В. Тимофеев

osa@pic24.ru

volatile для «чайников»

1. volatile

1.1. Вступление

Данная статья предназначена для программистов встраиваемых систем, в основном - для начинающих работать с языком Си, но опытные программисты, надеюсь, тоже смогут что-то для себя подчерпнуть. При написании статьи я, возможно, чего-то не учел или где-то ошибся. Буду рад любым поправкам и любой критике.

Разбирая чужие исходники, часто натыкаюсь на ошибки программистов, связанные с недопониманием назначения квалификатора `volatile`. Результатом такого недопонимания является код, который дает редкие, совершенно непредсказуемые и, зачастую, очень разрушительные и необратимые сбои. Это особенно актуально для микроконтроллерных систем, где, во-первых, обработчики прерываний являются частью прикладного кода, а, во-вторых, регистры управления периферией отображаются в RAM общего назначения. Ошибки, связанные с неиспользованием (или неправильным использованием) квалификатора `volatile` трудно поддаются отладке из-за их непредсказуемости и неповторяемости. Бывает, что Си-код, в котором не предусмотрено использование `volatile` там, где его надо бы использовать, прекрасно работает, будучи собранным одним компилятором, и сбоит (или не работает вовсе), когда собирается другим.

Большинство примеров, приведенных в этой статье, написаны для компилятора GCC (Mplab C30), потому что, учитывая архитектуру ядра PIC24 и особенности компилятора, для него проще всего синтезировать маленькие наглядные примеры, в которых будет проявляться неправильное обращение с `volatile`. Многие из этих примеров будут собираться совершенно корректно на других (более простых) компиляторах, таких как PICC или MicroC. Но это не значит, что при работе с этими компиляторами ошибки неиспользования `volatile` не проявляются вовсе. Просто код демонстрации для этих компиляторов выглядел бы намного больше и сложнее.

1.2. Определение

(`volatile` в переводе с английского означает "нестабильный", "изменчивый")

Итак, `volatile` в языке Си - это квалификатор переменной, говорящий компилятору, что значение переменной может быть изменено в любой момент и что часть кода, которая производит над этой переменной какие-то действия (чтение или запись), не должна быть оптимизирована.

Что это значит? Известно, что одной из характеристик компиляторов, говорящих за их качество, является способность оптимизировать генерируемый объектный код. Для этого они объединяют повторяющиеся конструкции, сохраняют в регистрах общего назначения промежуточные результаты вычислений, выстраивают последовательность команд так, чтобы минимизировать долго выполняющиеся фрагменты кода

(например, обращение через косвенную адресацию), и т.д. Выполняя такую оптимизацию, они немного преобразует наш код, подменяя его идентичным с точки зрения алгоритма, но более быстрым и/или компактным. Но такую подмену можно делать не всегда. Рассмотрим пример:

```
char a = 0;
...
a |= 1;
a |= 2;
...
```

С точки зрения алгоритма устанавливаются два младших разряда в переменной a. Оптимизатор может сделать подмену такого кода одним оператором:

```
a |= 3;
```

выиграв таким образом пару тактов и пару ячеек ROM. Но представим себе, что эти же действия мы выполняем не над какой-то абстрактной переменной, а над периферийным регистром:

```
PORTB |= 1;
PORTB |= 2;
```

Вот в этом случае оптимизация с заменой на "PORTB |= 3" нас может не устроить! Управляя напрямую состояниями выводов контроллера, нам часто бывает важна последовательность изменения сигналов. Например, мы формируем сигналы SPI, и один вывод (PORTB.0) - это данные, а другой (PORTB.1) - синхроимпульсы. В этом случае нам нельзя изменять состояния этих выводов одновременно, т.к. при этом нельзя гарантировать, что управляемая микросхема по синхроимпульсу получит правильные данные. И уж тем более, нам бы не хотелось, чтобы оптимизации подвергся код, формирующий синхроимпульс длительностью в один такт:

```
PORTB |= 2;
PORTB &= ~2;
```

Такой код мог бы быть воспринят компилятором как два взаимообратных действия и первая строка могла бы не попасть в результирующий объектный код. Однако на практике мы видим, что такой оптимизации не производится. Так происходит именно потому, что переменная, на которую отображается регистр PORTB, объявлена с квалификатором `volatile`, например:

```
extern volatile      unsigned char PORTB @ 0x06;                    // в HT-PICC
extern volatile near unsigned char PORTB;                           // в MCC18
extern volatile near unsigned int  PORTB __attribute__((__sfr__)); // в MCC30
и т.д.
```

(в этом можно убедиться, заглянув в заголовочный файл для конкретного контроллера, поставляемый с компилятором).

Квалификатор `volatile` запрещает производить оптимизацию кода, выполняющего действия над регистром PORTB. Поэтому даже взаимообратные действия останутся нетронутыми оптимизатором, и мы можем быть уверенны в том, что на выходе сформируется импульс.

Итак, выполняя оптимизацию, компиляторы стремятся помочь нам сделать наш код максимально быстрым и максимально компактным. Однако, если не использовать `volatile`, в некоторых случаях оптимизация может сыграть с программистом злую шутку. Причем, надо сказать, чем умнее и мощнее компилятор, тем больше проблем он может создать при неграмотном использовании `volatile`.

2. Ошибки, связанные с volatile

Есть три основных типа ошибок, касающихся квалификатора `volatile`:

1. **неиспользование volatile там, где нужно** - обычно совершается программистами, которые не знают про существование `volatile`, или видели, но не понимают, что это такое;

2. **использование volatile там, где нужно, но не так, как нужно** -присуща программистам, знающим, насколько важен `volatile` при программировании параллельных процессов или при доступе к периферийным регистрам, но не учитывающие некоторые его нюансы;

3. **использование volatile там, где не нужно** - (бывает и такое) такое делают те, кто однажды обжегся на первых двух ошибках. Это не ошибка и она не приведет к неправильному поведению программы, но создаст свои неприятности.

2.1. Неиспользование volatile

2.1.1. Глобальные переменные

Самая частая ошибка, связанная с `volatile`, это просто неиспользование этого квалификатора там, где это нужно. Работая с микроконтроллерами,

программист почти всегда сам является автором кода основной программы и кода для обработки прерываний. Причем основная программа и прерывания должны обмениваться данными, а самым распространенным способом для этого является использование глобальных переменных (будь то счетчики, привязанные к периоду переполнения таймера, или буферы для хранения входных/выходных данных, или переменные состояния, или еще что-то).

Модификация переменной в прерывании

Рассмотрим пример для PIC24:

```
unsigned char Counter;

void __attribute__((__interrupt__, __auto_psv__)) _T1Interrupt (void)
{
    IFS0bits.T1IF = 0;
    Counter++;
}

void wait (unsigned char Time)
{
    Counter = 0;
    while (Counter < Time) continue;
}
```

При отключенной оптимизации данный код будет работать. Но стоит включить оптимизацию, как программа начнет зависать в функции wait(). Что происходит? Компилятор, транслируя функцию wait(), не знает про то, что переменная Counter может измениться в любой момент при возникновении прерывания. Он видит только то, что мы ее обнуляем, а затем сразу сравниваем с параметром Time. Другими словами компилятор предполагает, что переменная Time всегда сравнивается с нулем, и листинг функции wait() при включенной оптимизации будет выглядеть так:

```
0x5B4   wait:   clr.b   Counter
0x5B6           cp0.b   w0          ; <-----
0x5B8           bra     nz, 0x5B6
0x5BA           return
```

(Примечание: компилятор Си единственный однобайтовый аргумент передает в функцию через регистр w0)

Что мы видим в этом коде: производится обнуление переменной Counter, а затем параметр функции Time, переданный в нее через регистр w0, сравнивается с нулем, больше не обращая внимания на истинное значение переменной Coutner, которая исправно увеличивается при каждом прерывании по таймеру. Другими словами, мы попали в вечный цикл. Как уже говорилось, дело в том, что компилятор не предполагает, что функция будет прервана каким-то кодом, который будет производить операции над переменными, участвующими в работе функций. Здесь нам на помощь и приходит

квалификатор `volatile`:

```
volatile unsigned char Counter;

void __attribute__((__interrupt__)) _T1Interrupt (void)
{
    IFS0bits.T1IF = 0;
    Counter++;
}

void wait (unsigned char Time)
{
    Counter = 0;
    while (Counter < Time) continue;
}
```

Теперь при трансляции компилятор сгенерит код, который будет каждый раз обращаться к переменной Counter:

```
0x5B4   wait:   mov.b   w0, w1
0x5B6           clr.b   Counter
0x5B8           mov.b   Counter, w0      ; <-----
0x5BA           sub     w0, w1, [w15]
0x5BC           bra     nc, 0x5B8
0x5BE           return
```

Модификация переменной в параллельной задаче

Если программа работает под управлением ОСРВ, т.е. выполнение функции может быть прервано в какой-то момент, а потом опять передано ей с того места, где она прервалась. Не важно, кооперативный ли планировщик у ОС (т.е. сам программист решает, где функции быть прерванной) или вытесняющий (тут программист вообще ничего не решает, и его функция может быть прервана абсолютно в любой момент времени более приоритетной задачей). Важно то, что компилятор не знает о том, что в середине функции может быть выполнен какой-то посторонний код. Вот фрагмент кода из одной реальной программы:

```
unsigned char Button;

void Task_Button (char NewLevel)
{
    unsigned int Temp;

    for (;;)
    {
        //...
        // Переключение контекста
        //...

        Temp = PORTB;
        //...
        // Обработка дребезга
        //...

        Button = 0;
        if (Temp & 1) Button = 1;
        if (Temp & 2) Button = 2;
        if (Temp & 4) Button = 3;
    }
}

void Task_Work ()
{
    if (!Button) PIN_RED_LED = 1;
    for (;;)
    {
        //...
        // Переключение контекста
        // и пр.
        //...

        switch (Button)
        {
            case 1: //... Действия по кнопке 1
                    break;
            case 2: //... Действия по кнопке 2
                    break;
            case 3: //... Действия по кнопке 3
                    break;
        }
    }
}
```

Программа хорошо работала, будучи собранной компилятором HT-PICC18, но при переносе этого же кода на PIC24 (компилятор MCC30) работать перестала, т.е. совсем не реагировала на кнопки. Проблема была в том, что оптимизатор MCC30, в отличие от оптимизатора HT-PICC18, учел, что на момент выполнения switch значение переменной Button уже хранится в одном из регистров общего назначения (w0) (в PIC18 всего 1 аккумулятор, поэтому при работе с ним такое поведение менее вероятно):

```
;                               if (!Button) PIN_RED_LED = 1;
    mov.b   Button, w0
    bra     nz, 1f
    bset.b  PORTB, 0
1:
    ...
    ...
;                               switch (Button)
    sub.b   w0, #2, [w15]
    bra     z, Case2
    sub.b   w0, #3, [w15]
    bra     z, Case3
    sub.b   w0, #1, [w15]
    bra     nz, SwithEnd
Case1:
    ...
Case2:
    ...
Case3:
    ...
SwithEnd:
```

Автор этого кода рассказал, что при отладке сломал кнопку, пытаясь нажать ее сильнее, чтобы она хоть как-то сработала :). Его ошибка заключалась в том, что он не использовал `volatile` при объявлении переменной Button. Сделай он это - и компилятор знал бы, что при каждом обращении к переменной Button нужно выполнять инструкции для фактического обращения к ячейке памяти, а не использовать для ускорения промежуточный результат.

После правильного объявления переменной:

```
volatile unsigned char Button;

void Task_Button (char NewLevel)
{
    ...
}

void Task_Work ()
{
    ...
}
```

проблема исчезла:

```
;                               if (!Button) PIN_RED_LED = 1;
    mov.b   Button, w0
    bra     nz, 1f
    bset.b  PORTB, 0
1:
    ...
    ...
;                               switch (Button)
    mov.b   Button, w0
    sub.b   w0, #2, [w15]
    bra     z, Case2
    mov.b   Button, w0
    sub.b   w0, #3, [w15]
    bra     z, Case3
    mov.b   Button, w0
    sub.b   w0, #1, [w15]
    bra     nz, SwithEnd
Case1:
    ...
Case2:
    ...
Case3:
    ...
SwithEnd:
```

2.1.2. Локальные переменные

Часто для создания небольших задержек пользуются такими функциями:

```
void Delay (char D)
{
    char i;
    for (i = 0; i < D; i++) continue;
}
```

Однако, некоторые компиляторы видят в этих функциях бесполезный код и вообще не включают его в результирующий объектный код. Если эта задержка применялась для снижения скорости программного i2c (или SPI) под компилятором, например, HT-PICC, то при переносе на ядро AVR с компилятором WinAVR программа перестанет работать, вернее, она будет работать так быстро, что управляемая микросхема не будет успевать обрабатывать сигналы из-за того, что все вызовы функции Delay будут упразднены.

Чтобы этого не происходило, нужно использовать квалификатор `volatile`:

```
void Delay (char D)
{
    volatile char i;
    for (i = 0; i < D; i++) continue;
}
```

2.1.3. Указатели на volatile

Также распространенной ошибкой является использование обычного (не `volatile`) указателя на `volatile` -переменную. Чем это может нам навредить? Рассмотрим пример (из реальной программы), в котором был использован указатель на регистр порта ввода/вывода. Программа по SPI управляла двумя одинаковыми микросхемами, подключенными на разные выводы микроконтроллера, порт, к которому подключена активная микросхема, выбирался через указатель:

```
unsigned int *Port;

void SPI_Send (unsigned char Data)
{
    char i = 8;

    do
    {
        if (Data & 0x80) *Port |=  1;      // Set data bit
        else             *Port &= ~1;

        *Port |=  2;                       // Make clock pulse
        Data <<= 1;                        // Shift data
        *Port ^= ~2;
    } while (--i);

}

void main (void)
{
    //...
    Port = &PORTB;
    //...
    SPI_Send(0x55);
    SPI_Send(0x66);
    //...

}
```

Произошла та же ситуация, что и в предыдущем примере: под одним компилятором (MCC18) код работал, а под другим (MCC30) - перестал. Причина крылась в неправильно объявленном указателе. Дизассемблер функции SPI_Send() выглядел так:

```
0050C   SPI_Send:       mov.w   port,w1
0050E                   mov.b   #0x8,w2         ;    i = 8
00510   While:          cp0.b   w0,#0           ;      if (Data & 0x80)
00512                   bra     ges, 0x000518
00514                   bset    [w1],#0         ;        *Port |=  1
00516                   bra     0x00051a
00518                   bclr    [w1],#0         ;        *Port &= ~1
                                                ;        *Port |=  2 ß----
0051A                   add.b   w0,w0,w0        ;        Data <<= 1
0051C                   bclr    [w1],#1         ;        *Port &= ~2
0051E                   dec.b   w2,w2           ;    while (--n)
00520                   bra nz, 0x000510
00522                   return
```

Обратим внимание на то, что компилятор "выкинул" установку бита 1 ("*Port |= 2"), посчитав ее лишней, т.к. почти сразу же за ней этот бит снова обнуляется, т.е. он выполнил оптимизацию без учета особенностей регистра, на который указывала переменная Port, а сделал он так потому, что программист не объяснил компилятору, что регистр непростой. Для исправления ошибки нужно было объявить переменную Port как указатель на `volatile` переменную:

```
volatile unsigned int *Port;

void SPI_Send (unsigned char Data)
{
    ...
}
```

Теперь компилятор знает, что оптимизацию над переменной, на которую указывает Port, производить нельзя. И новый листинг это отражает:

```
   0050C   SPI_Send:        mov.w   port,w1
   0050E                    mov.b   #0x8,w2          ;    i = 8
   00510   While:           cp0.b   w0               ;      if (Data & 0x80)
   00512                    bra     ges, 0x000518
   00514                    bset    [w1],#0          ;        *Port |=  1
   00516                    bra     0x00051a
   00518                    bclr    [w1],#0          ;        *Port &= ~1
   0051A                    bset    [w1],#1          ;        *Port |=  2 <-----
   0051C                    add.b   w0,w0,w0         ;        Data <<= 1
   0051E                    bclr    [w1],#1          ;        *Port &= ~2
   00520                    dec.b   w2,w2            ;    while (--n)
   00522                    bra nz, 0x000510
   00524                    return
```

2.1.4. Аргумент функции

Если бы в функцию SPI_Send из предыдущего примера нужно было бы передавать адрес порта (т.е. не держать его в глобальной переменной, а передавать в качестве аргумента), то не нужно забывать, что сам аргумент функции должен быть также описан с квалификатором `volatile`:

```
void SPI_Send (unsigned char Data, volatile unsigned int *Port)
{
   ...
}


void main (void)
{
    //...
    SPI_Send(0x55, &PORTB);
    SPI_Send(0x66, &PORTB);
    //...
}
```

В противном случае мы получим все те же ошибки, что и в предыдущем примере.

2.2. Неправильное использование volatile

2.2.1. volatile != атомарность

Существует некоторое заблуждение насчет защищенности переменных, объявленных как `volatile`. Т.е. программист, используя `volatile` - переменную, уверен, что компилятор сам позаботится о том, чтобы операции с переменной защищались запретами прерываний или еще какими-то хитрыми действиями. Однако, это не так. Объявление переменной как `volatile` совсем не гарантирует нам атомарность операций с ней. Тут программистов подстерегают две проблемы:

1. обращение к многобайтовой переменной;

2. чтение/модификация/запись через аккумулятор.

Рассмотрим пример обращения к переменной, занимающей более чем одну ячейку памяти (для 16-разрядных контроллеров это int32, int64, float, double; для 8-разрядных - еще и int16). Я часто приводил этот пример, приведу еще раз (для HT-PICC18):

```
volatile unsigned int ADCValue;     // Результат чтения АЦП (производится
                                    // в прерывании)

void interrupt isr (void)
{
    if (ADIF && ADIE)
    {
        ADIF     = 0;
        ADCValue = (ADRESH << 8) | ADRESL;  // Читаем последнее измерение
        ADGO     = 1;                       // Запускаем следующее
    }
}

void main ()
{
    //...
    while (1)
    {
        //...
        if (ADCValue < 100) Alarm();        // Сравнение напряжения с пороговыми
        if (ADCValue > 900) Alarm();        // значениями и вызов тревоги
        //...
    }
}
```

Обратим внимание, что переменная ADCValue имеет размерность 2 байта (для хранения 10-битного результата АЦП). Чем опасен данный код? Рассмотрим листинг одного из сравнений (допустим, первого):

```
;                   if (ADCValue < 100) Alarm();
    MOVLW    0
    SUBWF    ADCValue + 1, W
    MOVLW    100
    BTFSC    STATUS, Z
    SUBWF    ADCValue, W
    BTFSS    STATUS, C
    RCALL    Alarm
```

Допустим, значение напряжения на входе АЦП такое, что результат преобразования равен 255 (0x0FF). И последнее значение переменной ADCValue, соответственно, тоже = 0x0FF. С этим значением начинает выполняться код сравнения со значением 100 (0x064). Сначала сравниваются старшие байты переменной и константы (0x00 с 0x00), а затем - младшие (0x64 и 0xFF). Результат, казалось бы, очевиден. Однако здесь кроется неприятность. Хоть результат АЦ-преобразования и равен 0xFF, на него влияют несколько факторов: стабильность напряжения питания (или опорного напряжения), стабильность входного сигнала, близость уровня измеряемого напряжения к порогу смены единицы младшего разряда, наводки, шумы и пр. Поэтому результат АЦ-преобразования имеет некий джиттер в одну-две единицы младшего разряда. Т.е. результат АЦП может скакать между значениями 0xFF и 0x100. И если между выполнением сравнений возникнет прерывание, то может произойти следующее:

1. значение ADCValue = 0x0FF;
2. произведено сравнение старших байтов: 0x00 и 0x00;
3. возникло прерывание по ADIF, в котором значение переменной ADCValue обновилось на 0x100;
4. производится сравнение младших байтов: 0x64 и уже 0x00!
5. т.к. программа думает, что было сравнение 0x000 < 0x064, то она вызывает функцию Alarm.

И квалификатор `volatile` здесь не спасает. Здесь спасет только запрет прерываний на время выполнения сравнений.

```
        ADIE = 0;
        if (ADCValue < 100) { ADIE = 1;  Alarm();}
        if (ADCValue > 900) { ADIE = 1;  Alarm();}
        ADIE = 1;
...

void Alarm (void)
{
    ADIE = 1;       // Не забыть включить прерывание при выполнении условий
    ...
```

Так может быть, `volatile` вообще не нужен? Прерывания-то все равно запрещаются? Да, прерывания запрещаются, но `volatile`, все-таки нужен. Зачем? Рассмотрим почти такой же код для компилятора C30:

```
unsigned int ADCValue;

void main (void)
{
    //...
    Temp = ADCValue;

    while (1)
    {
        //...
        IEC0bits.ADIE = 0;
        if (ADCValue < 100) { IEC0bits.ADIE = 1; Alarm();}
        if (ADCValue > 900) { IEC0bits.ADIE = 1; Alarm();}
        IEC0bits.ADIE = 1;
        //...
    }
}
```

Вот здесь-то volatile и пригодится! Обратим внимание на строчку перед вечным циклом - присваивание значения переменной ADCValue переменной Temp. А заодно посмотрим листинг:

```
00536      mov.w    ADCValue, w1
00538      mov.w    w1, Temp
;                        while (1)
;                        {
;                           IEC0bits.AD1IE = 0;
0053A      bclr.b   0x0095,#5
;                        if (ADCValue < 100) { IEC0bits.AD1IE =1; Alarm();}
0053C      mov.w    #0x63,w0
0053E      sub.w    w1,w0,[w15]
00540      bra      leu, 0x000548
;                        if (ADCValue > 900) { IEC0bits.AD1IE =1; Alarm();}
00542      mov.w    #0x384,w0
00544      sub.w    w1,w0,[w15]
00546      bra      leu, 0x00054c
00548      bset.b   0x0095,#5
0054A      rcall    Alarm
;                        IEC0bits.AD1IE = 1;
0054C      bset.b 0x0095,#5
```

Как видим, внутри цикла вообще нет обращения к переменной ADCValue, а вместо этого сравнение производится с регистром W1, куда была скопирована переменная ADCValue еще перед циклом. Поэтому, как ни будет изменяться ADCValue, наша программа этого не заметит. Так что `volatile` в данном случае нужен обязательно, просто не следует забывать, что этот квалификатор не гарантирует нам атомарности операций над объявленной переменной.

2.2.2. volatile указатели

Иногда бывает такое, что сам указатель на какую-то переменную должен быть защищен от оптимизации (например, есть его модификация в теле

прерывания). Часто программисты совершают одну и ту же ошибку, т.е. при объявлении такого указателя ставят `volatile` не туда, куда нужно:

```
volatile char *p;
или
char volatile *p;
```

Обе записи идентичны, но они не делают саму переменную p `volatile` -переменной. Обе записи означают "переменная p является указателем на `volatile` char". Последствия такого определения очевидны: при изменении значения указателя в прерывании или в параллельной задаче программа этого может не заметить, т.к. будет работать с указателем через РОН.

Правильное определение выглядит так:

```
char * volatile p;
```

Если же нужен `volatile` -указатель на `volatile` -переменную, то он объявляется так:

```
volatile char * volatile p;
```

2.3. Использование volatile не к месту

Скажу два слова об этой проблеме. У некоторых однажды нарвавшихся на проблемы, связанные с отсутствием `volatile` там, где он должен был быть, появляется какой-то комплекс `volatile` -мании (или оптимизация-фобии), когда в страхе, что *"а вдруг компилятор этот код выкинет"*, чуть ли не все переменные объявляются с квалификатором `volatile`. В принципе, ничего криминального в таком использовании `volatile` нет, код будет работать правильно, но он будет громоздкий и неповоротливый. `volatile` не позволит компилятору применять оптимизацию, и некоторые фрагменты, которые, будучи оптимизированы, выполнились бы за 50-100 тактов, будут выполняться за 1000-2000 тактов. То же самое касается объемов кода (в десятки раз он, конечно, не вырастет, а вот раза в два - запросто). Другими словами появится неудовлетворенность оттого, что *"и контроллер быстрый, и компилятор хороший, а код все равно тормозит"*.

Здесь четких рекомендаций я дать не смогу. В большинстве случаев при написании программ на Си я стараюсь придерживаться следующих правил:

- глобальные переменные, используемые и в прерываниях, и в программе (или в прерываниях различных приоритетов), нужно объявлять как `volatile`;

- глобальные переменные, обрабатываемые двумя и более задачами при работе под многозадачной ОС, нужно объявлять как `volatile`;

- указатели на периферийные регистры, а также на переменные, объявленные как `volatile`, нужно объявлять как указатели на `volatile`;

- все, что не попадает под первые три правила и не связано с периферией, рекомендуется писать, абстрагируясь от железа. Тогда становится понятно, что циклы вида "for (i = 0; i<100; i++) {};" не несут на себе никакой алгоритмической нагрузки и могут быть удалены. А если они должны быть оставлены, то переменные следует объявлять как `volatile`.

- во всех остальных случаях `volatile` будет лишним.

3. Дополнительно

Еще несколько замечаний:

volatile можно ставить в любое место в типе

Как уже указывалось выше, следующие записи эквивалентны:

```
volatile static int N;
static volatile int N;
static int volatile N;
```

Лично я для наглядности `volatile` выношу вперед.

volatile и typedef

`volatile` может быть использован при определении типов по всем правилам:

```
typedef volatile char vchar;
```

Теперь все переменные типа vchar будут `volatile` -переменными.

volatile и структуры

`volatile` может быть применен как ко всей структуре целиком, так и к отдельным ее полям. В зависимости от этого компилятор будет производить оптимизацию по-разному. Например, в этой структуре все поля будут `volatile`:

```
typedef volatile struct
{
    unsigned char  Counter;
    unsigned char *Buffer;
} T_BUFFER;
```

(Обращу ваше внимание на то, что здесь Buffer является `volatile` - указателем на не- `volatile` переменную. Т.е. эквивалент определения: "**unsigned char * volatile Buffer**", а не "**volatile unsigned char *Buffer**".)

А в этой - только счетчик:

```
typedef struct
{
     volatile unsigned char  Counter;
              unsigned char *Butter;
} T_BUFFER;
```

volatile и extern

Если переменная во внешнем модуле объявлена как `volatile`:

```
volatile int Counter;
```

, то и в заголовочном файле она должна быть объявлена с квалификатором `volatile`:

```
extern volatile int Counter;
```

Виктор Тимофеев, июнь, 2010

osa@pic24.ru