Volatile Type Qualifier

the volatile type qualifier in C is used to inform the compiler that a variable's value may be changed at any time without any action being taken by the code the compiler finds nearby.

This prevents the compiler from optimizing code in a way that assumes the value of the variable cannot change unexpectedly.

The volatile qualifier is essential in certain scenarios, particularly in embedded systems, where hardware peripherals, interrupts, or multi-threaded programs can alter the value of a variable.

Proper Use of C's volatile Keyword

- Memory-mapped peripheral registers
- Global variables modified by an interrupt service routine
- Global variables accessed by multiple tasks within a multi-threaded application

Declaration

1. **Declaration 1:**

```c
volatile uint8_t pReg;
```

2. **Declaration 2:**

```c
uint8_t volatile pReg;
```

- uint8_t volatile * pReg;
- volatile uint8_t * pReg;
  - pointer to a volatile unsigned 8-bit integer

- int * volatile p;
  - Volatile pointers to non-volatile data

- int volatile * volatile p;
  - volatile pointer to a volatile variable

declare a pointer to a volatile unsigned 8-bit integer

1. **Declaration 1:**

```c
volatile uint8_t * pReg;
```

2. **Declaration 2:**

```c
uint8_t volatile * pReg;
```

Explanation

- *volatile uint8_t * pReg; :* This declares pReg as a pointer to a volatile uint8_t. It means that the uint8_t value that pReg points to is volatile, and therefore, it may change at any time without any action from the code.

- *uint8_t volatile * pReg; :* This declaration does the same thing. The volatile keyword can be placed after the type specifier (uint8_t) and it has the same effect as placing it before the type specifier.

When to Use volatile

Hardware Registers:

When accessing memory-mapped hardware registers in embedded systems. These registers can be changed by hardware, so the compiler should always read their values directly from memory rather than using cached values.

```c
volatile int *status_register = (int *)0x40021000;
```

Interrupt Service Routines (ISRs):

- Variables that can be modified by an interrupt service routine (ISR). The main program and the ISR may both access the variable, and the ISR can change the variable at any time.

```c
volatile int timer_flag;

void timer_ISR(void) {
    timer_flag = 1;
}

void main(void) {
    while (!timer_flag) {
        // Wait for the timer ISR to set the flag
    }
}
```

Shared Variables in Multi-threaded Programs:

- Variables shared between different threads or tasks in a multi-threaded environment. Although other synchronization mechanisms (like mutexes) are usually required to ensure safe access, volatile ensures the variable's value is not cached between accesses.

```c
volatile int shared_data;

void thread1(void) {
    shared_data = 1;
}

void thread2(void) {
    while (!shared_data) {
        // Wait for thread1 to set the data
    }
}
```

How **volatile** *Works*

When a variable is declared with the volatile qualifier, the compiler generates code that always reads the variable from memory instead of using a cached value stored in a register. This ensures the program sees the most recent value of the variable at all times.

Example

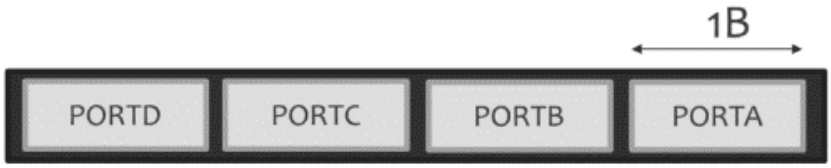
c نسخ الكود

```c
#include <stdint.h>

#define STATUS_REG   (*(volatile uint32_t *)0x40021000)

void check_status(void) {
    while (STATUS_REG != 0) {
        // Perform some action based on the status register
    }
}
```

In this example, **STATUS_REG** is a memory-mapped hardware register. By declaring it as volatile, the compiler is instructed to always read its value from the specified memory address, ensuring the latest value is used each time it is accessed.

Now let's see how access the register absolute address

Write 0xFFFFFFFF on SIU register which have absolute address 0x30610000




```c
#include <stdint.h>

typedef union {
    uint32_t ALL_ports;
    struct {
        uint32_t PORTA:8 ;
        uint32_t PORTB:8 ;
        uint32_t PORTC:8 ;
        uint32_t PORTD:8 ;
    } SIU_fields;
} SIU_R;

#define SIU_REGISTER_ADDRESS 0x306100

int main(void) {
    // Define a pointer to the SIU register
    volatile SIU_R* PORTS = (volatile SIU_R*) SIU_REGISTER_ADDRESS;

    // Set all bits of the register to 0xFFFFFFFF
    PORTS->ALL_ports = 0xFFFFFFFF;

    // Set only PORTA to 0xFF, other fields remain unchanged
    PORTS->SIU_fields.PORTA = 0xFF;

    // Your code here

    return 0;
}
```

Solution 2

((volatile unsigned long)(0x306100)) = 0xFFFFFFFF;

Or

#define MYREGISTER *((volatile unsigned long*)(0x306100))

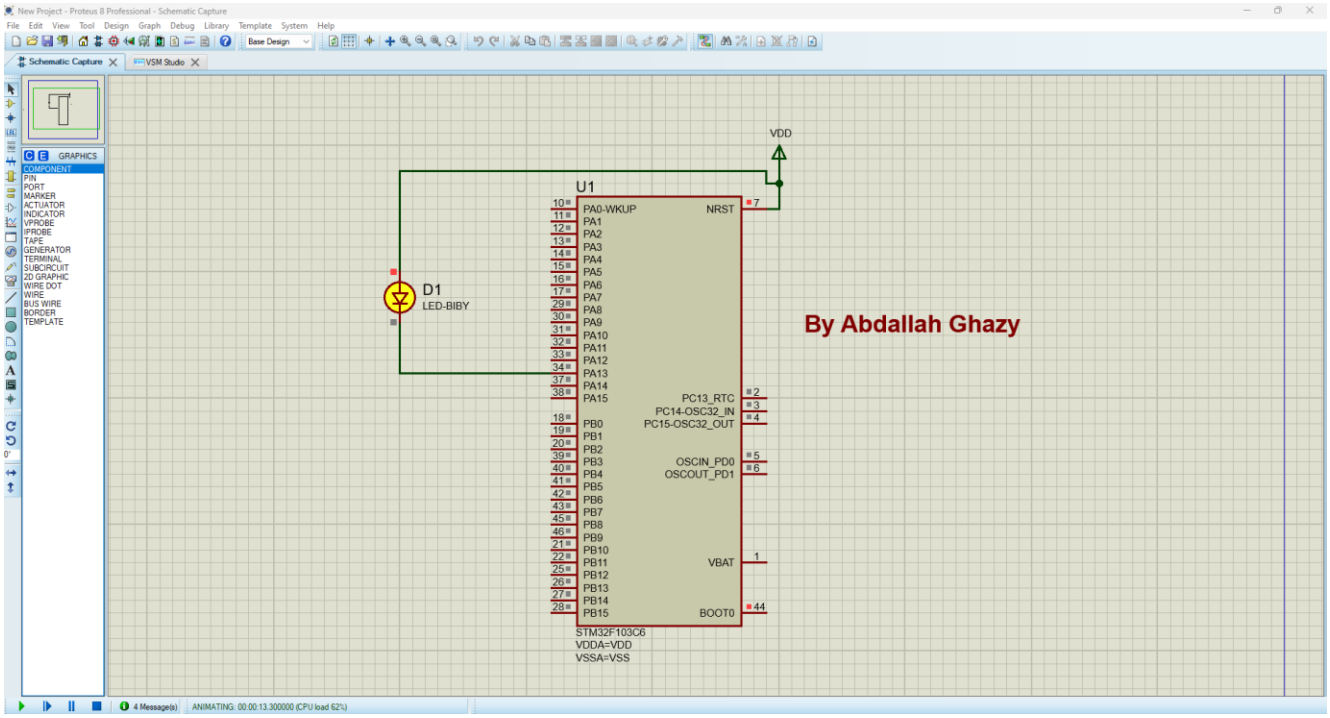
Toole led on Stm32f103CX

Led is connected to GPIO port A13

To make a GPIO toggling in STM32, you need to work with two peripherals:

- RCC (reset and clock control)
  - The RCC is necessary because the GPIO has disabled clock by default

- GPIOx (general purpose input/output).



```c
#include <stdint.h>

typedef volatile unsigned int vint32;

#define HIGH 1
#define LOW  0

#define RCC_BASE       0x40021000
#define PORTA_BASE     0x40010800

#define RCC_APB2ENR    *((volatile uint32_t*)(RCC_BASE+0x18))
#define GPIOA_CRH      *((volatile uint32_t*)(PORTA_BASE+0x04))
//#define GPIOA_ODR    *((volatile uint32_t*)(PORTA_BASE+0x0C))
#define GPIOA_ODR       (PORTA_BASE + 0x0C)

#define RCC_APB2ENR_IOPAEN (1<<2)

typedef union {
	vint32 allFields;

	struct {
		vint32 :13;
		vint32 pin13 :1;
	}pin;

} R_ODR_t;

int main(void) {

	volatile R_ODR_t *R_ODR =  (volatile R_ODR_t*)GPIOA_ODR;

	RCC_APB2ENR |= RCC_APB2ENR_IOPAEN;
	GPIOA_CRH &= 0xff0fffff;
	GPIOA_CRH |= 0x00200000;

	while (1) {

		//GPIOA_ODR |= 1 << 13;
		R_ODR->pin.pin13 = HIGH;
		for (int i = 0; i < 5000; i++);

		//GPIOA_ODR &= ~(1 << 13);
		R_ODR->pin.pin13 = LOW;
		for (int i = 0; i < 5000; i++);
	}

	return 0;
}
```

Understanding const and volatile Together

Qualifiers Explanation:

- **const**: Indicates that the value of the variable should not be changed by the program. The compiler will enforce this restriction, meaning you cannot modify the value through that pointer or reference.

- **volatile**: Indicates that the value of the variable can change at any time without any action being taken by the code the compiler finds nearby. This is used to tell the compiler not to optimize accesses to this variable, as it might be changed by hardware or other threads.

Pointer Definitions:

- **uint32_t volatile * const Preg1**:
  - Preg1 is a constant pointer to a volatile uint32_t.
  - This means that the address stored in Preg1 cannot be changed (i.e., Preg1 itself is constant), but the value at that address can be changed unexpectedly (e.g., by hardware).

- Example:

```c
uint32_t volatile * const Preg1 = (uint32_t volatile *)0xFFFF0000;
// You can read from or write to *Preg1, but you can't change Preg1 itself.
```

*uint32_t const volatile * const Preg2:*

- Preg2 is a constant pointer to a const volatile uint32_t.
- This means that the address stored in Preg2 cannot be changed (i.e., Preg2 itself is constant), and the value at that address is volatile but also constant (read-only) in the context of the program.

- Example:

```c
uint32_t const volatile * const Preg2 = (uint32_t const volatile *)0xFFFF0004;
// You can only read from *Preg2. The address Preg2 itself can't be changed.
```

Practical Implications

- **const volatile**: This combination is often used for hardware registers that are read-only but may be updated by hardware. It ensures that the compiler does not optimize away accesses to the register because the hardware might change its value. At the same time, it prevents the programmer from modifying the register's value directly.

Summary

- **const** prevents modification of the pointer itself.
- **volatile** prevents optimization of the memory accesses to handle unexpected changes.
- **const volatile** indicates that the data is read-only from the perspective of the programmer, but it may be updated by external factors (e.g., hardware).

Usage Context

- **Preg1**:
  - **Example**: A hardware register that can be written to and read from, but the address of the register should not be changed.
  - **Use Case**: Register configuration where you configure something once and monitor the status.

- **Preg2**:
  - **Example**: A read-only status register that can be changed by hardware but should not be modified by the programmer.
  - **Use Case**: Reading status flags or error codes from hardware peripherals.