriver.h"

// Global variables
volatile uint16_t adcValue;

// Some code
```

I am asking this question because when debugging, I can see no difference between both approaches although the best practices says that in my case (a global variable that changes directly from the hardware), then using `volatile` is mandatory.

microcontroller c embedded

edited Nov 30 '18 at 19:45
Bence Kaulics
5,930 10 27 53

asked Nov 29 '18 at 12:30
Pryda
1,023 5 20

1 A number of debug environments (certainly gcc) apply no optimisations. A production build normally will (depending on your choices). This can lead to 'interesting' differences between builds. Looking at the linker output map is informative. – Peter Smith Nov 29 '18 at 12:53

22 "in my case (Global variable that changes directly from the hardware)" - Your global variable is *not* changed *by hardware* but only by your C code, of which the compiler is aware. - The hardware register in which the ADC provides it's results, however, *must* be volatile, because the compiler cannot know if/when its value will change (it changes if/when the ADC hardware finishes a conversion.) – JimmyB Nov 29 '18 at 14:18

2 Did you compare the assembler generated by both versions? That should show you what is happening under the hood – Mawg Nov 29 '18 at 14:18

3 @stark: BIOS? On a microcontroller? Memory-mapped I/O space will be non-cacheable (if the architecture even has a data cache in the first place, which is not assured) by design consistency between the caching rules and the memory map. But volatile has nothing to do with memory controller cache. – Ben Voigt Nov 29 '18 at 20:26

1 @Davislor The language standard doesn't need to say anything more in general. A read to a volatile object will perform a real load (even if the compiler recently did one and would usually know what the value is) and a write to such object would perform a real store (even if the same value was read from the object). So in `if(x==1) x=1;` the write may be optimized away for a non volatile `x` and cannot be optimized if `x` is volatile. OTOH if special instructions are needed to access external devices, it's up to you to add those (f.ex. if a memory range needs to be made write through). – curiousguy Dec 1 '18 at 4:05

9 Answers

A definition of `volatile`

87

`volatile` tells the compiler that the variable's value may change without the compiler knowing. Hence the compiler cannot assume the value did not change just because the C program seems not to have changed it.

✓

On the other hand, it means that the variable's value may be required (read) somewhere else the compiler does not know about, hence it must make sure that every assignment to the variable is actually carried out as a write operation.

Use cases

`volatile` is required when

- representing hardware registers (or memory-mapped I/O) as variables - even if the register will never be read, the compiler must not just skip the write operation thinking "Stupid programmer. Tries to store a value in a variable which he/she will never ever read back. He/she won't even notice if we omit the write." Conversly, even if the program never writes a value to the variable, its value may still be changed by hardware.
- sharing variables between execution contexts (e.g. ISRs/main program) (see @kkramo's answer)

Effects of `volatile`

When a variable is declared `volatile` the compiler must make sure that every assignment to it in program code is reflected in an actual write operation, and that every read in program code reads the value from (mmapped) memory.

For non-volatile variables, the compiler assumes it knows if/when the variable's value changes and can optimize code in different ways.

For one, the compiler can reduce the number of reads/writes to memory, by keeping the value in CPU registers.

Example:

```
void uint8_t compute(uint8_t input) {
  uint8_t result = input + 2;
  result = result * 2;
  if ( result > 100 ) {
    result -= 100;
  }
  return result;
}
```

Here, the compiler will probably not even allocate RAM for the `result` variable, and will never store the intermediate values anywhere but in a CPU register.

If `result` was volatile, every occurrence of `result` in the C code would require the compiler to perform an access to RAM (or an I/O port), leading to a lower performance.

Secondly, the compiler may re-order operations on non-volatile variables for performance and/or code size. Simple example:

```
int a = 99;
int b = 1;
int c = 99;
```

could be re-ordered to

```
int a = 99;
int c = 99;
int b = 1;
```

which may save an assembler instruction because the value `99` won't have to be loaded twice.

If `a`, `b` and `c` were volatile the compiler would have to emit instructions which assign the values in the exact order as they are given in the program.

The other classic example is like this:

```
volatile uint8_t signal;

void waitForSignal() {
  while ( signal == 0 ) {
    // Do nothing.
  }
}
```

If, in this case, `signal` were not `volatile`, the compiler would 'think' that `while( signal == 0 )` may be an infinite loop (because `signal` will never be changed by code *inside the loop*) and might generate the equivalent of

```
void waitForSignal() {
  if ( signal != 0 ) {
    return;
  } else {
    while(true) { // <-- Endless loop!
      // do nothing.
    }
  }
}
```

Considerate handling of `volatile` values

As stated above, a `volatile` variable can introduce a performance penalty when it is accessed more often than actually required. To mitigate this issue, you can "un-volatile" the value by assignment to a non-volatile variable, like

```
volatile uint32_t sysTickCount;
```

```
void doSysTick() {
  uint32_t ticks = sysTickCount; // A single read access to sysTickCount

  ticks = ticks + 1;

  setLEDState( ticks < 500000L );

  if ( ticks >= 1000000L ) {
    ticks = 0;
  }
  sysTickCount = ticks; // A single write access to volatile sysTickCount
}
```

This may be especially beneficial in ISR's where you want to be as quick as possible not accessing the same hardware or memory multiple times when *you* know it is not needed because the value will not change while your ISR is running. This is common when the ISR is the 'producer' of values for the variable, like the `sysTickCount` in the above example. On an AVR it would be especially painful to have the function `doSysTick()` access the same four bytes in memory (four instructions = 8 CPU cycles per access to `sysTickCount`) five or six times instead of only twice, because the programmer does know that the value will be not be changed from some other code while his/her `doSysTick()` runs.

With this trick, you essentially do the exact same thing the compiler does for non-volatile variables, i.e. read them from memory only when it has to, keep the value in a register for some time and write back to memory only when it has to; but this time, *you* know better than the compiler if/when reads/writes *must* happen, so you relieve the compiler from this optimization task and do it yourself.

Limitations of `volatile`

Non-atomic access

`volatile` does **not** provide atomic access to multi-word variables. For those cases, you will need to provide mutual exclusion by other means, *in addition* to using `volatile`. On the AVR, you can use `ATOMIC_BLOCK` from `<util/atomic.h>` or simple `cli(); ... sei();` calls. The respective macros act as a memory barrier too, which is important when it comes to the order of accesses:

Execution order

`volatile` imposes strict execution order only with respect to other volatile variables. This means that, for example

```
volatile int i;
volatile int j;
int a;

...
```

```
i = 1;
a = 99;
j = 2;
```

is guaranteed to *first* assign 1 to `i` and *then* assign 2 to `j`. However, it is *not* guaranteed that `a` will be assigned in between; the compiler may do that assignment before or after the code snippet, basically at any time up to the first (visible) read of `a`.

If it weren't for the memory barrier of the above mentioned macros, the compiler would be allowed to translate

```
uint32_t x;

cli();
x = volatileVar;
sei();
```

to

```
x = volatileVar;
cli();
sei();
```

or

```
cli();
sei();
x = volatileVar;
```

(For the sake of completeness I must say that memory barriers, like those implied by the sei/cli macros, may actually obviate the use of `volatile`, if *all* accesses are bracketed with these barriers.)

edited Dec 2 '18 at 22:08 answered Nov 29 '18 at 15:07

JimmyB
3,415 15 18

7 Good discussion of un-volatiling for performance :) – awjlogan Nov 29 '18 at 15:24

3 I always like to mention the definition of volatile in ISO/IEC 9899:1999 6.7.3 (6): `An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects.` More people should read it. – Jeroen3 Nov 29 '18 at 19:00

3 It may be worth mentioning that `cli` / `sei` is too heavy solution if your only goal is to achieve a memory barrier, not prevent interrupts. These macros generate actual `cli` / `sei` instructions, and additionally clobber memory, and it's this clobbering which results in the barrier. To have only a memory barrier without disabling interrupts you can define your own macro with the body like `__asm__ __volatile__("":::"memory")` (i.e. empty assembly code with memory clobber). – Ruslan Nov 29 '18 at 19:47

3 @NicHartley No. C17 5.1.2.3 §6 defines the *observable behavior*: "Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine." The C standard isn't really clear of where memory barriers are needed overall. At the end of an expression that uses `volatile` there is a sequence point, and everything after it must be "sequenced after". Meaning that expression *is* a memory barrier of sorts. Compiler vendors chose to spread all kinds of myths to put the responsibility of memory barriers on the programmer but that violates the rules of "the abstract machine". – Lundin Nov 30 '18 at 8:04

2 @JimmyB Local volatile maybe useful for code like `volatile data_t data = {0}; set_mmio(&data); while (!data.ready);` . – Maciej Piechotka Dec 2 '18 at 9:21

13

The volatile keyword tells the compiler that access to the variable has an observable effect. That means every time your source code uses the variable the compiler MUST create an access to the variable. Be that a read or write access.

The effect of this is that any change to the variable outside the normal code flow will also be observed by the code. E.g. if an interrupt handler changes the value. Or if the variable is actually some hardware register that changes on it's own.

This great benefit is also its downside. Every single access to the variable goes through the variable and the value is never held in a register for faster access for any amount of time. That means a volatile variable will be slow. Magnitudes slower. So only use volatile where it is actually necessary.

In your case, as far as you shown code, the global variable is only changed when you update it yourself by `adcValue = readADC();` . The compiler knows when this happens and will never hold the value of adcValue in a register across something that may call the `readFromADC()` function. Or any function it doesn't know about. Or anything that will manipulate pointers that might point to `adcValue` and such. There really is no need for volatile as the variable never changes in unpredictable ways.

answered Nov 29 '18 at 13:14

Goswin von Brederlow
648 4 10

6 I agree with this answer but "magnitudes slower" sounds too dire. – kkrambo Nov 29 '18 at 13:44

6 A CPU register can be accessed in less than a cpu cycle on modern superscalar CPUs. On the other hand an access to actual uncached memory (remember some external hardware would change this, so no CPU caches allowed) can be in the range of 100-300 CPU cycles. So, yes, magnitudes. Won't be so bad on an AVR or similar micro controller but the question doesn't specify hardware. – Goswin von Brederlow Nov 29 '18 at 13:53

7 In embedded (microcontroller) systems, the penalty for RAM access is often much less. The AVR's, for example, take only two CPU cycles for a read from or write to RAM (a register-register move takes one cycle), so the savings of keeping things in registers approach (but never actually reach) max. 2 clock cycles per access. - Of course, relatively speaking, saving a value from register X to RAM, then immediately reloading that value into register X for further calculations will take 2x2=4 instead of 0 cycles (when just keeping the value in X), and is hence infinitely slower :) – JimmyB Nov 29 '18 at 14:26

1 It's 'magnitudes slower' in the context of "writing to or reading from a particular variable", yes. However in the context of a complete program that likely does significantly more than read from/write to one variable over and over again, no, not really. In that case the overall difference is likely 'small to negligible'. Care should be taken, when making assertions about performance, to clarify if the assertion relates to one particular op or to a program as a whole. Slowing down an infrequently-used op by a factor of ~300x is almost never a big deal. – aroth Nov 30 '18 at 3:18

1 You mean, that last sentence? That's meant much more in the sense of "premature optimization is the root of all evil". Obviously you shouldn't use `volatile` on everything *just because*, but you also shouldn't shy away from it in cases where you think it's legitimately called for because of preemptive performance worries, either. – aroth Nov 30 '18 at 13:46

9

The main use of the volatile keyword on embedded C applications is to mark a global variable that is *written to* in an interrupt handler. It's certainly not optional in this case.

Without it, the compiler can't prove that the value is ever written to after initialization, because it can't prove the interrupt handler is ever called. Therefore it thinks it can optimize the variable out of existence.

answered Nov 29 '18 at 13:01

vicatcu
20.6k 8 67 140

2 Certainly other practical uses exist, but imho this is the most common. – vicatcu Nov 29 '18 at 13:02

1 If the value is only read in an ISR (and changed from main()), you potentially have to use volatile as well to guarantee ATOMIC access for multi byte variables. – Rev1.0 Nov 29 '18 at 13:23

15 @Rev1.0 No, volatile *does not* guarantee aromicity. That concern must be addressed separately. – Chris Stratton Nov 29 '18 at 14:03

1 There is no read from hardware nor any interrupts in the code posted. You are assuming things from the question which aren't there. It cannot really be answered in its current form. – Lundin Nov 29 '18 at 14:05

3 "mark a global variable that is written to in an interrupt handler" nope. It's to mark a variable; global or otherwise; that it may be changed by something outside of the compilers understanding. Interrupt not required. It could be shared memory or someone sticking a probe into the memory (latter not recommended for anything more modern than 40 years) – UKMonkey Nov 29 '18 at 16:07

9

There exist two cases where you must use `volatile` in embedded systems.

- When reading from a hardware register.

  That means, the memory-mapped register itself, part of hardware peripherals inside the MCU. It will likely have some cryptic name like "ADC0DR". This register must be defined in C code, either through some register map delivered by the tool vendor, or by yourself. To do it yourself, you'd do (assuming 16 bit register):

  ```
  #define ADC0DR (*(volatile uint16_t*)0x1234)
  ```

  where 0x1234 is the address where the MCU has mapped the register. Since `volatile` is already part of the above macro, any access to it will be volatile-qualified. So this code is fine:

  ```
  uint16_t adc_data;
  adc_data = ADC0DR;
  ```

- When sharing a variable between an ISR and the related code using the result of the ISR.

  If you have something like this:

  ```
  uint16_t adc_data = 0;

  void adc_stuff (void)
  {
    if(adc_data > 0)
    {
      do_stuff(adc_data);
    }
  }

  interrupt void ADC0_interrupt (void)
  {
    adc_data = ADC0DR;
  ```

```
}
```

Then the compiler might think: "adc_data is always 0 because it isn't updated anywhere. And that ADC0_interrupt() function is never called, so the variable can't be changed". The compiler usually doesn't realize that interrupts are called by hardware, not by software. So the compiler goes and removes the code `if(adc_data > 0){ do_stuff(adc_data); }` since it thinks it can never be true, causing a very strange and hard-to-debug bug.

By declaring `adc_data` `volatile`, the compiler is not allowed to make any such assumptions and it is not allowed to optimize away the access to the variable.

Important notes:

- An ISR shall always be declared inside the hardware driver. In this case, the ADC ISR should be inside the ADC driver. None else but the driver should communicate with the ISR - everything else is spaghetti programming.
- When writing C, all communication between an ISR and the background program *must* be protected against race conditions. *Always*, every time, no exceptions. The size of the MCU data bus does not matter, because even if you do a single 8 bit copy in C, the language cannot guarantee atomicity of operations. Not unless you use the C11 feature `_Atomic`. If this feature isn't available, you must use some manner of semaphore or disable the interrupt during read etc. Inline assembler is another option. `volatile` does not guarantee atomicity.

  What can happen is this:
  -Load value from stack into register
  -Interrupt occurs
  -Use value from register

  And then it doesn't matter if the "use value" part is a single instruction in itself. Sadly, a significant portion of all embedded systems programmers are oblivious to this, probably making it the most common embedded systems bug ever. Always intermittent, hard to provoke, hard to find.

An example of a correctly written ADC driver would look like this (assuming C11 `_Atomic` isn't available):

adc.h

```
// adc.h
#ifndef ADC_H
#define ADC_H

/* misc init routines here */

uint16_t adc_get_val (void);
```

```
#endif
```

adc.c

```
// adc.c
#include "adc.h"

#define ADC0DR (*(volatile uint16_t*)0x1234)

static volatile bool semaphore = false;
static volatile uint16_t adc_val = 0;

uint16_t adc_get_val (void)
{
  uint16_t result;
  semaphore = true;
    result = adc_val;
  semaphore = false;
  return result;
}

interrupt void ADC0_interrupt (void)
{
  if(!semaphore)
  {
    adc_val = ADC0DR;
  }
}
```

- This code is assuming that an interrupt cannot be interrupted in itself. On such systems, a simple boolean can act as semaphore, and it need not be atomic, as there is no harm if the interrupt occurs before the boolean is set. The down-side of the above simplified method is that it will discard ADC reads when race conditions occur, using the previous value instead. This can be avoided too, but then the code turns more complex.
- Here `volatile` protects against optimization bugs. It has nothing to do with the data originating from a hardware register, only that the data is shared with an ISR.
- `static` protects against spaghetti programming and namespace pollution, by making the variable local to the driver. (This is fine in single-core, single-thread applications, but not in multi-threaded ones.)

edited Apr 10 at 6:35

answered Nov 29 '18 at 15:12

Lundin
5,719 13 35

Hard to debug is relative, if the code is removed, you will notice that your valued code has gone - that is a pretty bold statement that something is amiss. But I agree, there can be very strange and hard to debug effects. – Arsenal Nov 29 '18 at 15:58

@Arsenal If you have a nice debugger that inlines assembler with the C, and you know at least a little bit asm, then yes it *can* be easy to spot. But for larger complex code, a large chunk of machine-generated asm isn't trivial to go through. Or if you don't know asm. Or if your debugger is crap and doesn't show asm (cougheclipsecough). – Lundin Nov 30 '18 at 7:47

Could be I'm a bit spoiled by using Lauterbach debuggers then. If you try to set a breakpoint in code that got optimized away it'll set it somewhere different and you know something is going on there. – Arsenal Nov 30 '18 at 9:39

@Arsenal Yep, the kind of mixed C/asm that you can get in Lauterbach is by no means standard. Most debuggers display the asm in a separate window, if at all. – Lundin Nov 30 '18 at 10:02

`semaphore` should definitely be `volatile` ! In fact, it's *the* most basic use case wich calls for `volatile` : Signal something from one execution context to another. - In your example, the compiler could just omit `semaphore = true;` because it 'sees' that its value is never read before it gets overwritten by `semaphore = false;` . – JimmyB Nov 30 '18 at 13:05

5

In the code snippets presented in the question, there is not yet a reason to use volatile. It's irrelevant that the value of `adcValue` comes from an ADC. And `adcValue` being global should get you suspicious of whether `adcValue` should be volatile but it's not a reason by itself.

Being global is a clue because it opens up the possibility that `adcValue` can be accessed from more than one program context. A program context includes an interrupt handler and an RTOS task. If the global variable is changed by one context then the other program contexts cannot assume they know the value from a previous access. Each context must re-read the variable value every time they use it because the value may have been changed in a different program context. A program context is not aware when an interrupt or task switch occurs so it must assume that any global variables used by multiple contexts may change between any accesses of the variable due to a possible context switch. This is what the volatile declaration is for. It tells the compiler that this variable can change outside of your context so read it every access and don't assume you already know the value.

If the variable is memory-mapped to a hardware address, then the changes made by the hardware is effectively another context outside the context of your program. So memory-mapped is also a clue. For example, if your `readADC()` function accesses a memory-mapped value to get the ADC value then that memory-mapped variable should probably be volatile.

So getting back to your question, if there is more to your code and `adcValue` gets accessed by other code that runs in a different context, then yes, `adcValue` should be volatile.

answered Nov 29 '18 at 13:21
kkrambo
1,800 7 13

> "Global variable that changes directly from the hardware"

4

Just because the value is coming from some hardware ADC register, doesn't mean that it is "directly" changed by hardware.

In your example, you just call readADC(), which returns some ADC register value. This is fine with respect to the compiler, knowing that adcValue is assigned a new value at that point.

It would be different if you were using an ADC interrupt routine to assign the new value, that is called when a new ADC value is ready. In that case, the compiler would have no clue about when the corresponding ISR is called and may decide that adcValue won't be accessed in this way. This is where volatile would help.

answered Nov 29 '18 at 12:59
Rev1.0
9,033 5 34 70

1 As your code never "calls" ISR function, Compiler sees that variable is only updated in a function which nobody calls. So compiler optimizes it. – Swanand Nov 29 '18 at 13:04

1 It depends of the rest of the code, if adcValue is not being read anywhere (like only read through the debugger), or if it's read only once in one place, the compiler will likely optimize it. – Damien Nov 29 '18 at 13:07

2 @Damien: It always "depends", but I was aiming to address the actual question "Should I use the keyword volatile in this case ?" as short as possible. – Rev1.0 Nov 29 '18 at 13:14

4

The behavior of the `volatile` argument largely depends on your code, the compiler, and the optimization done.

There are two use cases where I personally use `volatile`:

- If there is a variable I want to look at with the debugger, but the compiler has optimized it (means it has deleted it because it found out it is not necessary to have this variable), adding `volatile` will force the compiler to keep it and hence can be seen on debug.
- If the variable might change "out of the code", typically if you have some hardware accessing it, or if you map the variable directly to an address.

In embedded also there are sometimes quite some bugs in the compilers, doing optimization that actually doesn't work, and sometimes `volatile` can solve the problems.

Given you your variable is declared globally, it probably won't be optimized, as long as the variable is being used on the code, at least written and read.

Example:

```
void test()
{
    int a = 1;
    printf("%i", a);
}
```

In this case, the variable will probably be optimized to printf("%i", 1);

```
void test()
{
    volatile int a = 1;
    printf("%i", a);
}
```

won't be optimized

Another one:

```
void delay1Ms()
{
    unsigned int i;
    for (i=0; i<10; i++)
    {
        delay10us( 10);
    }
}
```

In this case, the compiler might optimize by (if you optimize for speed) and thus discarding the variable

```
void delay1Ms()
{
      delay10us( 10);
      delay10us( 10);
      delay10us( 10);
      delay10us( 10);
      delay10us( 10);
      delay10us( 10);
      delay10us( 10);
      delay10us( 10);
      delay10us( 10);
      delay10us( 10);
}
```

For your use case, "it might depend" on the rest of your code, how `adcValue` is being used elsewhere and the compiler version / optimization settings you use.

Sometimes it can be annoying to have a code that works with no optimization, but breaks once optimized.

```
uint16_t adcValue;
void readFromADC(void)
{
  adcValue = readADC();
  printf("%i", adcValue);
}
```

This might be optimized to printf("%i", readADC());

```
uint16_t adcValue;
void readFromADC(void)
{
  adcValue = readADC();
  printf("%i", adcValue);
  callAnotherFunction(adcValue);
}
```

--

```
uint16_t adcValue;
void readFromADC(void)
{
  adcValue = readADC();
  printf("%i", adcValue);
}

void anotherFunction()
{
    // Do something with adcValue
}
```

These probably won't be optimized, but you never know "how good the compiler is" and might change with the compiler parameters. Usually compilers with good optimization are licensed.

edited Nov 30 '18 at 5:15
Peter Mortensen
1,612 3 15 22

answered Nov 29 '18 at 12:37
Damien
3,564 1 5 20

1 For example a=1; b=a; and c=b; the compiler might think wait a minute, a and b are useless, let's just put 1 to c directly. Of course you won't do that in your code, but the compiler is better than you at finding these, also if you try to write optimized code right away it would be unreadable. – Damien Nov 29 '18 at 12:47

2 A correct code with a correct compiler will not break with optimizations turned on. The correctness of the compiler is a bit of a problem, but at least with IAR I haven't encountered a situation where the optimization lead to breaking code where it shouldn't. – Arsenal Nov 29 '18 at 13:03

5 A lot of cases where optimization breaks the code is when you are venturing into UB territory too.. – pipe Nov 29 '18 at 13:09

2 Yes, a side-effect of volatile is that it can aid debugging. But that is not a good reason to use volatile. You should probably turn off optimizations if easy debugging is your goal. This answer doesn't even mention interrupts. – kkrambo Nov 29 '18 at 13:42

2 Adding to the debugging argument, `volatile` forces the compiler to store a variable in RAM, and to update that RAM as soon as a value is assigned to the variable. Most of the time, the compiler does not 'delete' variables, because we usually don't write assignments without effect, but it may decide to keep the variable in some CPU register and may later or never write that register's value to RAM. Debuggers often fail in locating the CPU register in which the variable is held and hence cannot show its value. – JimmyB Nov 29 '18 at 14:22

1

Lots of technical explanations but I want to concentrate on the practical application.

The `volatile` keyword forces the compiler to read or write the variable's value from memory every time it is used. Normally the compiler will try to optimize but not making unnecessary reads and writes, e.g. by keeping the value in a CPU register rather than accessing memory each time.

This has two main uses in embedded code. Firstly it is used for hardware registers. Hardware registers can change, e.g. a ADC result register can be written by the ADC peripheral. Hardware registers can also perform actions when accessed. A common example is the data register of a UART, which often clears interrupt flags when read.

The compiler would normally try to optimize away repeated reads and writes of the register on the assumption that the value will never change so there is no need to keep accessing it, but the `volatile` keyword will force it to perform a read operation every time.

The second common use is for variables used by both interrupt and non-interrupt code. Interrupts are not called directly, so the compiler can't determine when they will execute, and thus assumes that any accesses inside the interrupt never happen. Because the `volatile` keyword forces the compiler to access the variable every time, this assumption is removed.

It is important to note that the `volatile` keyword is not complete solution to these issues, and care must be taken to avoid them. For example, on an 8 bit system a 16 bit variable requires two memory accesses to read or write, and thus even if the compiler is forced to make those accesses they occur sequentially, and it is possible for hardware to act on the first access or an interrupt to occur between the two.

answered Dec 3 '18 at 11:02
user
1,484 7 18

In the absence of a `volatile` qualifier, an object's value may be stored in more than one place

0

during certain parts of the code. Consider, for example, given something like:

```
int foo;
int someArray[64];
void test(void)
{
  int i;
  foo = 0;
  for (i=0; i<64; i++)
    if (someArray[i] > 0)
      foo++;
}
```

In the early days of C, a compiler would have processed the statement

```
foo++;
```

via the steps:

```
load foo into a register
increment that register
store that register back to foo
```

More sophisticated compilers, however, will recognize that if the value of "foo" is kept in a register during the loop, it will only need to be loaded once before the loop, and stored once after. During the loop, however, that will mean that the value of "foo" is being kept in two places--within the global storage, and within the register. This won't be a problem if the compiler can see all the ways that "foo" might be accessed within the loop, but may cause trouble if the value of "foo" is accessed in some mechanism the compiler doesn't know about (such as an interrupt handler).

It might have been possible for the authors of the Standard to add a new qualifier that would explicitly invite the compiler to make such optimizations, and say that the old-fashioned semantics would apply in its absence, but cases where the optimizations are useful vastly outnumber those where it would be problematic, so the Standard instead allows compilers to assume that such optimizations are safe in the absence of evidence that they aren't. The purpose of the `volatile` keyword is to supply such evidence.

A couple point of contention between some compiler writers and programmers occurs with situations like:

```
unsigned short volatile *volatile output_ptr;
unsigned volatile output_count;

void interrupt_handler(void)
{
  if (output_count)
  {
    *((unsigned short*)0xC0001230) = *output_ptr; // Hardware I/O register
    *((unsigned short*)0xC0001234) = 1; // Hardware I/O register
    *((unsigned short*)0xC0001234) = 0; // Hardware I/O register
    output_ptr++;
    output_count--;
  }
}

void output_data_via_interrupt(unsigned short *dat, unsigned count)
{
  output_ptr = dat;
  output_count = count;
  while(output_count)
     ; // Wait for interrupt to output the data
}

unsigned short output_buffer[10];

void test(void)
{
  output_buffer[0] = 0x1234;
  output_data_via_interrupt(output_buffer, 1);
  output_buffer[0] = 0x2345;
  output_buffer[1] = 0x6789;
  output_data_via_interrupt(output_buffer,2);
}
```

Historically, most compilers would either allow for the possibility that writing a `volatile` storage location could trigger arbitrary side-effects, and avoid caching any values in registers across such a store, or else they will refrain from caching values in registers across calls to functions that are not qualified "inline", and would thus write 0x1234 to `output_buffer[0]` , set things up to output the data, wait for it to complete, then write 0x2345 to `output_buffer[0]` , and continue on from there. The Standard doesn't *require* implementations to treat the act of storing the address of `output_buffer` into a `volatile` -qualified pointer as a sign that something might happen to it via means the compiler doesn't understand, however, because the authors thought compiler the writers of compilers intended for various platforms and purposes would recognize when doing so would serve those purposes on those platforms without having to be told. Consequently, some "clever" compilers like gcc and clang will assume that even though the address of `output_buffer` is written to a volatile-qualified pointer between the two stores to `output_buffer[0]` , that's no reason to assume that anything might care about the value held in that object at that time.

Further, while pointers that are directly cast from integers are seldom used for any purpose other than to manipulate things in ways that compilers aren't likely to understand, the Standard again does not require compilers to treat such accesses as `volatile` . Consequently, the first write to `*((unsigned short*)0xC0001234)` may be omitted by "clever" compilers like gcc and clang, because the maintainers of such compilers would rather claim that code which neglects to qualify such things as `volatile` is "broken" than recognize that compatibility iwth such code is useful. A lot of vendor-supplied header files omit `volatile` qualifiers, and a compiler which is compatible with vendor-supplied header files is more useful than one that isn't.

answered Nov 30 '18 at 17:07

supercat
40.4k 1 66 115