Volatile variable

`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.

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.

Using volatile in MCU

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;
```

```
uint16_t adc_data = 0;

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

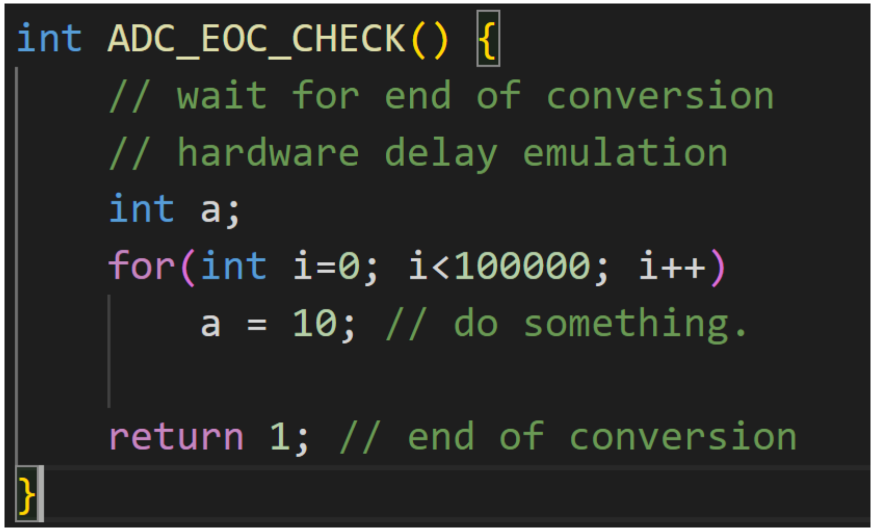
```
interrupt void ADC0_interrupt (void)
{
  adc_data = ADC0DR;
}
```

Using volatile in MCU

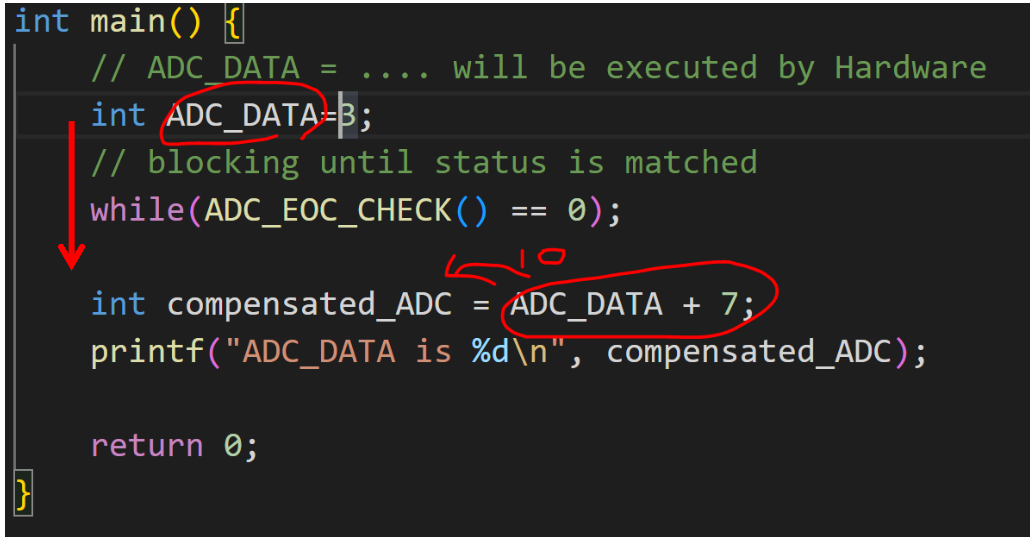
```c
// adc.c
#include "adc.h"

#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;
  }
}
```

Memory I/O를 최대한 줄이도록 컴파일됨

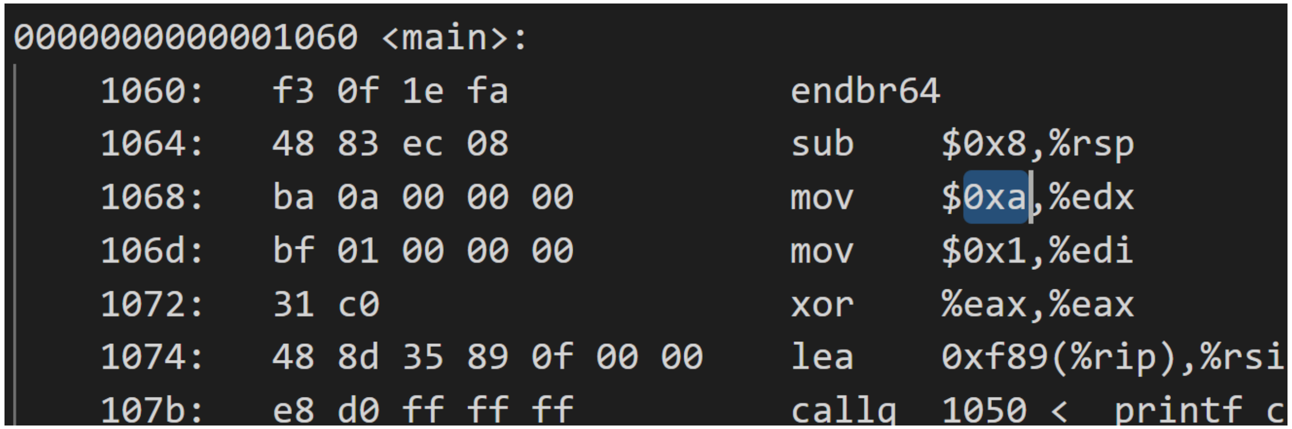
ADC_DATA에 값을 write한 뒤 긴 시간 write없어도 스스로 값이 바뀌지 않는다고 생각해야 함 (메모리 값 유지되므로)



```c
int ADC_EOC_CHECK() {
    // wait for end of conversion
    // hardware delay emulation
    int a;
    for(int i=0; i<100000; i++)
        a = 10; // do something.

    return 1; // end of conversion
}
```



```c
int main() {
    // ADC_DATA = .... will be executed by Hardware
    int ADC_DATA=3;
    // blocking until status is matched
    while(ADC_EOC_CHECK() == 0);

    int compensated_ADC = ADC_DATA + 7;
    printf("ADC_DATA is %d\n", compensated_ADC);

    return 0;
}
```

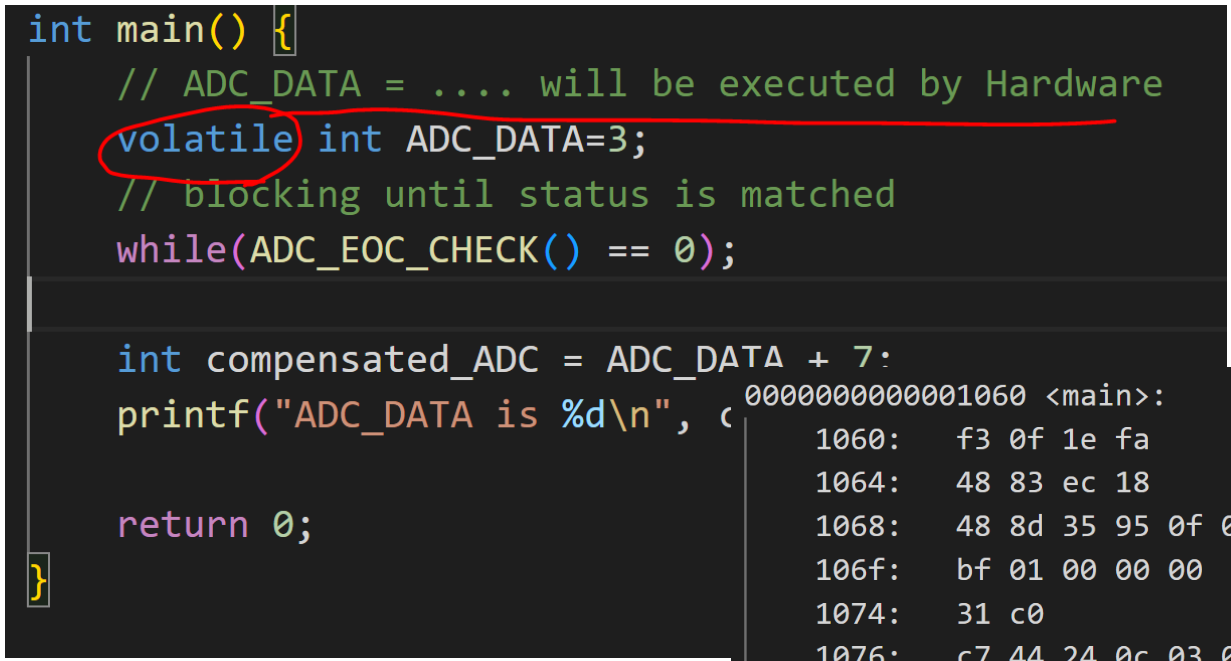


```
0000000000001060 <main>:
    1060:    f3 0f 1e fa             endbr64
    1064:    48 83 ec 08             sub    $0x8,%rsp
    1068:    ba 0a 00 00 00          mov    $0xa,%edx
    106d:    bf 01 00 00 00          mov    $0x1,%edi
    1072:    31 c0                   xor    %eax,%eax
    1074:    48 8d 35 89 0f 00 00    lea    0xf89(%rip),%rsi
    107b:    e8 d0 ff ff ff          callq  1050 <__printf_c
```

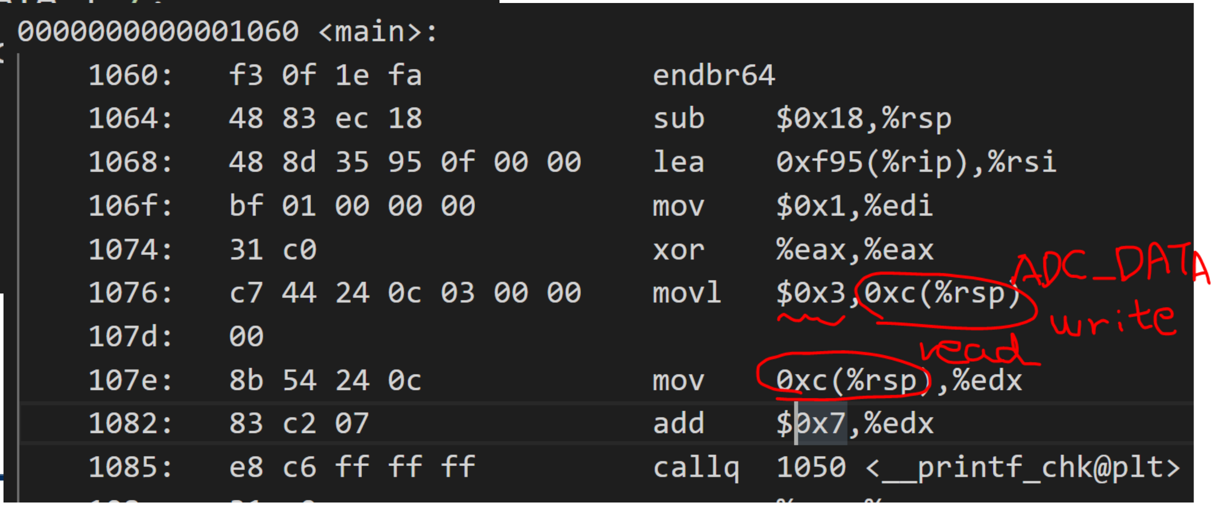
- 따라서 굳이 ADC_DATA를 메모리로부터 읽어올 필요없음.
- ADC_DATA에 값을 쓰고 읽는 코드는 사라지고, 3과 7을 컴파일 타임에 더해서 최종 10을 사용함

명시적으로 Memory I/O를 반드시 수행함

- 변수에 접근 (read/write)하는 코드는 메모리 접근해서 읽고 쓰는 코드로 변환됨
  - 아무리 짧은 구간이든, 아무리 긴 시간이든 c코드에서 그 변수에 assign하지 않더라도 스스로 값이 바뀔수 있다면.. (휘발성, volatile) 명시적으로 메모리로부터 그 값을 읽어오는 것이 맞다. ➔ 접근하는 메모리 주소가 실제 하드웨어에 매핑되어 있다면 (memory mapped I/O) 반드시 그 변수 영역을 volatile로 선언하라.



```c
int main() {
    // ADC_DATA = .... will be executed by Hardware
    volatile int ADC_DATA=3;
    // blocking until status is matched
    while(ADC_EOC_CHECK() == 0);

    int compensated_ADC = ADC_DATA + 7;
    printf("ADC_DATA is %d\n", c

    return 0;
}
```



```
0000000000001060 <main>:
    1060:    f3 0f 1e fa              endbr64
    1064:    48 83 ec 18              sub    $0x18,%rsp
    1068:    48 8d 35 95 0f 00 00     lea    0xf95(%rip),%rsi
    106f:    bf 01 00 00 00           mov    $0x1,%edi
    1074:    31 c0                    xor    %eax,%eax
    1076:    c7 44 24 0c 03 00 00     movl   $0x3,0xc(%rsp)
    107d:    00
    107e:    8b 54 24 0c              mov    0xc(%rsp),%edx
    1082:    83 c2 07                 add    $0x7,%edx
    1085:    e8 c6 ff ff ff           callq  1050 <__printf_chk@plt>
```