Книга по работе с WinAVR и AVR Studio

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

Продолжение. Начало в №1-12/2010; №1-3/2011

ASSEMBLER

Есть несколько причин использовать ассемблер при разработке программ. Среди них можно выделить основные:

- Применение моделей МК, не имеющих достаточного количества ОЗУ
- Получение максимально быстродействующих участков кода
- Реализация возможностей, которые нельзя или сложно выполнить стандартными средствами Си.

За исключением первого варианта, все эти задачи могут быть решены при помощи ассемблерных вставок в Си-программе. Хотя GCC ориентирован на разработку программ на языках C и C++, все же имеется поддержка ассемблерных вставок и файлов, целиком написанных на ассемблере.

Нет необходимости использовать отдельный компилятор ассемблера, достаточно, оформив соответственно ассемблерный исходный текст, вызвать Си-компилятор GCC, который в свою очередь в нужный момент вызовет препроцессор, ассемблер и компоновщик, сгенерировав необходимый набор выходных файлов. Это позволяет сделать процесс разработки на ассемблере неотличимым от привычного для Си-программиста.

Как было сказано ранее, в проект могут входить файлы исходных текстов на Си и ассемблере. Для ассемблерных файлов принято расширение «s», и тут следует обратить внимание на одну важную особенность: если расширение файла имеет нижний регистр, то файл будет компилироваться непосредственно, а если расширение будет в верхнем регистре, т.е. «S», произойдет обязательный вызов препроцессора перед компиляцией. Различие регистра происходит даже для файловых систем, не чувствительных к регистру имен файлов (в том числе для Windows). При необходимости можно принудительно указать необходимость запуска препроцессора при помощи опции компилятора:

-x assembler-with-cpp

Исходные тексты полностью на ассемблере

GCC использует расширенный синтаксис ассемблера по сравнению с тем, что встроен в AVR Studio. Это потому, что **GAS** (GNU Assembler) генерирует перемещаемый объектный код.

Прежде всего, поддерживаются в полной мере все директивы Си-препроцессора, включая условную компиляцию, макросы-функции и т.п. Кроме того, поддерживаются псевдооператоры (директивы), унаследованные от GAS. Полная документация по этому диалекту ассемблера заслуживает отдельной книги. Здесь же перечислены только наиболее важные и востребованные возможности (особенности).

1. Комментарии в Си-стиле. Для комментариев в ассемблерных текстах можно использовать не только традиционную точку с запятой (однострочный комментарий), но и пару сочетаний символов «/*» и «*/» (многострочный комментарий). Так же как комментарий воспринимается любая строка, начинающаяся с символа «#», если после этого символа присутствует хотя бы один пробел (табуляция).
2. Псевдооператор **.byte** для выделения ячейки ОЗУ (1 байт).
3. Псевдооператор **.ascii** для определения строки символов (не завершающихся нулем автоматически).
4. Псевдооператор **.asciiz** для определения строки символов, оканчивающейся нулем (добавляется автоматически).
5. Директива **.data** для переключения на секцию памяти .data.
6. Директива **.text** для переключения на секцию памяти .text.
7. Директива **.section**, при помощи которой задается выбор нужной секции памяти.
8. Директива **.set** для определения константы (эквивалент директивы .equ).
9. Директива **.global** для определения public-символа, т.е. символа, видимого в других модулях программы.
10. Локальные и глобальные метки. После директивы **.global** могут использоваться локальные метки, представляющие собой число с двоеточием. Директива **.global** делает предыдущие локальные метки невидимыми для текущей функции, т.е. вновь могут быть определены метки 0:, 1: и т.д. Для перехода к локальным меткам нужно использовать суффикс «**b**» или «**f**» для обозначения направления перехода – назад (*backward*, вверх по тексту) или вперед (*forward*, вниз по тексту) соответственно. Глобальная метка должна начинаться с нецифрового символа.
11. Директива **.extern** для определения внешнего символа, т.е. символа, определенного в другом модуле программы.

Поддерживается и стандартная директива **.org**, однако ее применение бессмысленно, т.к. для перемещаемого объектного кода расположение объектов в памяти формирует компоновщик, выбирая необходимые адреса автоматически.

Имеются и дополнительные операторы ассемблера:

lo8(w) – возвращает младшие 8 бит 16-битного числа **w**

hi8(w) – возвращает старшие 8 бит 16-битного числа **w**

pm(addr) – позволяет получить адрес константы в памяти программ в формате, принятом для доступа (как известно, адресация памяти программ в AVR ведется по 16-битным словам, а доступ – по адресу байта).

Пример небольшой программы на ассемблере для микроконтроллера **AT90S1200**, которая формирует 100 кГц меандр на выводе **PD6** (используется кварц 10,7 МГц):

```
#include <avr/io.h>              ; примечание 1
work    =       16               ; примечание 2
tmp     =       17
inttmp  =       19
intsav  =       0
SQUARE  =       PD6              ; примечание 3
                                 ; примечание 4:
tmconst= 10700000 / 200000      ; 100 кГц => 200000 фронтов в сек.
fuzz=   8                       ; число тактов на вызов прерывания

        .section .text
        .global main                                ; примечание 5
main:
        rcall   ioinit
1:
        rjmp    1b                                  ; примечание 6

        .global TIMER0_OVF_vect                     ; примечание 7
TIMER0_OVF_vect:
        ldi     inttmp, 256 - tmconst + fuzz
        out     _SFR_IO_ADDR(TCNT0), inttmp         ; примечание 8

        in      intsav, _SFR_IO_ADDR(SREG)          ; примечание 9

        sbic    _SFR_IO_ADDR(PORTD), SQUARE
        rjmp    1f
        sbi     _SFR_IO_ADDR(PORTD), SQUARE
        rjmp    2f
1:      cbi     _SFR_IO_ADDR(PORTD), SQUARE
2:

        out     _SFR_IO_ADDR(SREG), intsav
        reti

ioinit:
```

```
        sbi     _SFR_IO_ADDR(DDRD), SQUARE

        ldi     work, _BV(TOIE0)
        out     _SFR_IO_ADDR(TIMSK), work

        ldi     work, _BV(CS00)             ; tmr0:  CK/1
        out     _SFR_IO_ADDR(TCCR0), work

        ldi     work, 256 - tmconst
        out     _SFR_IO_ADDR(TCNT0), work

        sei

        ret

        .global __vector_default            ; примечание 10
__vector_default:
        reti

        .end
```

Примечания:

1. Подключение того же самого заголовочного файла, что и для программы на Си. Следует помнить, что не любой заголовочный файл может быть подключен к ассемблерному тексту.
2. Определение локальной константы может быть выполнено так же при помощи характерной для Си директивы препроцессора **#define work 16**.
3. Использование константы, определенной в подключенном заголовочном файле.
4. При вычислениях констант ассемблер использует разрядность чисел, характерной для хост-платформы (т.е. для Windows это будет ***32 разряда***), в отличие от Си, который использует по умолчанию тип ***int***. Чтобы получить меандр 100 кГц, необходимо переключать уровень **PD6** 200000 раз в секунду, что весьма критично ко времени исполнения. Необходимо учитывать затраты процессорного времени на генерацию прерывания и возврат из него, для чего и используется корректирующая константа fuzz (8 – т.е. 4 такта на прерывание, 2 такта на переход по вектору прерывания и 2 такта для переключения **PD6**).
5. Любая внешняя функция должна быть объявлена как **.global**. Использование стандартного имени **main** приводит к тому, что компоновщик будет использовать его в качестве точки входа в программу (как для Си **main()**).
6. Основной цикл программы – просто бесконечный пустой цикл. Вся работа ведется по прерываниям от таймера. Обратите внимание на то, как осуществлен переход к локальной метке (использование суффикса «**b**»).
7. Функция обработчика прерывания должна иметь имя из числа определенных констант для векторов прерывания (см. avr/interrupt.h – Прерывания) – это позволит компоновщику разместить ее в нужном месте таблицы векторов. Необходимо помнить, что ни ассемблер, ни компоновщик не могут проверить корректность обработчика (например, наличие команды RETI для возврата) – это должен реализовать программист.
8. Как было показано ранее (см. avr/sfr_defs.h – Регистры специальных функций AVR), для получения адреса регистра ввода-вывода необходимо использовать макрос _SFR_IO_ADDR() (AT90S1200 не имеет ОЗУ, поэтому обращение к регистру ввода-вывода как к ячейке памяти невозможно, т.е. необходимо использовать команды IN/OUT).
9. При разработке обработчиков прерывания следует помнить, что контекст основной программы не должен изменяться во время прерывания. В данном примере не предпринято мер по сохранению контекста, т.к. основной цикл ничего не делает, но в реальных ситуациях следует позаботиться о том, чтобы не изменить содержимое регистров, используемых компилятором в основной программе, а так же регистра состояния SREG. Для сохранения контекста можно применять стек или ячейки ОЗУ.
10. Как было показано ранее (см. avr/interrupt.h – Прерывания), для всех неиспользуемых векторов прерывания может использоваться адрес «обработчика по умолчанию» **__vector_default**. Этот символ должен быть объявлен **.global**, тогда компоновщик заполнит всю таблицу незадействованных векторов его адресом (в противном случае таблица будет заполнена переходами к адресу 0x0000). Разумеется, этот обработчик должен обеспечивать нормальный возврат из прерывания – команду RETI.

Ассемблерные вставки в Си-программе

Ассемблерные вставки используются в программах Си, если для решения поставленной задачи достаточно буквально нескольких команд ассемблера в критически важном участке программы и переписывание всей программы нецелесообразно. Особенность ассемблерных вставок в том, что для них применяется особый синтаксис ассемблера, позволяющий компилятору самостоятельно подбирать, например, регистры для операндов команд. Таким образом, компилятор «знает», какие ресурсы задействованы в ассемблерной вставке и использует это знание для наиболее оптимального построения предыдущего и последующего (относительно вставки) Си-кода.

Особенностью ассемблерных вставок является то, что внутри них реализуется автоматически доступ к переменным, определенным в тексте Си-программы.

Оператор asm()

Для ассемблерной вставки используется ключевое слово (оператор) asm, синтаксис которого проще всего рассмотреть на примере (считывание порта D):

asm(«in %0, %1» : «=r» (value) : «I» (_SFR_IO_ADDR(PORTD)));

Внутри скобок оператора asm находятся строка, состоящая из трех частей, разделенных двоеточиями:

«in %0, %1» – это первая часть, инструкция ассемблера. Она обязательно помещается в двойные кавычки. Состоит из обычной мнемоники ассемблерной команды, операнды которой имеют особый вид: номер операнда, предваряемый символом процента (операнды нумеруются с нуля). Вместо обозначения операндов по номеру с процентом, можно использовать поименованные операнды. В этом случае вместо %0 нужно использовать имя операнда в квадратных скобках, а последующие части «тела» оператора asm так же предварять этим именем (в этом случае они могут следовать в другом порядке, т.к. назначение операнда определяется уже не порядковым номером, а именем):

asm volatile («in [x1], [x2]» : [x1] «=r» (value) : [x2] «I» (_SFR_IO_ADDR(PORTD)));

«=r» (value) – вторая часть, соответствующая операнду %0 – список выходных параметров инструкции ассемблера. Эта часть состоит из размещенного в кавычках условного обозначения параметра результата и (через пробел) в скобках наименование переменной для его сохранения.

«I» (_SFR_IO_ADDR(PORTD)) – третья часть, соответствующая операнду %1 – список входных параметров инструкции. Как и предыдущая часть, эта состоит так же из указанного в кавычках условного обозначения параметра и в скобках – его значения.

Такая запись может показаться несколько избыточной и странной, однако пока продолжим знакомиться с особенностями синтаксиса ассемблерных вставок, вскоре вопрос о смысле этой «избыточности» будет снят.

В операторе **asm** может быть еще четвертая часть (в примере она отсутствует) – список регистров, от содержимого которых оператор зависит или которые зависят от него. Дело в том, что фактически оператор **asm** может быть оттранслирован компилятором в несколько иные инструкции ассемблера, причем их количество может быть отлично от указанных программистом. В частности, после компиляции вышеприведенного примера результирующий код будет следующим (выдержка из S-файла, генерируемого GCC):

```
lds r24,value
/* #APP */
in r24, 12
/* #NOAPP */
sts value,r24
```

Комментарии вставлены компилятором, чтобы проинформировать ассемблер о том, какая инструкция введена программистом. Т.е. фактически одна команда ассемблера оттранслировалась в три. Регистр **r24** был выбран компилятором (компилятор мог выбрать и любой иной регистр по своему усмотрению). Команда загрузки **r24** и сохранения его результата в переменой **value** – так же добавлены компилятором. Очевидно, что первая команда не имеет смысла, если включена оптимизация, она будет удалена оптимизатором. Более того, если выяснится, что значение **value** более нигде в функции не используется – оптимизатором будет удалены все эти команды! Чтобы исключить воздействие оптимизатора на ассемблерную вставку, необходимо использовать ключевое слово **volatile**:

asm volatile («in %0, %1» : «=r» (value) : «I» (_SFR_IO_ADDR(PORTD)));

Обратите внимание на то, что компилятор автоматически добавляет «пролог» и «эпилог» к ассемблерному тексту вставки: пролог служит для загрузки в регистры значения из входных переменных, а эпилог реализует сохранение результата в переменных. Пролог добавляется всегда перед написанными программистом инструкциями, эпилог – всегда после. С этой особенностью связан ряд тонкостей, которые рассмотрены далее.

Все части оператора могут быть на разных строках:

```
asm (
      «in %0, %1» :
      «=r» (value) :
      «I» (_SFR_IO_ADDR(PORTD))
);
```

В некоторых случаях вторая и последующие части оператора **asm** не требуются. Не смотря на то, что можно просто их опустить вместе с двоеточиями, более хорошей практикой будет сохранение «пустых» мест:

```
asm(“nop” : : );
```

Кроме того, в одном операторе **asm** может быть перечислено сразу несколько команд ассемблера:

```
asm volatile (
      “nop\n\t”
      “nop\n\t”
      “nop\n\t”
      : :
);
```

В этом случае каждая команда ассемблера может начинаться с новой строки, как и принято в ассемблере. Желательно принудительно указывать наличие символов перевода строки и табуляции **“\n\t”**, иначе генерируемый компилятором ассемблерный код будет нечитаемым.

Вторая и последующие части оператора asm относятся ко всему оператору, т.е. в случае многострочного оператора они должны быть указаны только один раз в самом конце.

Особое внимание следует обратить на то, что для сохранения команд **NOP** в коде программы при включенной оптимизации всегда необходимо использовать **volatile**.

В операторе asm можно использовать все инструкции, как и в обычном ассемблере, в том числе метки. Эти метки можно использовать для инструкций переходов: **Особенности использования меток** рассмотрены далее. Кроме прочего, можно использовать символьные имена для особых регистров:

__SREG__ – регистр состояния SREG
__SP_L__ – младший байт указателя стека
__SP_H__ – старший байт указателя стека
__tmp_reg__ – вспомогательный регистр r0
__zero_reg__ – регистр r1 – всегда обнулен

Регистр __tmp_reg__ может использоваться без необходимости сохранения его содержимого, а __zero_reg__ должен сохранять свое значение всегда (т.е. не может использоваться в качестве операнда-приемника результата).

Продолжение в №5/2011