Understanding the `volatile` Keyword in Embedded C Programming

A.Jegan

What is the `volatile` variable?

- A volatile variable is one that can change unexpectedly.
- Compiler can make no assumptions about the value of the variable.
- C compiler optimizer must be careful to reload the variable every time it's used instead of holding a copy in a register.

The Need for `volatile` in Embedded Systems

- Preventing Compiler Optimizations
- Used in hardware interaction, especially for memory-mapped peripherals.
- when variables are shared by multiple tasks in a multi-threaded application
- To avoid loop hoisting, loop unrolling
- To avoid Variable optimization
- Non-automatic variables referenced within an interrupt service routine

Preventing Compiler Optimizations

- The `volatile` keyword prevents compilers from optimizing code related to the marked variable.
- It ensures that the value of the variable is always read from its memory location.
- This is crucial for variables that can change outside the normal program flow.

C Code Example: Using volatile

```c
#include <stdio.h>
// Function to simulate an external update
void externalUpdate(volatile int* ptr) {
    *ptr = *ptr + 1;
}
int main() {
    volatile int flag = 0;
    while (1) {
        if (flag == 1) {
            printf("Flag updated to 1\n");
            break;
        }
        externalUpdate(&flag);
    }
    return 0;
}
```

Use in Embedded Systems and Hardware Interaction

- In embedded systems, variables may correspond to hardware registers.
- The `volatile` keyword ensures correct interaction with hardware by reading the actual value from memory each time.
- Essential for handling hardware events or interrupts.

C Code Example: Using volatile

```c
#include <stdint.h>
// Define the memory-mapped peripheral address
#define STATUS_REGISTER_ADDR 0x40001000
// Create a pointer to the status register
volatile uint32_t* const STATUS_REGISTER = (
    volatile uint32_t*)STATUS_REGISTER_ADDR;
int main() {
    // Wait until bit 0 of the status register
    is set
    while ((*STATUS_REGISTER & 0x01) == 0) {
        // Polling - do nothing here and keep
    checking
    }
    // Bit 0 is set, perform the required action
    // ...
    return 0;
}
```

Multithreading Applications

- In multithreaded applications, `volatile` ensures a variable is always read from memory.
- It's not a substitute for thread synchronization but is relevant in low-level threading scenarios.
- Helps in maintaining consistency across multiple threads.

C Code Example:

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

volatile int stopFlag = 0;

void* threadFunction(void* arg) {
    while (!stopFlag) {
        printf("Thread is running...\n");
        sleep(1); // Simulate work
    }
    printf("Thread is stopping...\n");
    return NULL;
}
```

C Code Example:

```
int main() {
    pthread_t threadId;

    // Create a thread
    if (pthread_create(&threadId, NULL,
                       threadFunction, NULL) !=
  0) {
        perror("Failed to create thread");
        return 1;
    }
    sleep(5); // Simulate main thread work
    stopFlag = 1; // Signal the thread to stop
    pthread_join(threadId, NULL); // Wait for
  the thread to finish
    printf("Thread has stopped\n");
    return 0;
}
```

Signal Handlers

- For variables accessed in signal handlers and the main program.
- `volatile` ensures that the program and handler see the correct value.
- Important for handling asynchronous changes by signal handlers.

C Code Example:

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile sig_atomic_t signalFlag = 0;

void signalHandler(int signal) {
    if (signal == SIGINT) {
        signalFlag = 1;
    }
}

int main() {
    // Set up the signal handler
    struct sigaction sa;
    sa.sa_handler = signalHandler;
    sa.sa_flags = 0; // or SA_RESTART to restart system calls
    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);
    printf("Waiting for SIGINT (Ctrl+C)...\n");
    // Main loop
    while (!signalFlag) {
        // Main program does its work here
        sleep(1); // Simulate some work
    }
    printf("SIGINT received, exiting program.\n");
    return 0;
}
```

Further Resources

- https://www.embeddedrelated.com/thread/4749/when-and-how-to-use-the-volatile-keyword-embedded-c-pr
- https://barrgroup.com/embedded-systems/how-to/c-volatile-keyword
- https://blog.mbedded.ninja/programming/languages/c/embedded-systems-and-the-volatile-keyword