ADC

Обеспечение целостности данных в прерываниях

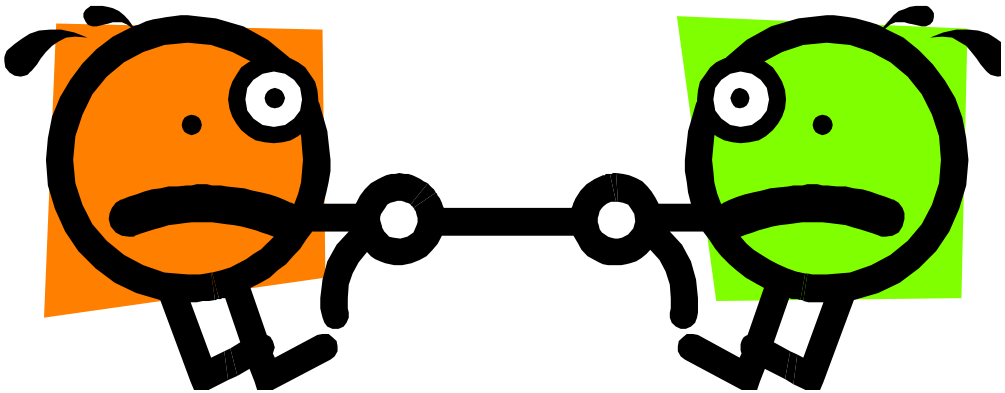
Темы

- Совместное использование данных
- Модификатор volatile
- Атомарные операции
- Разрушение данных
- Методы защиты данных
- Методы доступа к данным
- Заключение

Задачи

- Понять почему требуется использование модификатора volatile для переменных, разделяемых основной программой и программами обработки прерываний
- Понять что такое атомарные операции над переменными
- Изучить методы для обеспечения безопасности совместно используемых данных

Совместное использование данных

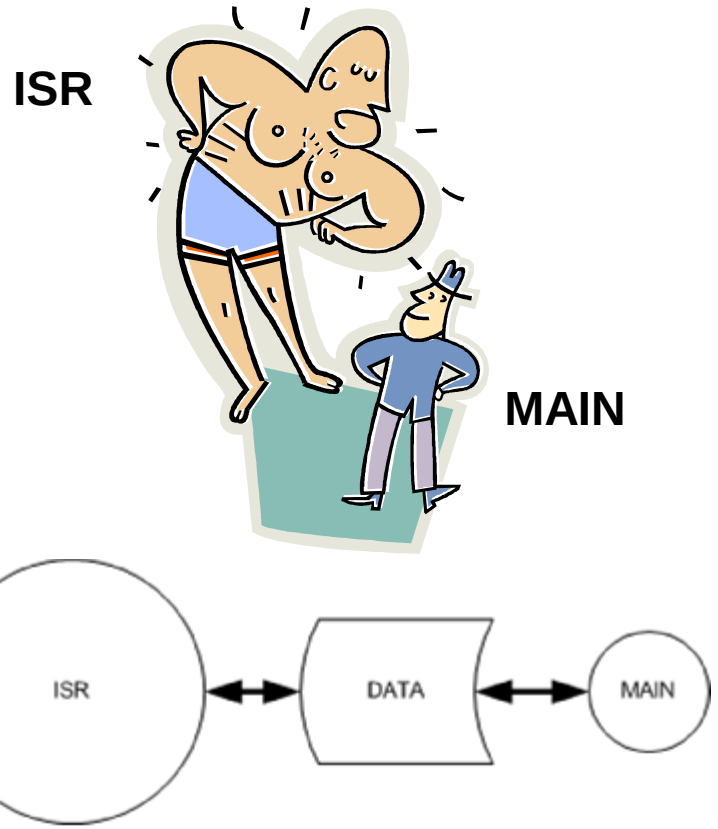


Разделяемые переменные non-ISR <-> ISR

Исходный код

```
#include <p24fxxxx.h>

#define FALSE   (0)
#define TRUE    (!FALSE)

unsigned int systemTimeMs;

int main(void)
{
    char    timeout;

    timeout = FALSE;
    systemTimeMs = 0;
    // setup T1 to 1ms and enable
    do
    {
        if (0x02FF <= systemTimeMs)
        {
            timeout = TRUE;
        }
    } while (!timeout);
    return(timeout);
}

void __attribute__((interrupt, auto_psv))
       _T1Interrupt(void)
{
    _T1IF = 0;
    systemTimeMs++;
}
```

Иллюстрация



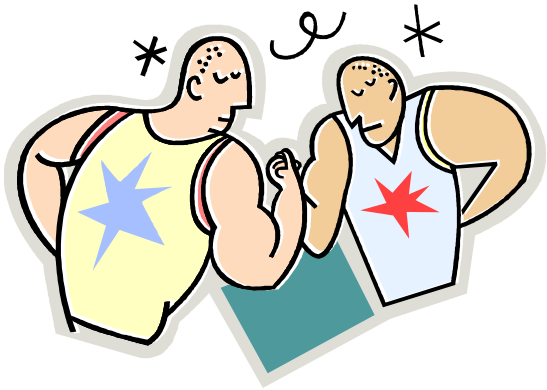
```c
#include <p24fxxxx.h>

#define FALSE   (0)
#define TRUE    (!FALSE)

unsigned int systemTimeMs;

int main(void)
{
    char    timeout;

    timeout = FALSE;
    systemTimeMs = 0;
    // setup T1 to 1ms and enable
    do
    {
        if (0x02FF <= systemTimeMs)
        {
            timeout = TRUE;
        }
    } while (!timeout);
    return(timeout);
}

void __attribute__((interrupt, auto_psv)) _T1Interrupt(void)
{
    _T1IF = 0;
    systemTimeMs++;
}
```

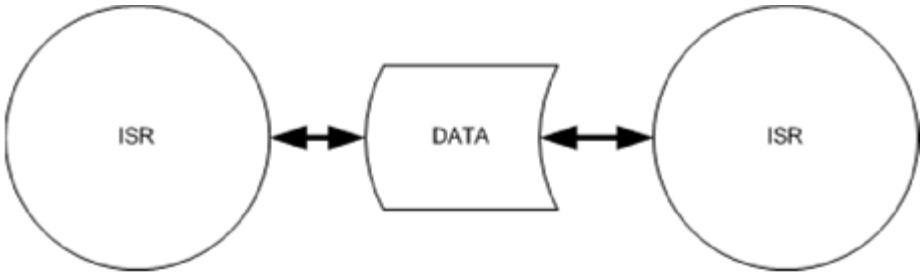
Разделяемые переменные ISR <-> ISR

Исходный код

```
#include <p24fxxxx.h>

int             integral = 0;
unsigned int    targetSpeed = 0x0400;
unsigned int    powerOut = 0;
unsigned int    avgCum = 0;

int main (void)
{
    // setup ADC
    // setup T1 at integral period x
    while(1){}
}

// PID
void __attribute__((interrupt, auto_psv))
      _T1Interrupt(void)
{
    unsigned int error;

    _T1IF = 0;
    error = (targetSpeed - (avgCum >> 4));
    integral += error;
    powerOut = (5 * error) + (2 * integral);
}

// ADC Speed Sample
void __attribute__((interrupt, auto_psv))
      _ADC1Interrupt(void)
{
    unsigned int rawADC;

    _AD1IF = 0;
    rawADC = ADC1BUF0;
    avgCum = ((avgCum - (avgCum >> 4)) + rawADC);
}
```

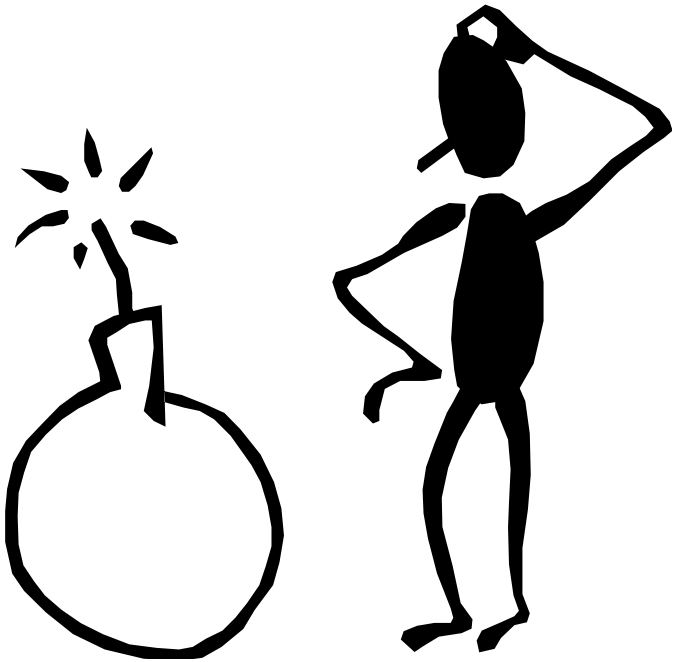
Иллюстрация





```
#include <p24fxxxx.h>

int             integral = 0;
unsigned int    targetSpeed = 0x0400;
unsigned int    powerOut = 0;
unsigned int    avgCum = 0;

int main (void)
{
    // setup ADC
    // setup T1 at integral period x
    while(1){}
}
// PID
void __attribute__((interrupt, auto_psv)) _T1Interrupt(void)
{
    unsigned int error;

    _T1IF = 0;
    error = (targetSpeed - (avgCum >> 4));
    integral += error;
    powerOut = (5 * error) + (2 * integral);
}
// ADC Speed Sample
void __attribute__((interrupt, auto_psv)) _ADC1Interrupt(void)
{
    unsigned int rawADC;

    _AD1IF = 0;
    rawADC = ADC1BUF0;
    avgCum = ((avgCum - (avgCum >> 4)) + rawADC);
}
```

Модификатор Volatile



Определение

Модификатор volatile используется для указания компилятору, что данная переменная может быть изменена сторонним процессом, за пределами области определения данной функции.

Компилятор должен при каждом обращении к переменной (чтение или запись) в исходном коде обеспечить выполнение соответствующей операции (загрузки или сохранения) для области памяти, которая выделена для этой переменной.

Оптимизированный код без модификатора Volatile

Исходный код

```
#include <p24fxxxx.h>
#define FALSE   (0)
#define TRUE    (!FALSE)

unsigned int systemTimeMs;

int main(void)
{
    char    timeout;
    timeout = FALSE;
    systemTimeMs = 0;
    // setup T1 to 1ms and enable
    do {
        if (0x02FF <= systemTimeMs) {
            timeout = TRUE;
        }
    } while (!timeout);
    return(timeout);
}
```

```
void __attribute__((interrupt,
    auto_psv)) _T1Interrupt(void)
{
    _T1IF = 0;
    systemTimeMs++;
}
```

Оптимизированный код без модификатора Volatile

Дизассемблер

```
6:                  unsigned int  systemTimeMs;
7:                  int main(void)
8:                  {
9:                      char    timeout;
10:                     timeout = FALSE;
11:                     systemTimeMs = 0;
  0290  EF2800      clr.w systemTimeMs
  0292  37FFFF      bra 0x000292
12:                     // setup T1 to 1ms and enable
13:                     do {
14:                         if (0x02FF <= systemTimeMs) {
15:                             timeout = TRUE;
16:                         }
17:                     } while (!timeout);
18:                     return(timeout);
19:                 }
```

Оптимизированный код с модификатором Volatile

Дизассемблер

```
6:                  unsigned int volatile systemTimeMs;
7:                  int main(void)
8:                  {
9:                      char    timeout;
10:                     timeout = FALSE;
11:                     systemTimeMs = 0;
  0290  EF2800      clr.w systemTimeMs
  0292  202FE1      mov.w #0x2fe,w1
12:                     // setup T1 to 1ms and enable
13:                     do {
14:                         if (0x02FF <= systemTimeMs) {
  0294  804000      mov.w systemTimeMs,w0
  0296  500F81      sub.w w0,w1,[w15]
  0298  36FFFD      bra leu, 0x000294
15:                             timeout = TRUE;
16:                         }
17:                     } while (!timeout);
18:                     return(timeout);
19:                 }
  029A  200010      mov.w #0x1,w0
  029C  060000      return
```

Другие случаи применения модификатора volatile

Локальная переменная

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

Атомарная операция

```
volatile unsigned int ADCValue;
void main (void) {
  // . . .
  Temp = ADCValue;
  // . . .
  while (1) {
    IEC0bits.ADIE = 0;
    if (ADCValue < 100) {
      IEC0bits.ADIE = 1; Alarm();
    }
    IEC0bits.ADIE = 1;
  }
}
```

Указатель на volatile переменную

```
volatile unsigned int *Port;
void SPI_Send (unsigned char Data) {
  char i = 8;
  do {
     if (Data & 0x80) *Port |= 1;
     else             *Port &= ~1;
     *Port |= 2;
     Data <<= 1;
     *Port ^= ~2;
  } while (--i);
}

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

Особенности применения модификатора volatile

Переменная – volatile
Указатель – не volatile

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

Переменная – не volatile
Указатель – volatile

```
char * volatile p;
```

Переменная – volatile
Указатель – volatile

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

volatile тип

```
typedef volatile char volchar_t;
```

volatile структура

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

volatile поле структуры

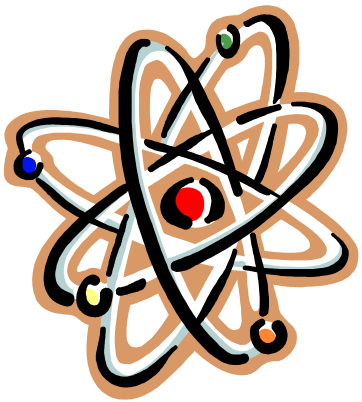
```
typedef struct {
  volatile unsigned char Counter;
  unsigned char *Butter;
} buffer_t;
```

Что модификатор Volatile делает, а что не делает

Модификатор volatile указывает компилятору учитывать возможность изменения переменной вне области определения данной функции.

Модификатор volatile НЕ требует от компилятора обеспечить защиту переменной при ее модификации при доступе к ней из различных частей программы. Т.е. этот модификатор НЕ обеспечивает атомарности операций над переменной. Более того, модификатор может даже ухудшить ситуацию из-за увеличения операций доступа к разделяемой переменной.

Атомарные операции

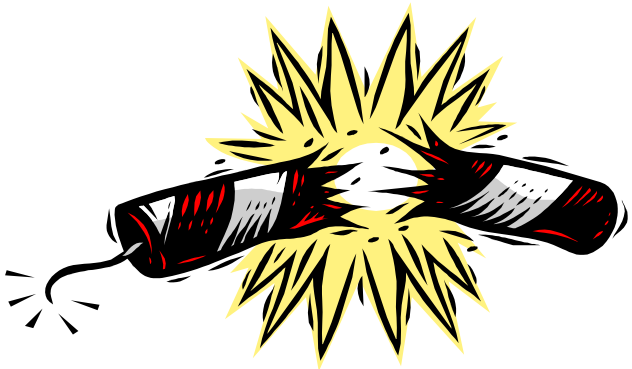


Что означает атомарность операции ?

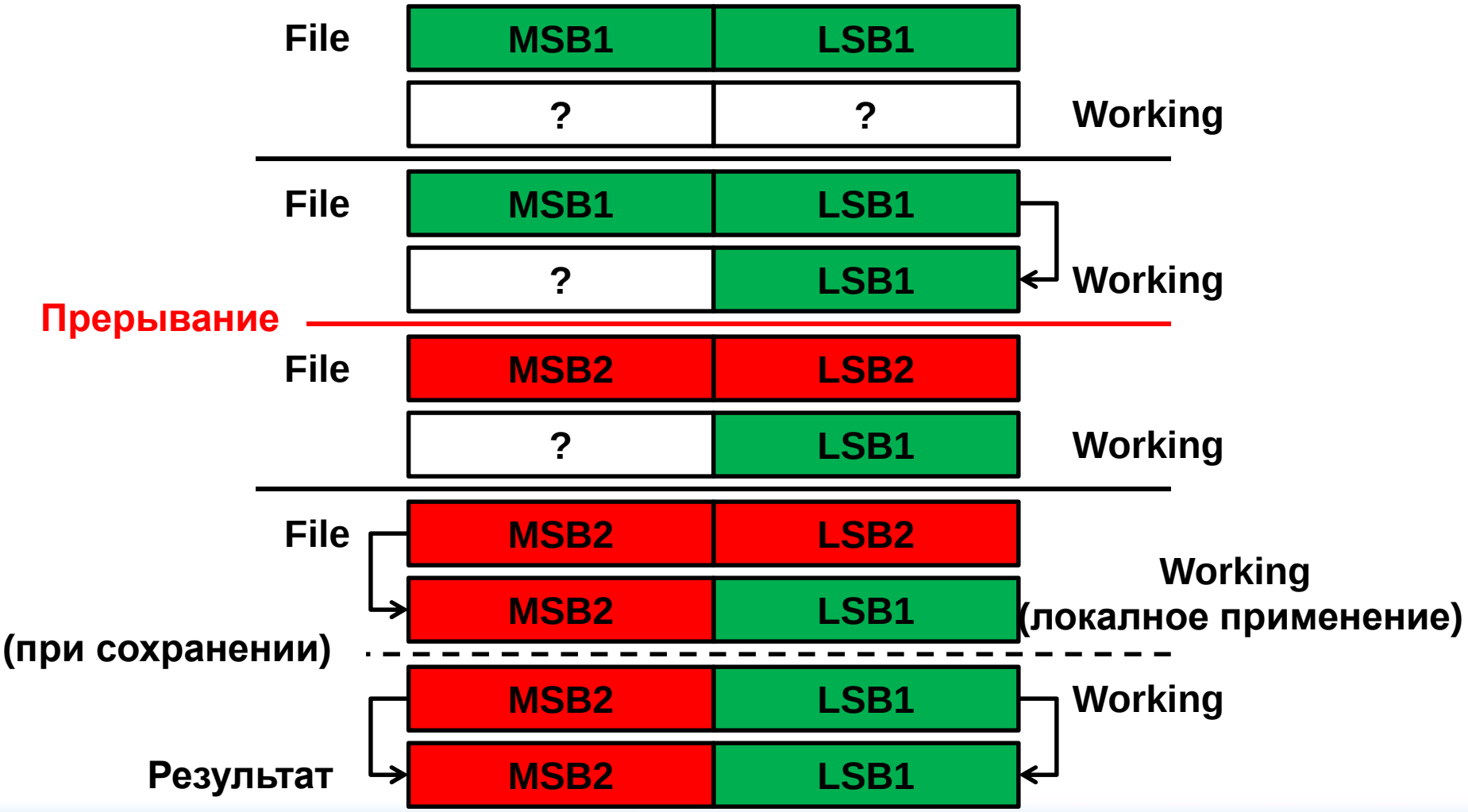
Операция считается атомарной, если она гарантирует непрерываемый доступ к данной переменной. На уровне ассемблера могут существовать специальные атомарные операции, однако, не существует способа автоматически обеспечить атомарность для всех типов переменных в языке ANSI C.

Для операций над переменными, чья размерность отличается от размерности данных микроконтроллера, невозможно гарантировать атомарность.

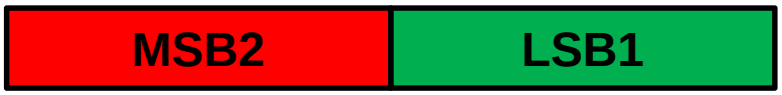
Разрушение данных



Пример разрушающей последовательности



Нарушение байтовой целостности данных



Результат

Корректная часть

Испорченная часть

Разделяемая переменная типа WORD

16-bit тип базовый для семейств 24/30/33 PIC® MCU

```
typedef struct tagSysTime
{
    unsigned char   hour;
    unsigned char   min;
    unsigned char   sec;
    unsigned int    systemTimeMs;
} SysTime;

SysTime volatile    sysTime;

int main(void)
{
    char    timeout;
    . . .
    do
    {
        if (0x02FF <= sysTime.systemTimeMs)
        {
            timeout = TRUE;
        }
    } while (!timeout);
    . . .
}
```

```
24:  if (0x02FF <= sysTime.systemTimeMs)
  0296  804021     mov.w 0x0804,w1
  0298  202FE0     mov.w #0x2fe,w0
  029A  508F80     sub.w w1,w0,[w15]
  029C  360002     bra leu, 0x0002a2
25: {
```

Разделяемая переменная типа PACKED WORD

16-bit тип базовый для семейств 24/30/33 PIC® MCU

```
typedef struct tagSysTime
{
    unsigned char   hour;
    unsigned char   min;
    unsigned char   sec;
    unsigned int    systemTimeMs;
} __attribute__((packed)) SysTime;

SysTime volatile    sysTime;

int main(void)
{
    char    timeout;
    . . .
    do
    {
        if (0x02FF <= sysTime.systemTimeMs)
        {
            timeout = TRUE;
        }
    } while (!timeout);
    . . .
}
```

```
24: if (0x02FF <= sysTime.systemTimeMs)
  0296  BFC803     mov.b 0x0803,w0
  0298  FB8080     ze w0,w1
  029A  BFC804     mov.b 0x0804,w0
  029C  FB8000     ze w0,w0
  029E  DD0048     sl w0,#8,w0
  02A0  700001     ior.w w0,w1,w0
  02A2  780080     mov.w w0,w1
  02A4  202FE0     mov.w #0x2fe,w0
  02A6  508F80     sub.w w1,w0,[w15]
  02A8  360002     bra leu, 0x0002ae
25: {
```

Разделяемая переменная типа PACKED WORD

Разрушение данных при инкременте 0x01FF -> 0x0200

```
typedef struct tagSysTime
{
    unsigned char   hour;
    unsigned char   min;
    unsigned char   sec;
    unsigned int    systemTimeMs;
} __attribute__((packed)) SysTime;

SysTime volatile    sysTime;
```

```
24: if (0x02FF <= sysTime.systemTimeMs)
  0296  BFC803     mov.b 0x0803,w0
  0298  FB8080     ze w0,w1
  029A  BFC804     mov.b 0x0804,w0
  029C  FB8000     ze w0,w0
  029E  DD0048     sl w0,#8,w0
  02A0  700001     ior.w w0,w1,w0
  02A2  780080     mov.w w0,w1
  02A4  202FE0     mov.w #0x2fe,w0
  02A6  508F80     sub.w w1,w0,[w15]
  02A8  360002     bra leu, 0x0002ae
25: {
```

1) Перед началом выполнения оператора Си переменная systemTimeMs имеет значение 0x01FF.
2) LSB переменной systemTimeMs загружается в W1, W1 = 0x00FF.
3) В этот момент возникает прерывание и значение systemTimeMs изменяется на 0x0200.
4) Возврат из прерывания.
5) MSB переменной systemTimeMs загружается в W0 и выравнивается, W0 = 0x0200.
6) W0 и W1 формируют значение 0x02FF, которое является неверным, т.к. должно быть 0x01FF.
7) Операция сравнения выполняется с некорректным значением

Разделяемая переменная типа LONG

Разрушение данных при инкременте 0x0001FF00 -> 0x00020000

```
unsigned long   volatile systemTimeMs;
```

```
24: if (0x0002FFFF <= systemTimeMs)
  0296  804002      mov.w systemTimeMs,w2
  0298  804013      mov.w 0x0802,w3
  029A  2FFFE0      mov.w #0xfffe,w0
  029C  200021      mov.w #0x2,w1
  029E  510F80      sub.w w2,w0,[w15]
  02A0  598F81      subb.w w3,w1,[w15]
  02A2  360002      bra leu, 0x0002a8
25: {
```

1) Перед началом выполнения оператора Си переменная systemTimeMs имеет значение 0x0001FFFF.
2) LSW переменной systemTimeMs загружается в W2, W2 = 0xFFFF.
3) В этот момент возникает прерывание и значение systemTimeMs изменяется на 0x00020000.
4) Возврат из прерывания.
5) MSW переменной systemTimeMs загружается в W3, W3 = 0x0002.
6) W3 и W2 формируют значение 0x0002FFFF, , которое является неверным, т.к. должно быть 0x0001FFFF.
7) Операция сравнения выполняется с некорректным значением

Разделяемая переменная типа STRUCT

Разрушение данных при инкременте 23:59:59.999 -> 00:00:00.00

```
typedef struct tagSysTime
{
    unsigned char   hour;
    unsigned char   min;
    unsigned char   sec;
    unsigned int      systemTimeMs;
} SysTime;
SysTime volatile    sysTime;
```

```
25:  timeStamp = sysTime;
  0296  804030     mov.w sysTime,w
  0298  804041     mov.w 0x0808,w1
  029A  884000     mov.w w0,timeStamp
  029C  884011     mov.w w1,0x0802
  029E  804051     mov.w 0x080a,w1
  02A0  884021     mov.w w1,0x0804
26: if (0x02FF <= sysTime.systemTimeMs)
  02A2  804051     mov.w 0x080a,w1
  02A4  202FE0     mov.w #0x2fe,w0
  02A6  508F80     sub.w w1,w0,[w15]
  02A8  360002     bra leu, 0x0002ae
```

1) Перед началом выполнения оператора Си переменная sysTime начинает со значения 23:59:59.999.
2) sysTime.hrs и sysTime.mins загружаются в W0, W0 = 23:59.
3) В этот момент возникает прерывание и значение sysTime изменяется на 00:00:00.000.
4) Возврат из прерывания.
5) sysTime.sec загружается в W1, W1 = 00.
6) W0 срхраняется в timeStamp.hrs = 23 и timeStamp.mins = 59, что корректно
7) W1 сохраняется в timeStamp.sec = 00, что некорректно.
8) sysTime.systemTimeMs сохраняется в W1, W1 = 000.
9) W1 сохраняется как timeStamp.systemTimeMs = 000, что некорректно.
10) Если timeStamp используется для запоминания времени события, то запомнится 23:59:00.000, что неверно.

Разделяемая переменная типа PACKED STRUCT

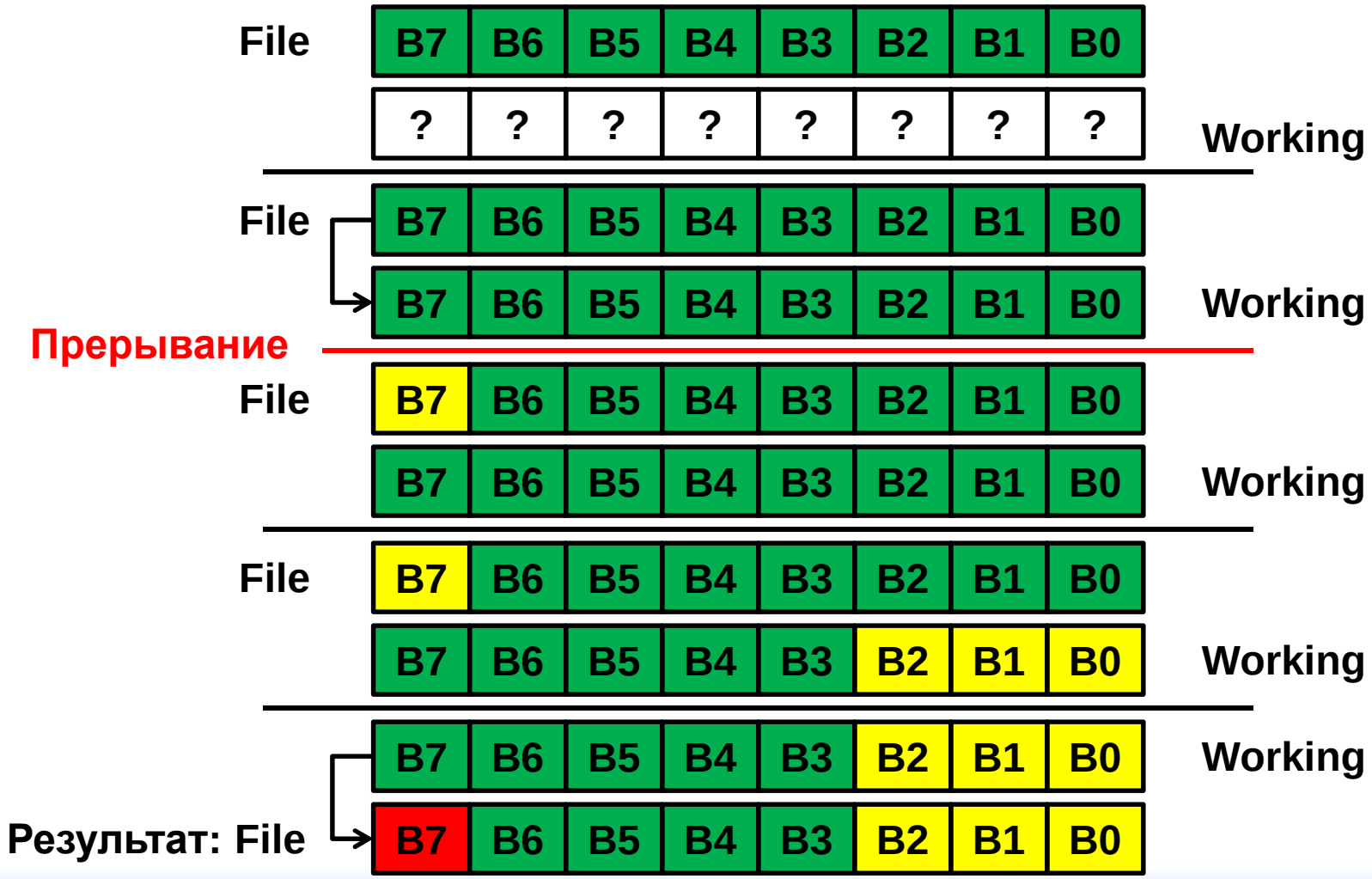
Побайтное разрушение данных

```
typedef struct tagSysTime
{
    unsigned char   hour;
    unsigned char   min;
    unsigned char   sec;
    unsigned int      systemTimeMs;
} __attribute__((packed)) SysTime;
```

```
25:  timeStamp = sysTime;
  0296  208000     mov.w #0x800,w0
  0298  208061     mov.w #0x80C,w1
  029A  090004     repeat #4
  029C  785831     mov.b [w1++],[w0++]
26: if (0x02FF <= sysTime.systemTimeMs)
  029E  BFC809     mov.b 0x0809,w0
  02A0  FB8080     ze w0,w1
  02A2  BFC80A     mov.b 0x080a,w0
  02A4  FB8000     ze w0,w0
  02A6  DD0048     sl w0,#8,w0
  02A8  700001     ior.w w0,w1,w0
  02AA  780080     mov.w w0,w1
```

1) Перед началом выполнения оператора Си переменная sysTime начинает со значения XX:XX:XX.XX.
2) В W0 загружается адрес структуры timeStamp.
3) В W1 загружается адрес структуры sysTime.
4) Содержимое sysTime копируется в timeStamp при помощи 5 повторяемых побайтных операций mov.b..
5) Прерывание после 1, 2, 3 или 4 шага приведет к неправильному содержимому структуры timeStamp от данного байта и до конца.

Битовое разрушение данных



Битовое разрушение данных

РЕЗУЛЬТАТ

- Верное состояние
- Правильное изменение
- Разрушение

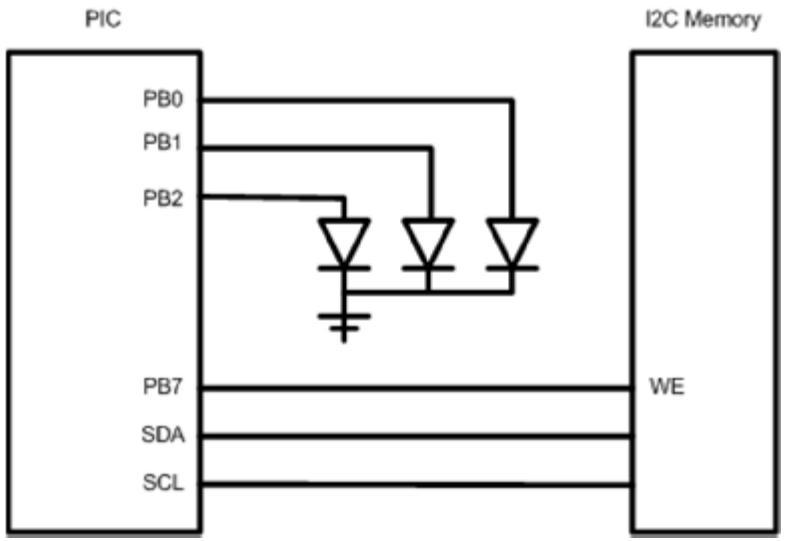
Битовое разрушение данных

```
typedef struct tagSensorChannel
{
    unsigned int    channel       :3;
    unsigned int    polarity      :1;
    unsigned int    sampleNow     :1;
    unsigned int    sensorFailure :1;
    unsigned int    numSamples    :4;
    unsigned int    ADCResult;
} SensorChannel;
SensorChannel volatile    sensor
```

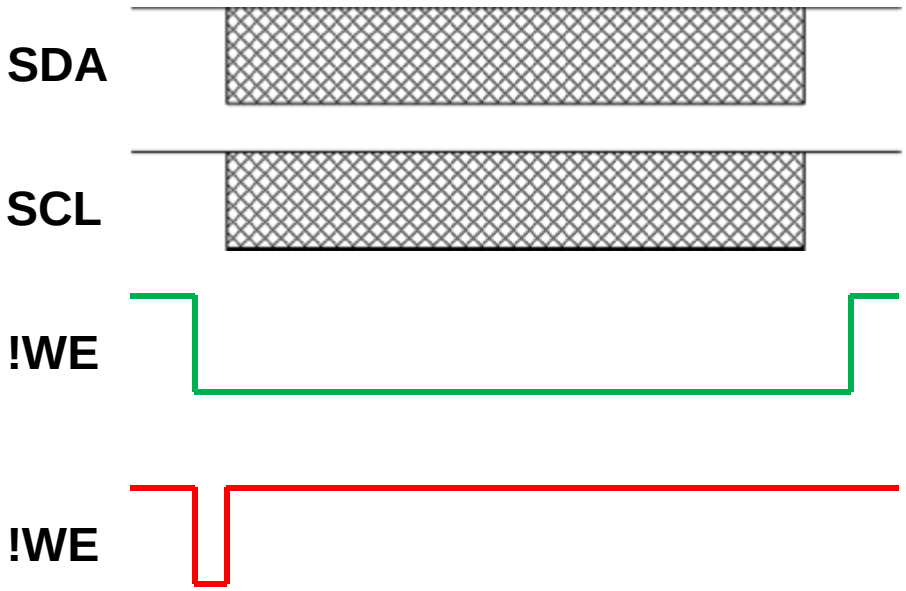
```
23:  sensor.channel = 6;
  0294  208001     mov.w #0x800,w1
  0296  784091     mov.b [w1],w1
  0298  B3CF80     mov.b #0xf8,w0
  029A  60C000     and.b w1,w0,w0
  029C  B34060     ior.b #0x6,w0
  029E  B7E800     mov.b w0,sensor
24:  if(sensor.sampleNow)
  02A0  BFC800     mov.b sensor,w0
  02A2  604070     and.b w0,#16,w0
  02A4  E00400     cp0.b w0
  02A6  32FFF6     bra z, 0x000294
```

1) Перед началом выполнения оператора Си sensor.sampleNow = FALSE.
2) Т.к. необходимо присвоить полю sensor.channel значение 6, в W1 загружается байт, содержащий sensor.channel. В этом же байте хранится и поле sensor.sampleNow.
3) В этот момент возникает прерывание и поле sensor.sampleNow становится TRUE.
4) Возврат из прерывания.
5) Маской 0xF8 биты W1, отвечающие за sensor.channel устанавливаются в 0, оставляя другие биты этого байта нетронутыми
6) Значение 6 сохраняется только в тех битах, которые отвечают за sensor.channel. Другие биты этого байта не изменяются
7) Весь байт сохраняется в структуре sensor. При этом остальные биты сохраняют то значение, которое было в шаге 2. Т.е sensor.sampleNow возвращается в FALSE, что неверно

Разрушение данных в PORT/SFR



Разрушение данных в PORT/SFR



Разрушение данных в PORT/SFR

Исходный код

```
#include <p24fxxxx.h>
#define LED_NT1         LATBbits.LATB0
#define LED_NT2         LATBbits.LATB1
#define LED_LINK        LATBbits.LATB2
#define I2C_WENABLE     LATBbits.LATB7


void setLedStateDriver(
     unsigned char ledToSet,
     unsigned char stateToSet);


int main(void)
{
     // setup I2C and enable
     I2C_WRITE_ENABLE = 1;
     setLedStateDriver(0,0);
     do
     {
          setLedStateDriver(0,1);
     } while (1);
}
```

```
void setLedStateDriver(unsigned char
ledToSet, unsigned char stateToSet) {
     switch(ledToSet){
          case 0:
               LED_LINK = stateToSet;
               break;
          case 1:
               LED_NT1 = stateToSet;
               break;
          case 2:
               LED_NT2 = stateToSet;
               break;
          default:
               break;
     }
     return;
}

void __attribute__((interrupt, auto_psv))
_MI2C1Interrupt(void)

{
     I2C_WRITE_ENABLE = 0;
}
```

Разрушение данных в PORT/SFR

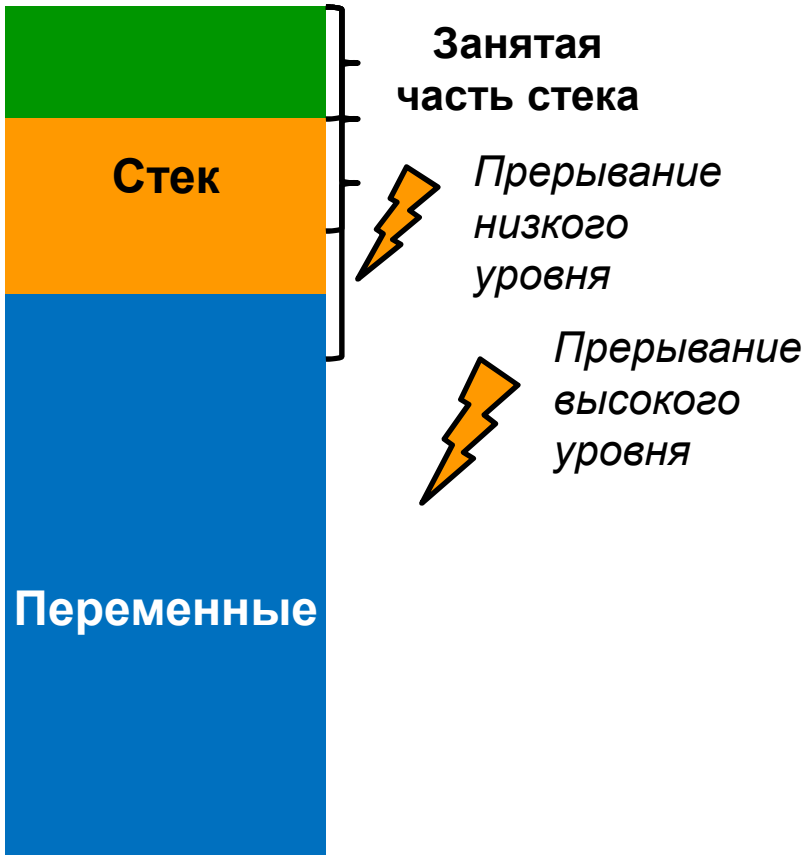
Дизассемблер

```
22: void setLedStateDriver(unsigned
    char ledToSet, unsigned char
    stateToSet) {
23:     switch(ledToSet) {
  029E  504FE1     sub.b w0,#1,[w15]
            . . .
  02A8  37000D     bra 0x0002c4
24:         case 0:
25:             LED_LINK = stateToSe
  02AA  60C0E1     and.b w1,#1,w1
  02AC  DD08C2     sl w1,#2,w1
  02AE  BFC2CC     mov.b 0x02cc,w0
  02B0  A12400     bclr w0,#2
  02B2  704001     ior.b w0,w1,w0
  02B4  B7E2CC     mov.b w0,0x02cc
  02B6  37000C     bra 0x0002d0
26:             break;
27:         case 1:
28:             LED_NT1 = stateToSet;
  02B8  60C0E1       and.b w1,#1,w1
             . . .
```

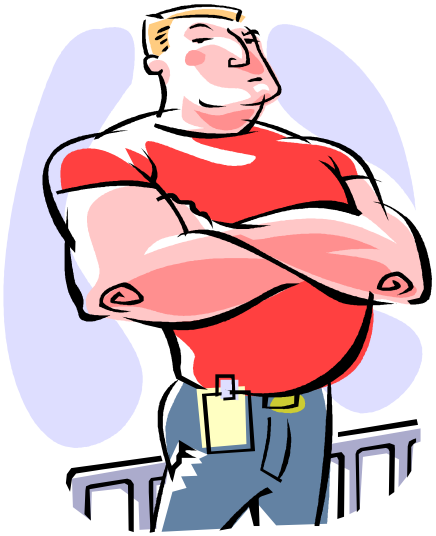
1) Перед началом выполнения оператора Си LATB7 = 1.
2) W1 содержит stateToSet. Значение маски 0b00000X00 генерируется для бита LATB2 и сохраняется в W1.
3) LSB регистра LATB, включая LATB7, считывается и сохраняется в W0.
4) В этот момент возникает прерывание и LATB7 устанавливается в 0 для разрешения записи в I2C память.
5) Битовая позиция для LATB2 в W0 сбрасывается. Остальные биты не меняются.
6) Операция OR W0 с W1, устанавливает значение stateToSet в позицию LATB2. Остальные биты не меняются.
7) W0 сохраняется в LSB регистра LATB, устанавливая stateToSet на вывод LATB2. Однако, при этом записываются все биты W0, в том состоянии, которое было на шаге 3.
8) Таким образом, LATB7 устанавливается в 1 и запись в I2C память становится невозможной.

Разрушение данных при переполнении стека

- **Каждый уровень прерывания или рекурсии сохраняет следующие данные в стек:**
  - контекст
  - локальные переменные
  - аргументы (для функций)
- **Размер стека всегда ограничен !!!**
- **Переполнение стека приводит к разрушению других данных**



Методы защиты данных



Полный запрет прерываний (IPL 7, GIE)

```
void protectedMemcpy(void *destination,
                     const void *source, size_t num)
{
    int volatile oldIpl;

    oldIpl = SRbits.IPL;
    SRbits.IPL2 = 1;        // Если здесь возникнет прерывание,
    SRbits.IPL1 = 1;        // биты IPL сохраняются и
    SRbits.IPL0 = 1;        // восстанавливаются из стека
    memcpy(destination, source, num);
    SRbits.IPL = oldIpl;
}
```

Полный запрет прерываний

- Обеспечивает гарантию атомарной операции
- Работает для всех уровней пользовательских прерываний
- Не работает для TRAPS
- Запрещает все прерывания на все время выполнения атомарных операций, что может повлиять на критические ко времени прерывания.

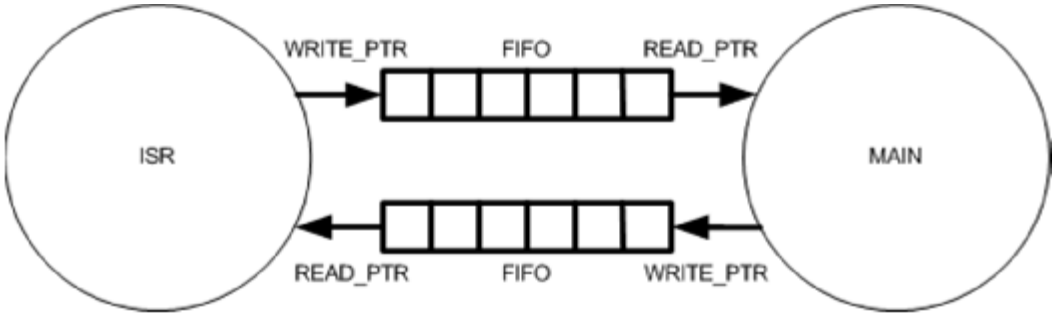
Индивидуальный запрет прерываний

```
if(1 == _T1IE)
{
    _T1IE = 0;
    timeMs = systemTimeMs;
    _T1IE = 1;
}
else
{
    timeMs = systemTimeMs;
}
```

Индивидуальный запрет прерываний

- Запрещает только отдельные прерывания, позволяя другим прерываниям оставаться работоспособными
- Написание кода для обработки прерывания требует хорошего знания работы остальных частей кода.
- Может возникнуть проблема восстановления прерывания при работе из разных модулей

Очереди



Использование XOR для изменения отдельных бит

- Операция AND не может перевести бит 0 в бит 1
- Операция OR не может перевести бит 1 в бит 0
- Операция XOR может установить любое состояние бита за одну инструкцию.

```
; Исходно  TRISB = 0xFFFF, Требуется безопасно записать 0xA в 4 младших бита
mov.w TRISB, W0        ; Загружает значение TRISB в регистр W0;
xor.w W0, #0x000A, W0  ; Атомарно и безопасно модифицирует
                       ; четыре младших бита регистра W0;
and.w W0, #0x000F, W0  ; Накладывает на регистр W0 маску, выделяя 4 младших бита
xor.w TRISB            ; Атомарно и безопасно (не трогая остальных битов!)
                       ; модифицирует TRISB; TRISB = 0xFFFA;
```

```
__asm__ __volatile__ "xor%0":"=U"(r):"a"(fieldMask & (newValue ^ currentValue)));
```

- www.avix-rt.com «AvixAtomicSFRManipulation»
- www.pic24.ru/doku.php/articles/mchp/c30_atomic_access

Использование аппаратных атомарных возможностей

- DISI – запрет прерываний на заданное количество инструкций (16 бит PIC)
- BTG – атомарное манипулирование единичным битом (16 бит XC)
- LL,SC,LLD,SCD – атомарные операции над кэшированными ячейками памяти (32 бит PIC)

- Требует ассемблерных вставок
- Не для всех типов микроконтроллеров
- DISI Не работает для IPL = 7
- DISI требует знания количества защищаемых циклов
- Errata

Способы доступа к переменным

Модификатор Extern

- Используется для предоставления доступа к переменной из различных Си модулей.
- Неограниченный прямой доступ к памяти, включая работу с указателями.
- Отсутствует проверка ограничений и валидности содержимого.
- Отсутствует информация о переменной и ее состоянии (инициализация).
- Требует защиты доступа переменной в каждом месте, где такой доступ осуществляется.
- Язык Си требует обеспечения соответствия типов в разных модулях.
- Исключает возможность оптимизации разделяемых Extern переменных в ISR.
- Часто рассматривается как локальная переменная, т.е. может несколько раз использоваться в уравнениях.

Модификатор Extern

Main.c

```
extern unsigned int volatile avgCum;

int main (void)
{
while(1){}
}

// PID, IPL = 3
void __attribute__((interrupt,
    auto_psv)) _T1Interrupt(void)
{
    unsigned int error;
    _T1IF = 0;
    error = (targetSpeed - (3 *
        avgCum) - (avgCum >> 1));
    integral += error;
    powerOut = (5 * error) + (2 *
        integral);
}
```

Adc.c

```
unsigned int volatile    avgCum = 0;

// ADC Speed Sample, IPL = 4
void __attribute__((interrupt,
    auto_psv)) _ADC1Interrupt(void)
{
    unsigned int rawADC;

    _AD1IF = 0;
    rawADC = ADC1BUF0;
    avgCum = ((avgCum - (avgCum >>
        4)) + rawADC);
}
```

Модификатор Extern

Main.c

```
extern unsigned int volatile avgCum;

int main (void)
{
while(1){}
}

// PID, IPL = 3
void __attribute__((interrupt,
    auto_psv)) _T1Interrupt(void)
{
    unsigned int error;
    _T1IF = 0;
    error = (targetSpeed - (3 *
        avgCum) - (avgCum >> 1));
    integral += error;
    powerOut = (5 * error) + (2 *
        integral);
}
```

Дизассемблер

```
21: error = (targetSpeed - (3 *
        avgCum) - (avgCum >> 1));
  029E  804032  mov.w avgCum,w2
        // First Value Used
  02A0  B91163  mul.su w2,#3,w2
        // Interrupt
  02A2  804010  mov.w targetSpeed,w0
  02A4  500102  sub.w w0,w2,w2
  02A6  D50806  lsr.w avgCum,w0
        // Second Value Used
  02A8  510100  sub.w w2,w0,w2
22: integral += error;
  02AA  804000  mov.w integral,w0
  02AC  410000  add.w w2,w0,w0
        . . .
```

Доступ через функции

- Обеспечивает инкапсуляцию объекта данных.
- Запрещает прямой доступ к памяти. Указатель не может быть получен.
- Обеспечивает проверку ограничений и валидность содержимого.
- Обеспечивает наличие информации о переменной и ее состоянии, т.е инициализирована ли она, находится ли в допустимых пределах и т.д....
- Защита переменной обеспечивается в едином месте.
- Позволяет оптимизировать ISR, т.к переменная может не требовать более модификатора volatile.
- Доступ выглядит как функция, а не как переменная, что минимизирует риск ее использования как локальной переменной, например, в уравнениях.

Доступ через функции

Adc.c

```
unsigned int     avgCum = 0;

// ADC Speed Sample, IPL = 4
void __attribute__((interrupt,
    auto_psv)) _ADC1Interrupt(void)
{
    unsigned int rawADC;
    _AD1IF = 0;
    rawADC = ADC1BUF0;
    avgCum = ((avgCum - (avgCum >>
        4)) + rawADC);
}
```

```
unsigned int getAdcAvgCum(void)
{
    unsigned int volatile   myAvgCum;
    SuspendADInterrupts();
    myAvgCum = avgCum;
    ResumeADImterrupts();
    return(myAvgCum);
}
```

Единая обработка разрешений прерываний

```
void SuspendADInterrupts(void)
{
  if(osIntSaveDisableCounter == 0) {
    osSavedIntLevelNested = AD_IEN;
    AD_IEN = 0;
  }
  osIntSaveDisableCounter++;
}

void ResumeADInterrupts(void)
{
  if (osIntSaveDisableCounter) osIntSaveDisableCounter--;
  if ((osIntSaveDisableCounter == 0) &&
      (osSavedIntLevelNested)) {
        AD_IEN = 1;
  }
}
```

Доступ через функции

Main.c

```
int main (void)
{
    while(1){}
}

// PID, IPL = 3
void __attribute__((interrupt,
    auto_psv)) _T1Interrupt(void)
{
    unsigned int    error;
    unsigned int    myAvgCum;

    _T1IF = 0;
    myAvgCum = getAdcAvgCum();
    error = (targetSpeed - (3 *
        myAvgCum) - (myAvgCum >> 1));
    integral += error;
    powerOut = (5 * error) + (2 *
     integral);
}
```

Дизассемблер

```
21: _T1IF = 0;
  02E0  A96084     bclr.b 0x0084,#3
22: myAvgCum = getAdcAvgCum();
  02E2  07FFE7     rcall getAdcAvgCum
23: error = (targetSpeed - (3 *
         myAvgCum) - (myAvgCum >> 1));
  02E4  B90163     mul.su w0,#3,w2
  02E6  804011     mov.w targetSpeed,w1
  02E8  508102     sub.w w1,w2,w2
  02EA  D10000     lsr.w w0,w0
  02EC  510100     sub.w w2,w0,w2
24: integral += error;
  02EE  804000     mov.w integral,w0
  02F0  410000     add.w w2,w0,w0
      . . .
```

Заключение

Для корректной работы программ необходимо:

- Применять модификатор volatile к переменным, значение которых может изменяться в другом контексте
- Минимизировать число разделяемых переменных, которые могут привести к нарушению целостности
- Обеспечить атомарный доступ к разделяемым переменным
- Минимизировать количество атомарных операций доступа
- Обеспечить достаточную глубину стека и обработку исключений переполнения стека.

Дополнительная информация

- Атомарный доступ к структурам (А. Борисов)
- www.pic24.ru/doku.php/articles/mchp/c30_atomic_access

- volatile для «чайников» (В. Тимофеев)
- http://pic24.ru/doku.php/osa/articles/volatile_for_chainiks

Заключение

Спасибо !!!

Вопросы ?