Книга по работе с WinAVR и AVR Studio

Роман Абраш
г. Новочеркасск
E-mail: **arv@radioliga.com**

Продолжение. Начало в №1-12/2010; №1-4/2011

Операнды оператора asm()

Параметры оператора asm, используемые в качестве операндов инструкций ассемблера, обозначаются, как было сказано ранее, определенными символами, помещаемыми в двойные кавычки (см. **таблицу 4**).

В **таблице 4** перечислены все допустимые символы для обозначения параметров. Обнаружив такой символ, компилятор самостоятельно подставит вместо него соответствующее значение из числа допустимых. При этом могут быть странные на первый взгляд ошибки, если символ выбран программистом неверно. Например, программист применил для операнда-приемника инструкции ORI символ "r". Компилятор может выбрать для этого любой регистр из числа имеющихся в его распоряжении в текущий момент. При этом может быть выбран и регистр, например, **r3**, что приведет к ошибке, т.к. для инструкции ORI допустимо применять только регистры из старшей половины. Компилятор может выбрать и «правильный» регистр, и это может привести к сомнениям: почему ранее работавшая без ошибок ассемблерная вставка вдруг стала вызывать ошибку. Поэтому правильным выбором для ORI будет символ "d".

В **таблице 5** приведены все инструкции ассемблера, требующие операндов, с указанием подходящих им символов. Однако, эти требования недостаточно строгие, т.к. не ограничивают, например, диапазон номеров битов от 0 до 7 и т.п.

Любой символ может предваряться символом-модификатором:

= – обозначает, что операнд только для записи. Используется для операндов результата.

+ – обозначает, что операнд для записи и чтения.

& – обозначает, что для вывода должен использоваться новый регистр.

Используемые для результата операнды всегда должны быть только для записи и иметь вид «леводопустимого выражения». Компилятор не проверяет совместимость типов операнда и присваемого ему значения.

Таблица 4

Символ	Что обозначает	Допустимые значения
a	Старшие регистры без указателей	r16 … r23
b	Базовые регистровые пары – указатели	y, z
d	Старшие регистры	r16 … r31
e	Регистровые пары – указатели	x, y, z
q	Указатель стека	SPH:SPL
r	Любой регистр	r0 … r31
t	Вспомогательный регистр	r0
w	Специальный старший регистр	r24, r26, r28, r30
x	Регистровая пара X	x (r27:r26)
y	Регистровая пара Y	y (r29:r28)
z	Регистровая пара Z	z (r31:r30)
G	Константа в формате с плавающей точкой	0.0
I	6-битовое положительное число (константа)	0 … 63
J	6- битовое отрицательное число (константа)	-63 … 0
K	Целочисленная константа	2
L	Целочисленная константа	0
l	Младшие регистры	r0 … r15
M	Однобайтная константа	0 … 255
N	Целочисленная константа	-1
O	Целочисленная константа	8, 16, 24
P	Целочисленная константа	1
Q	Адрес памяти по указателю Y или Z со смещением	
R	Целочисленная константа	-6 to 5

Входные операнды должны быть определены как доступные только для чтения (без символа-модификатора). Однако, в случае, когда единственный операнд используется и как входное и как выходное значение, существует способ обойти это ограничение: необходимо вместо символа операнда использовать номер соответствующего операнда в инструкции:

```
asm volatile(«swap %0» : «=r» (value) : «0» (value));
```

Пример показывает, как переменная value используется в качестве входного значения и в нее же помещается результат ассемблерной команды, меняющей тетрады байта. "0" – это номер операнда команды SWAP. Такая запись указывает компилятору использовать для операнда-приемника результата то же самое значение, что было выбрано для операнда-источника. Однако следует знать, что компилятор может выбрать одно и то же значение и для источника и для приемника, даже если это явно не указано. Обычно это не опасно, но может быть фатальным, если значение операнда-результата модифицируется другими ассемблерными инструкциями до выполнения сохранения результата оператора asm.

```
asm volatile(«in %0,%1»    «\n\t»
             «out %1, %2»  «\n\t»
             : «=&r» (input)
             : «I» (_SFR_IO_ADDR(port)), «r» (output)
            );
```

Таблица 5

Инструкция	Символы операндов	Инструкция	Символы операндов
adc	r, r	add	r, r
adiw	w, I	and	r, r
andi	d, M	asr	r
bclr	I	bld	r, I
brbc	I, label	brbs	I, label
bset	I	bst	r, I
cbi	I, I	cbr	d, I
com	r	cp	r, r
cpc	r, r	cpi	d, M
cpse	r, r	dec	r
elpm	t, z	eor	r, r
in	r, I	inc	r
ld	r, e	ldd	r, b
ldi	d, M	lds	r, label
lpm	t, z	lsl	r
lsr	r	mov	r, r
movw	r, r	mul	r, r
neg	r	or	r, r
ori	d, M	out	I, r
pop	r	push	r
rol	r	ror	r
sbc	r, r	sbci	d, M
sbi	I, I	sbic	I, I
sbiw	w, I	sbr	d, M
sbrc	r, I	sbrs	r, I
ser	d	st	e, r
std	b, r	sts	label, r
sub	r, r	subi	d, M
swap	r		

В этом примере сначала осуществляется считывание порта **port**, а затем другое значение в этот же самый порт выводится. Компилятор вполне может выбрать для ввода и вывода один и тот же регистр, например, **r5**, при этом значение **r5**, считанное первой командой еще до того, как будет записано эпилогом в переменную **output**, будет изменено значением, загруженным из переменной **input** для вывода во второй команде. Чтобы этого избежать, применен модификатор **&**, который указывает компилятору выбрать для операнда источника обязательно другой регистр, т.е. не совпадающий ни с одним из выбранных для других операндов "r".

Нередка ситуация, когда ассемблерная вставка должна манипулировать числами более одного байта. В этом случае возникает проблема с обращением к нескольким байтам, составляющим один операнд. Для обозначения регистров, в которых хранится значение переменной, используются дополнительные символы – заглавные латинские символы «A», «B», «C» и т.д. Такая запись указывает компилятору выбрать очередной регистр для хранения очередного байта, составляющего переменную. Младший байт соответствует символу «A», более старшие последовательно назначаются для «B», «C» и т.д. Программист может использовать соответственно «%A0» для обращения к младшему байту первого операнда, «%A1» – младшему байту второго операнда и т.д.

В этом примере производится перестановка байтов в 16-битной переменной value:

```
asm volatile(«mov __tmp_reg__, %A0» «\n\t»
             «mov %A0, %B0»         «\n\t»
             «mov %B0, __tmp_reg__» «\n\t»
             : «=r» (value)
             : «0» (value)
            );
```

Перестановка байтов 32-битного переменной value:

```
asm volatile(«mov __tmp_reg__, %A0» «\n\t»
             «mov %A0, %D0»         «\n\t»
             «mov %D0, __tmp_reg__» «\n\t»
             «mov __tmp_reg__, %B0» «\n\t»
             «mov %B0, %C0»         «\n\t»
             «mov %C0, __tmp_reg__» «\n\t»
             : «=r» (value)
             : «0» (value)
            );
```

Для обозначения операнда, используемого и как источник, и как приемник результата, разумно использовать модификатор «**+**»:

```
asm volatile(«mov __tmp_reg__, %A0» «\n\t»
             «mov %A0, %D0»         «\n\t»
             «mov %D0, __tmp_reg__» «\n\t»
             «mov __tmp_reg__, %B0» «\n\t»
             «mov %B0, %C0»         «\n\t»
             «mov %C0, __tmp_reg__» «\n\t»
             : «+r» (value)
            );
```

Непредвиденная проблема может возникнуть еще и в случае, если осуществляется работа с указателем, т.е. регистровой парой. Предположим, программист использует следующее определения операнда:

```
      «e» (ptr)
```

Допустим, компилятор избрал для хранения этого операнда регистровую пару Z, т.е. %A0 соответствует ZL (r30), а %B0 – ZH (r31). Однако компилятор вызовет ошибку, если программист использует обращение к этой паре так:

```
      ld      r24, Z
```

Чтобы компилятор сгенерировал правильный код, необходимо использовать только такую конструкцию:

```
      ld      r24, %a0
```

То есть вопреки всему ранее сказанному надо использовать ***нижний*** регистр для указания младшего байта операнда! Это проблема реализации компилятора GCC текущей версии[40].

Зависимости оператора asm()

Как было сказано, последним элементом оператора **asm** может быть список зависимостей. Этот список может отсутствовать вместе с отделяющим его двоеточием. Однако если внутри ассемблерной вставки программист использует значения регистров, не указанных в списках операндов, он обязан уведомить компилятор об этом. В этом случае в код пролога будет добавлены команды сохранения перечисленных в этом списке регистров, а в коде эпилога эти значения будут восстановлены в прежнем виде.

Например, при помощи следующего кода осуществляется атомарное увеличение 8-битной переменной:

```
asm volatile(
    «cli»               «\n\t»
    «ld r24, %a0»       «\n\t»
    «inc r24»           «\n\t»
    «st %a0, r24»       «\n\t»
    «sei»               «\n\t»
    :
    : «e» (ptr)
    : «r24»
);
```

Примечательно в этом примере то, что используется указатель для обращения к переменной, иначе сохранение значения произошло бы только в коде эпилога, который будет добавлен после команды разрешения прерывания, т.е. это уже нарушило бы условие атомарности ассемблерной вставки. А с использованием указателя переменная обновляется до разрешения прерываний.

Более правильным было бы в этом случае использовать **__tmp_reg__** вместо **r24**:

```
asm volatile(
    «cli»      «\n\t»
    «ld __tmp_reg__, %a0»    «\n\t»
    «inc __tmp_reg__»        «\n\t»
    «st %a0, __tmp_reg__»    «\n\t»
    «sei»      «\n\t»
    :
    : «e» (ptr)
);
```

Вышеприведенные примеры имеют одну неприятную особенность: их нельзя использовать в участках программы, где прерывания уже запрещены, т.к. эти вставки принудительно разрешают прерывания. Казалось бы, эту проблему легко решить при помощи локальной переменной:

```
{
    uint8_t s;
    asm volatile(
        «in %0, __SREG__»           «\n\t»
        «cli»                       «\n\t»
        «ld __tmp_reg__, %a1»       «\n\t»
        «inc __tmp_reg__»           «\n\t»
        «st %a1, __tmp_reg__»       «\n\t»
        «out __SREG__, %0»          «\n\t»
        : «=&r» (s)
        : «e» (ptr)
    );
}
```

К сожалению, это не так, хотя и выглядит правильно. Ассемблерный код модифицирует переменную, на которую указывает указатель **ptr**. Но загрузка значения указателя в регистровую пару осуществляется в коде пролога, который выполняется еще до запрещения прерываний, т.е. вполне значение указателя может быть изменено на этом этапе. Разумеется, в этом случае результат работы ассемблерной вставки непредсказуем. Более того, при оптимизации значение указателя вообще может оказаться не в ОЗУ, а в регистрах. Самое меньшее, что можно в этом случае сделать, это применить специальный элемент списка зависимостей вставки **memory**:

[41] На момент написания книги это соответствует версии GCC 4.3.xx

```
{
    uint8_t s;
    asm volatile(
        «in %0, __SREG__»           «\n\t»
        «cli»                       «\n\t»
        «ld __tmp_reg__, %a1»       «\n\t»
        «inc __tmp_reg__»           «\n\t»
        «st %a1, __tmp_reg__»       «\n\t»
        «out __SREG__, %0»          «\n\t»
        : «=&r» (s)
        : «e» (ptr)
        : «memory»
    );
}
```

Это укажет компилятору учесть при оптимизации тот факт, что ассемблерная вставка использует память, которую нельзя модифицировать. Но наиболее простой и удобный способ все-таки состоит в том, чтобы использовать **volatile** при определении указателя **ptr**:

```
volatile uint_t *ptr;
```

Это обеспечит правильное поведение как самой ассемблерной вставки, так и оптимизатора в ее отношении.

Примечание. Случаи действительной необходимости в указании списка зависимостей достаточно редки. Почти всегда есть возможность избежать их, обеспечив компилятору больше свободы в оптимизации кода.

Макросы на ассемблере

Для многократного использования ассемблерных вставок в различных проектах имеет смысл разместить их в заголовочном файле в виде макросов. AVR-LIBC содержит немалое количество таких файлов, найти которые можно в директории avr/include. Однако такое использование ассемблерных вставок может вызвать предупреждения компилятора для режима соответствия ANSII-стандарту. Чтобы избежать предупреждений, достаточно писать **__asm__** вместо **asm** и **__volatile__** вместо **volatile** – это полные синонимы.

Более заметная проблема ассемблерных вставок в виде макросов связана с использованием меток. Макрос – это ведь просто подстановка текста, поэтому в случае использования обычных меток в ассемблерных вставках возможно появление одинаковых меток в различных участках программы, что недопустимо. Для избежания этого следует использовать особый синтаксис меток для ассемблерных вставок: в определении метки используется сочетание «**%=**», которое на этапе компиляции заменяется на некий уникальный номер, таким образом, гарантируя уникальность всех меток. Вот, например, как реализован один из макросов в iomacros.h:

```
#define loop_until_bit_is_clear(port,bit)  \
        __asm__ __volatile__ (             \
        «L_%=: « «sbic %0, %1» «\n\t»      \
                «rjmp L_%=»                \
                : /* no outputs */        \
                : «I» (_SFR_IO_ADDR(port)),
                  «I» (bit)
        )
```

Метка **L_%=** будет заменена на что-то типа **L_1234**, причем это будет гарантированно уникальная метка.

Функции на ассемблере

Макрос с ассемблерной вставкой будет приводить к появлению одного и того же кода каждый раз при использовании макроса. Для многих критичных к размеру памяти приложений это неприемлемо. В этом случае логичнее реализовать С-функцию целиком на ассемблере.

```
void delay(uint8_t ms)
{
    uint16_t cnt;
    asm volatile (
                    «\n»
    «L_dl1%=:»              «\n\t»
        «mov %A0, %A2»      «\n\t»
        «mov %B0, %B2»      «\n»
    «L_dl2%=:»              «\n\t»
        «sbiw %A0, 1»«\n\t»
        «brne L_dl2%=»      «\n\t»
        «dec %1»            «\n\t»
        «brne L_dl1%=»      «\n\t»
        : «=&w» (cnt)
        : «r» (ms), «r» (delay_count)
        );
}
```

Пример демонстрирует реализацию функции задержки программным циклом на заданное количество миллисекунд. В функции используется глобальная переменная **delay_count**, которая должна содержать значение тактовой частоты, деленное на 4000, причем это значение должно быть присвоено этой переменной до обращений к функции. Как было показано ранее, функция использует локальную переменную для сохранения значения используемых регистров.

Следующий пример показывает, как реализуется возврат значения из ассемблерной функции:

```
uint16_t inw(uint8_t port)
{
    uint16_t result;
    asm volatile (
        «in %A0,%1» «\n\t»
        «in %B0,(%1) + 1»
        : «=r» (result)
        : «I» (_SFR_IO_ADDR(port))
        );
    return result;
}
```

Функция **inw** возвращает 16-разрядное число, считанное из пары «смежных» портов ввода-вывода. port определяет «младший» порт.

Соглашения об именах Си и ассемблера

По умолчанию GCC использует одинаковые имена переменных и функций для Си и для ассемблера. Однако программист может изменить эту ситуацию при помощи особой формы оператора asm:

```
unsigned long value asm(«clock») = 3686400;
```

Это определение вынудит компилятор использовать имя **clock** вместо его значения. Такая «подмена» имеет смысл только для глобальных (статических) или внешних переменных, так как локальные переменные не имеют символьных имен для ассемблера. Кроме того, локальные переменные могут быть помещены в регистры.

Программист может назначить локальной переменной определенный регистр:

```
void Count(void)
{
    register unsigned char counter asm(«r3»);

    ... какие-то действия ...
    asm volatile(«clr r3»);
    ... какие-то действия ...
}
```

В этом примере показано, как заставить компилятор поместить локальную переменную counter в регистр **r3**. Однако, это не «вечное» закрепление: если оптимизатор компилятора сочтет, что значение переменной более не требуется сохранять, назначение регистра **r3** может быть переопределено. Если программист закрепляет слишком много регистров за переменными, компилятор может исчерпать ресурсы регистров для компиляции остального кода. Кроме этого, принудительное назначение регистра переменной чаще приводит к ухудшению эффективности кода вместо ожидаемой его оптимизации.

Существует и способ изменить имя функции:

```
extern long Calc(void) asm («CALCULATE»);
```

Вышеприведенный пример заставит компилятор при обращении к функции **Calc()** генерировать инструкцию вызова функции **CALCULATE**.

Окончание следует.