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

Книга по работе с WinAVR и AVR Studio

Продолжение. Начало в №1-12/2010

AVR-СПЕЦИФИЧНЫЕ МОДУЛИ

К специфичным для AVR относятся модули, реализующие ряд функций, выполнение которых возможно исключительно на платформе AVR-микроконтроллеров, т.е. аппаратно-зависимые функции. В основном эти функции реализуют поддержку особенностей архитектуры микроконтроллеров, например, встроенную EEPROM или возможность записи в память программ.

avr/boot.h – Поддержка загрузчиков AVR

Модуль определяет ряд макросов, упрощающих разработку программ пользователя, реализующих алгоритм самозагрузки программы (boot-loading). Это возможно не для любого микроконтроллера AVR. Макросы модуля позволяют работать со всей областью памяти программ микроконтроллеров.

Макросы не используют глобальное запрещение прерываний, это оставлено на совести программиста. Пример функции самозаписи памяти программ см. на ***врезке***.

Определения модуля

BOOTLOADER_SECTION

BOOTLOADER_SECTION – модификатор атрибутов переменной или функции в программе пользователя. Наличие этого макроса дает указание компоновщику разместить соответствующий объект в строго определенной области памяти, соответственно требованиям используемого микроконтроллера.

Пример: BOOTLOADER_SECTION void func1(void); - функция func1() будет размещена в области памяти микроконтроллера, отведенной для загрузчика.

GET_LOW_FUSE_BITS

GET_LOW_FUSE_BITS – константа, задающая адрес чтения младшего байта fuse-битов микроконтроллера для применения в boot_lock_fuse_bits_get()

GET_LOCK_BITS

GET_LOCK_BITS – константа, задающая адрес чтения байта lock-битов микроконтроллера для применения в boot_lock_fuse_bits_get()

GET_EXTENDED_FUSE_BITS

GET_EXTENDED_FUSE_BITS – константа, задающая адрес чтения байта расширенных fuse-битов микроконтроллера для применения в boot_lock_fuse_bits_get()

GET_HIGH_FUSE_BITS

GET_HIGH_FUSE_BITS – константа, задающая адрес чтения старшего байта fuse-битов микроконтроллера для применения в boot_lock_fuse_bits_get()

Врезка. Пример функции самозаписи памяти программ.

```
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
void boot_program_page (uint32_t page, uint8_t *buf){
uint16_t i;
uint8_t sreg;
// запрещаем прерывания
sreg = SREG;
cli();
eeprom_busy_wait();    // ожидаем, завершения предыдущей записи
boot_page_erase (page);// стираем указанную страницу
boot_spm_busy_wait();  // ожидаем завершения стирания
// цикл заполнения страницы памяти
for (i=0; i<SPM_PAGESIZE; i+=2){
// заполняем словами «в обратном порядке»
uint16_t w = *buf++;
w += (*buf++) << 8;
boot_page_fill (page + i, w);
}
boot_page_write (page);// записываем заполненную страницу
boot_spm_busy_wait();  // ожидаем завершения записи
// разрешение перехода к RWW-секции памяти. Это нужно, если
// после самозаписи должен быть осуществлен переход к основной программе
boot_rww_enable();
// восстановление состояния прерываний, как было при входе в функцию
SREG = sreg;
}
```

Макросы-функции, определенные в модуле

Так как большинство макросов выполнены в виде макросов-функций, далее они будут рассмотрены без упоминания того, что это не функции на самом деле.

boot_spm_interrupt_enable()
boot_spm_interrupt_enable()
Описание: разрешает прерывания после исполнения инструкции записи в память программ.

boot_spm_interrupt_disable()
boot_spm_interrupt_disable()
Описание: запрещает прерывания после исполнения инструкции записи в память программ.

boot_is_spm_interrupt()
boot_is_spm_interrupt()
Описание: проверяет, разрешены ли прерывания после исполнения инструкции записи в память программ.

boot_rww_busy()
boot_rww_busy()
Описание: проверяет, занята ли секция RWW памяти программ.

boot_spm_busy()
boot_spm_busy()
Описание: проверяет, завершена ли инструкция записи в память программ.

boot_spm_busy_wait()
boot_spm_busy_wait()
Описание: ожидает, пока не завершится инструкция записи в память программ.

boot_lock_fuse_bits_get()
boot_lock_fuse_bits_get(address)
Описание: возвращает считанный байт **fuse**-битов или **lock**-битов микроконтроллера в зависимости от значения **address**, которое может принимать одно из следующих значений: GET_LOW_FUSE_BITS, GET_EXTENDED_FUSE_BITS, GET_HIGH_FUSE_BITS или GET_LOCK_BITS.
Примечание: возвращается физически реальное значение соответствующего байта, т.е. запрограммированные биты являются нулями, а незапрограммированные биты – единицами.

boot_signature_byte_get()
boot_signature_byte_get(addr)
Описание: возвращает байт из области сигнатуры микроконтроллера. Для микроконтроллеров, поддерживающих калибровочные байты встроенных RC-генераторов, этот макрос так же может возвращать и эти значения. Значение **addr** может находиться в пределах от **0** до **0x1F** – см. документацию на примененный микроконтроллер.

boot_page_fill()
boot_page_fill(address, data)
Описание: записывает во временный буфер страницы по указанному адресу **address** заданные данные **data**.
Примечание: **address** – это адрес байта, **data** – это 16-битное значение (слово). ***AVR адресуют память побайтно, но за один раз записывают слово!*** Правильное использование макроса заключается в том, чтобы от обращения к обращению увеличивать значение **address** на 2, а в **data** помещать сразу по 2 байта, как единое слово.

boot_page_erase()
boot_page_erase(address)
Описание: выполняет стирание страницы памяти программ.
Примечание: **address** – это адрес байта, а не слова.

boot_page_write()
boot_page_write(address)
Описание: выполняет запись буфера в указанную **address** страницу памяти программ.
Примечание: **address** – это адрес байта, а не слова.

boot_rww_enable()
boot_rww_enable()
Описание: включает доступ к RWW-секции памяти программ.

boot_lock_bits_set()
boot_lock_bits_set(lock_bits)
Описание: устанавливает биты защиты загрузчика (BLB-биты).
Примечания:
1. Параметр **lock_bits** в данном макросе должен содержать единицы в тех позициях, которые должны быть запрограммированы, т.е. записаны в ноль.
2. Макрос устанавливает только BLB-биты защиты.
3. Как и любые биты защиты, единожды установленные BLB-биты могут быть стерты только при полном стирании памяти микроконтроллера.

boot_page_fill_safe()
boot_page_fill_safe(address, data)
Описание: выполняет то же действие, что и boot_page_fill(), но перед этим дожидается завершения ранее начатой операции самопрограммирования.

boot_page_erase_safe()
boot_page_erase_safe(address)
Описание: выполняет то же действие, что и boot_page_erase(), но перед этим дожидается завершения ранее начатой операции самопрограммирования.

boot_page_write_safe()
boot_page_write_safe(address)
Описание: выполняет то же действие, что и boot_page_write(), но перед этим дожидается завершения ранее начатой операции самопрограммирования.

boot_rww_enable_safe()
boot_rww_enable_safe()
Описание: выполняет то же действие, что и boot_rww_enable(), но перед этим дожидается завершения ранее начатой операции самопрограммирования.

boot_lock_bits_set_safe()
boot_lock_bits_set_safe(lock_bits)
Описание: выполняет то же действие, что и boot_lock_bits_set(), но перед этим дожидается завершения ранее начатой операции самопрограммирования.

avr/eeprom.h – Поддержка EEPROM AVR

Этот модуль содержит ряд простых функций и макросов, реализующих основные операции для работы со встроенным EEPROM микроконтроллеров AVR. Все функции действуют по принципу «ожидания», т.е. достаточно длительные. Если требуется реализация работы по прерываниям для максимально эффективного кода, она должна быть реализована пользователем самостоятельно.

Все функции выполняют проверку готовности EEPROM для очередного обращения, но они не контролируют состояние автомата записи в память программ (т.е. самопрограммирования). Если ваша программа содержит код самопрограммирования, вы должны самостоятельно реализовать проверку завершения любой операции самопрограммирования перед обращением к EEPROM.

Так как функции используют фиксированные регистры для работы, они не реентерабельны, т.е. если эти функции вызываются и из обычного цикла программы и из обработчиков прерываний, следует запрещать глобально прерывания перед обращениям к этим функциям из основной программы.

Определения модуля

Модуль вводит ряд определений (констант и макросов), часть из которых позволяет упростить совместимость исходного текста со средой разработки IAR.

_EEPUT()
_EEPUT(addr, val)
Описание: макрос, осуществляющий запись байта val в EEPROM по адреса addr. Введен для совместимости с IAR.

_EEGET()
_EEGET(var, addr)
Описание: макрос, осуществляющий чтение байта из EEPROM по адресу addr. Введен для совместимости с IAR.

EEMEM
EEMEM
Описание: атрибут, указывающий на то, что переменная должна размещаться в EEPROM. Пример использования: EEMEM int k; - переменная k должна быть размещена в EEPROM.

eeprom_is_ready()
eeprom_is_ready()
Описание: макрос, возвращающий не нулевое значение, если EEPROM готова для очередного обращения, 0 – в противном случае.

eeprom_busy_wait()
eeprom_busy_wait()
Описание: макрос, выполняющий цикл ожидания завершения операции записи в EEPROM.

Функции модуля

Большинство функций реализованы как **static inline**-функции или макросы-функции.

eeprom_read_byte()
uint8_t eeprom_read_byte (const uint8_t *p)
Параметры:
const uint8_t ***p** – указатель на байт в EEPROM
Возвращаемое значение: считанный из EEPROM байт.
Описание: функция выполняет считывание байта по указанному адресу из EEPROM.

eeprom_read_word()
uint16_t eeprom_read_word (const uint16_t *p)
Параметры:
const uint16_t ***p** – указатель на слово в EEPROM
Возвращаемое значение: считанное из EEPROM слово.
Описание: функция выполняет считывание 16-битного значения из EEPROM.

eeprom_read_dword()
uint32_t eeprom_read_dword (const uint32_t *p)
Параметры:
const uint32_t ***p** – указатель на двойное слово в EEPROM
Возвращаемое значение: считанное из EEPROM двойное слово.
Описание: функция выполняет считывание 32-битного числа из EEPROM.

eeprom_read_block()
void eeprom_read_block (void *dst, const void *src, size_t n)
Параметры:
void ***dst** – указатель на начало области в ОЗУ
const void ***src** – указатель на начало области EEPROM
size_t **n** – количество байт
Возвращаемое значение: нет.
Описание: функция выполняет считывание **n** байтов из EEPROM по адресу **src** и размещает их в ОЗУ, начиная с адреса **dest**.

eeprom_write_byte()
void eeprom_write_byte (uint8_t *p, uint8_t value)
Параметры:
uint8_t ***p** – указатель на ячейку в EEPROM

uint8_t **value** – записываемый байт
Возвращаемое значение: нет.
Описание: функция выполняет запись байта **value** в EEPROM по адресу **p**.

eeprom_write_word()
void eeprom_write_word (uint16_t *p, uint16_t value)
uint16_t ***p** – указатель на ячейку в EEPROM
uint16_t **value** – записываемое слово
Возвращаемое значение: нет.
Описание: функция выполняет запись 16-битного числа **value** в EEPROM по адресу **p**.

eeprom_write_dword()
void eeprom_write_dword (uint32_t *p, uint32_t value)
uint32 ***p** – указатель на ячейку в EEPROM
uint32 **value** – записываемое двойное слово
Возвращаемое значение: нет.
Описание: функция выполняет запись 32-битного числа **value** в EEPROM по адресу **p**.

eeprom_write_block()
void eeprom_write_block (const void *src, void *dst, size_t n)
Параметры:
void ***src** – указатель на начало области ОЗУ
void ***dst** – указатель на начало области в EEPROM
size_t **n** – количество байт
Возвращаемое значение: нет.
Описание: функция осуществляет запись **n** байт из области ОЗУ **src** в область EEPROM **dest**.

Следует признать, что модуль **eeprom.h** от релиза к релизу WinAVR претерпевает существенные модификации, в частности, неоднократно менялись местами параметры функций блочного чтения-записи, что, разумеется, порождает определенные сложности с совместимостью ранее разработанного кода и новой версии компилятора. Поэтому рекомендуется тщательно изучить документацию на свежий релиз WinAVR, т.к. в нем могут быть сюрпризы.

Из числа приятных сюрпризов, появившихся с версии WinAVR **20100110**, можно назвать появление «бережных» функций записи EEPROM **eeprom_update_xxxx()**, где **xxxx** – это суффикс **byte**, **word**, **dword** или **block**. Параметры этих функций такие же, как у соответствующих **eeprom_write_xxxx()**, а отличие в том, что новые функции перед тем, как записать новое значение в ячейку EEPROM, проверяют ее текущее значение и, если оба значения одинаковы, то запись в ячейку не производится. Так как ресурс записей EEPROM AVR ограничен 100000 записей, использование новых функций позволяет увеличить живучесть EEPROM без принятия каких-то особых мер.

Рекомендуется использовать эти функции всегда.

Врезка. Пример использования.

```
#include <avr/io.h>
FUSES ={
.low = LFUSE_DEFAULT,
.high = (FUSE_BOOTSZ0 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN),
.extended = EFUSE_DEFAULT,
};

int main(void){
return 0;
}
```

avr/fuse.h – Поддержка fuse-битов AVR

Модуль позволяет разместить в особой секции ELF-файла информацию о состоянии **fuse**-битов, которую может извлекать программное обеспечение программатора для установки **fuse**-битов при программировании микроконтроллера. Размещение этой информации в ELF-файле осуществляется при компоновке программы. К сожалению, в HEX-формате сведения о состоянии **fuse**-битов размещены быть не могут.

Данный модуль – универсальный, т.е. ориентирован на весь спектр поддерживаемых микроконтроллеров. Необходимо обязательно подключать к проекту модуль **io.h**, в котором определяются глобальные параметры, используемые затем в модуле **fuse.h**. Сам модуль **fuse.h** подключается в этом случае автоматически, поэтому нужды в явном его подключении нет.

В зависимости от типа примененного микроконтроллера **fuse**-биты могут располагаться в младшем, старшем или расширенном **fuse**-байте. Для каждого отдельно взятого fuse-бита вводятся константы-маски, соответствующие наименованию **fuse**-бита, например, **FUSE_SPIEN**, **FUSE_RSTDSBL** и т.д.[34]. Если необходимо установить несколько fuse-битов в одном байте, нужно объединять их при помощи битовой операции **AND**, т.е. так: (**FUSE_SPIEN** & **FUSE_RSTDISBL** & ….).

Для каждого из **fuse**-байтов конкретного микроконтроллера определены значения **fuse**-битов по умолчанию, которые следует применять, если изменять их не требуется: **LFUSE_DEFAULT** для младшего байта, **HFUSE_DEFAULT** для старшего и **EFUSE_DEFAULT** для расширенного.

Модуль определяет всего одну глобальную переменную-структуру **FUSES**, к заданию начальных значений для которой и сводится вся работа с **fuse**-битами. Данная переменная в зависимости от микроконтроллера может быть разного типа: если микроконтроллер содержит всего один **fuse**-байт, то эта переменная является просто переменной типа **int8_t**; если микроконтроллер имеет 2 **fuse**-байта, то **FUSES** представляет собой структуру с 2-я полями **low** и **high** соответственно типа **int8_t** каждое; для микроконтроллеров с тремя **fuse**-байтами **FUSES** представляет собой структуру с тремя полями типа **int8_t** – **low**, **high** и **extended**. Наконец, если МК содержит более 3-х **fuse**-байтов, **FUSES** определена как массив байтов соответствующей длины.

Следует учитывать, что компоновщик использует только значения переменной **FUSES**, которые были указаны на этапе начальной инициализации, и игнорирует все изменения, которые могли быть сделаны в ходе выполнения программы.

Пример использования см. на ***врезке***.

Важно, что **всегда** необходимо инициализировать **все** поля структуры FUSES.

Суть рассматриваемого модуля заключена в том, что получаемый в результате компиляции elf-файл содержит абсолютно всю информацию, необходимую для правильного программирования микроконтроллера. Остается только решить вопрос, какое именно программное обеспечение в состоянии воспользоваться этой информацией. В какой-то мере эту задачу может решить утилита **avrdude**, входящая в состав WinAVR, однако, только в связке с другой утилитой **avr-objcopy**. Avrdude – это универсальный программатор, поддерживающий огромное количество схем для прошивки микроконтроллеров AVR, а утилита **avr-objcopy** позволяет извлекать из elf-файла информационные блоки о flash, eeprom и fuse-битах (из соответствующих секций) и сохранять их в виде hex-файлов, которые уже могут использоваться **avrdude** для прошивки. К сожалению, обе эти утилиты имеют немалое количество опций запуска, поэтому в рамках данного изложения не рассматриваются (это планируется сделать позже).

avr/interrupt.h – Прерывания

Общие сведения

При обработке прерываний происходит конфликт между декларируемой Си независимостью от аппаратуры, и жесткой привязке векторов прерываний к этой самой аппаратуре. Таким образом, любая реализация обработки прерываний на Си становится уникальной для конкретной среды программирования.

В WinAVR принято, что каждому определенному вектору прерывания сопоставлена особым образом поименованная функция-обработчик. Определение в

[34] Т.к. fuse-биты отличаются в зависимости от модели микроконтроллера, следует ознакомиться с их наименованием по соответствующей документации к выбранному контроллеру; в настоящей документации они не приводятся.

программе функции, названной в соответствии с определенными соглашениями, приводит к тому, что компилятор генерирует для нее нестандартный код пролога и эпилога, в частности, обязательно сохраняет SREG в начале и восстанавливает в конце, а так же использует для возврата из функции команду RETI. Кроме того, в таблице векторов прерываний для каждой определенной пользователем функции-обработчика помещается команда перехода к соответствующей функции.

Далее кратко перечислены основные характерные особенности реализации обработчиков прерываний WinAVR. Подробности следует искать в описаниях соответствующих макросов.

Детали реализации этого подхода остаются за кадром, т.к. программисту достаточно определить при помощи макроса ISR() функцию – и она будет назначена (и соответственно оформлена) обработчиком определенного прерывания.

Особенность аппаратного строения AVR заключается в том, что во время работы одного обработчика прерываний все прочие прерывания запрещены. Такая ситуация в некоторых случаях может быть нежелательной, для чего можно (естественно, осознавая последствия, к которым это может привести) разрешить прерывания принудительно внутри функции-обработчика, используя макрос sei(). Но можно поступить и проще, указав для макроса ISR() параметр ISR_NOBLOCK, что заставит компилятор поместить соответствующую инструкцию глобального разрешения прерываний сразу после пролога функции (т.е. инициализирующего участка кода).

При разработке программ не исключена ситуация, когда случайно будет сгенерировано прерывание, для которого нет назначенного обработчика (ситуация «баг программы»). По умолчанию компилятор генерирует код, который в этом случае приведет к переходу на адрес 0x0000, т.е. поведение кода равносильно сбросу микроконтроллера. Однако такое поведение можно изменить, определив особый обработчик «неверного прерывания», который автоматически назначается все «незанятым» векторам прерываний. Делается это при помощи указания для ISR() «плохого» вектора BADISR_vect. Такой обработчик будет использован для всех векторов прерываний, для которых не указаны явно их обработчики.

Так же часто может возникать ситуация, когда один и тот же обработчик должен использоваться для нескольких прерываний. Можно, конечно, определить несколько абсолютно идентичных функций, однако, можно и сэкономить память, указав для ISR() параметр ISR_ALIASOF().

Еще одной занимательной ситуацией, связанной с прерываниями, является пробуждение микроконтроллера по прерыванию из «спящего» режима. В этом случае обработчик прерывания вызывается, однако, часто в нем никаких полезных действий выполняться не должно. Для таких случаев предусмотрен макрос EMPTY_INTERRUPT(), позволяющий определить обработчик, который не делает ничего.

Наконец, во многих случаях может оказаться, что неизбежно генерируемый компилятором для обработчика прерывания код пролога и эпилога оказывается слишком велик. Было бы удобно в таком случае использовать обработчик, реализованный на ассемблере, либо попытаться обойтись средствами Си для получения более оптимального кода. Сделать это можно при помощи параметра ISR_NAKED, указываемого для ISR(). В этом случае компилятор не генерирует вообще никакого пролога и эпилога, поэтому программист сам должен обеспечить сохранение контекста программы (в частности, содержимого SREG), и, главное, использовать макрос reti() для выхода из обработчика.

Для каждого микроконтроллера определены свои константы для указания соответствующего вектора прерывания. Эти константы содержат в своем составе суффиксс **_vect**. Для совместимости с предыдущими версиями WinAVR сохранена и возможность определять обработчик прерывания при помощи «сигналов», т.е. констант, имеющих «префикс» **SIG_** (префиксы и суффиксы соответствующих векторов определены в заголовочных файлах для каждого типа микроконтроллеров). Рекомендуется использовать «новую» форму **_vect**.

Макросы и определения

sei()

sei() – глобальное разрешение прерываний

cli()

cli() – глобальное запрещение прерываний

reti()

reti() – макрос принудительно помещает в код программы инструкцию RETI, обеспечивая возврат из прерывания.

ISR()

ISR(vector, attributes) – макрос для определения обработчика прерываний.

Первый параметр **vector** – это одна из констант, определяющих номер вектора прерывания. Реальные константы определены в соответствующем заголовочном файле модели микроконтроллера (оканчиваются на суффикс **_vect**, например, **ADC_vect**). Кроме этого можно использовать константу BADISR_vect для задания обработчика «по умолчанию».

Второй параметр **attributes** – необязательный, определяет опциональные варианты реализации соответствующего обработчика. Допустимо использовать несколько атрибутов из числа следующих (атрибуты разделяются запятыми):

ISR_BLOCK – атрибут по умолчанию, означает запрет прерываний, пока не отработает текущий обработчик

ISR_NOBLOCK – разрешает глобально прерывания сразу после пролога обработчика

ISR_NAKED – указывает, что **вся реализация** кода обработчика делается программистом (нет никакого пролога и эпилога)

ISR_ALIASOF(v) – позволяет назначить текущему обработчику несколько векторов (см. врезку для ISR_ALIASOF).

SIGNAL()

SIGNAL(vector) – макрос определения обработчика, сохраненный исключительно для совместимости со старым кодом. Не рекомендуется использование в новых разработках. В качестве параметра **vector** должен получать константу с префиксом **SIG_** из заголовочного файла конкретного микроконтроллера (например, **SIG_ADC**).

EMPTY_INTERRUPT()

EMPTY_INTERRUPT(vector) – макрос определения «пустого» обработчика, т.е. состоящего только из команды RETI. В качестве параметра принимаются те же значения, что и для ISR().

ISR_ALIAS()

ISR_ALIAS(vector, target_vector) – макрос, связывающий два вектора в один обработчик. Не рекомендуется использование в новых разработках. В качестве параметра vector и target_vector используется любая из констант, как и для ISR(), причем параметр target_vector к моменту вызова макроса должен быть уже определен (см. *врезку ISR_ALIAS*):

ISR_ALIASOF()

ISR_ALIASOF(target_vector) – макрос, используемый в качестве опционального параметра ISR() для того, чтобы связать обработчик, определенный ISR(), еще и с вектором target_vector (см. *врезку ISR_ALIASOF*).

Врезка ISR_ALIAS

```
// определяем обработчик внешнего запроса прерывания №0
ISR(INT0_vect){
PORTB = 42;
}
// назначаем этот обработчик и для запроса №1
ISR_ALIAS(INT1_vect, INT0_vect);
```

Врезка ISR_ALIASOF

```
// определяем обработчик внешнего запроса прерывания №0 и одновременно №1
ISR(INT0_vect, ISR_ALIASOF(INT1_vect)){
PORTB = 42;
}
```

BADISR_vect

BADISR_vect – константа «псевдовектора», обозначающая любой явно не заданный вектор. По умолчанию соответствует переходу на начало программной памяти, вызывая переинициализацию всего кода при поступлении «необрабатываемого» прерывания.

ISR_BLOCK

ISR_BLOCK – опциональный атрибут, используемый в ISR(), и обозначающий, что прерывания будут запрещены в течение работы всего обработчика прерывания. Если этот атрибут не задан явно (т.е. используется ISR() без атрибутов вообще), прерывания все равно будут запрещены.

ISR_NOBLOCK

ISR_NOBLOCK – опциональный атрибут, используемый в ISR(), и обозначающий, что прерывания будут разрешены в течение работы всего обработчика прерывания. Это позволит быстрее отреагировать на ожидающие в очереди другие запросы прерываний, однако увеличивает требования к объему стека.

ISR_NAKED

ISR_NAKED – опциональный атрибут, используемый в ISR(), и обозначающий, что компилятор не должен добавлять пролог и эпилог к функции. В этом случае программист обязан сам выполнить все необходимые меры, включая выход из обработчика макросом reti().

avr/io.h – Специфичные для AVR функции ввода-вывода

Этот заголовочный файл по существу реализует автоматическое подключение одного конкретного файла, соответствующего выбранному микроконтроллеру. Во всяком случае, для определения констант и макросов портов ввода-вывода и т.п. периферии программист более не должен применять никаких усилий, кроме подключения этого файла к проекту и задания типа микроконтроллера. Фактически происходит подключение целого ряда заголовочных файлов, в которых определены константы и макросы для обращения к различной периферии микроконтроллера (см. например avr/sfr_defs.h – Регистры специальных функций AVR).

Рассмотрение всех этих определений опущено, т.к. подразумевается, что программист, работающий с AVR, вполне информирован о его периферии. Из прочего стоит обратить внимание лишь на следующие константы:

RAMEND – соответствует адресу последней существующей ячейки ОЗУ, т.е. в сущности, определяет размер ОЗУ выбранного микроконтроллера.

XRAMEND – соответствует адресу последней существующей ячейки внешнего или дополнительно адресуемого ОЗУ (если его поддержка имеется в микроконтроллере).

E2END – соответствует адресу последней ячейки встроенного EEPROM

FLASHEND – соответствует адресу последней ячейки памяти программ, т.е. фактически определяет размер сегмента кода.

SPM_PAGESIZE – соответствует размеру страницы для операций самопрограммирования (т.е. записи из программы в сегмент кода).

avr/pgmspace.h – Поддержка обращения к сегменту кода AVR

В этом модуле определяются ряд функций и макросов, позволяющих обращаться к данным, хранящимся в памяти программ, т.е. сегменте кода. Для этого необходимо, чтобы микроконтроллер поддерживал инструкции LPM или ELPM.

Функции этого модуля – попытка облегчить процесс адаптации программ для IAR, однако полной совместимости нет по принципиальным особенностям GCC.

Если вы работаете со строками, которые постоянно размещены в ОЗУ, вы можете использовать функции для работы с ними из модуля string.h – Строки, но для неизменных строк более удачным будет размещение их в сегменте кода. В этом случае вы можете работать с ними при помощи функций этого модуля, которые совпадают по имени с аналогичными из string.h, но имеют суффикс _P – признак того, что строка находится в программной памяти (такой принцип так же использовался и в модуле stdio.h – Стандартный ввод-вывод).

Однако следует учитывать, что функции из настоящего модуля могут работать только со строками, размещенными в первых 64K адресного пространства микроконтроллера, т.к. не используют инструкцию ELPM. Это не страшно в обычных случаях. Но может вызвать проблемы, если имеется много текстовых констант или же используется самозагрузка кода. При написании программы вы не должны беспокоиться о месте размещения строк, т.к. компоновщик автоматически разместит все данные в программной памяти сразу после векторов прерывания.

Так же следует помнить о том, что все используемые для обращения к сегменту кода указатели представляют себя адреса байтов (хотя программная адресация у AVR – 16-битная).

Определения модуля

Модуль определяет ряд констант, атрибутов и макросов, облегчающих разработку программ, работающих с данными в сегменте кода.

PROGMEM

PROGMEM – макрос, устанавливающий атрибут, указывающий, что переменная на самом деле представляет собой константу в сегменте кода

PSTR()

PSTR(s) – макрос, используемый для определения указателя на строку **s**, размещенную в сегменте кода

PGM_P

PGM_P – макрос, определяющий указатель на символ в сегменте кода

PGM_VOID_P

PGM_VOID_P – макрос, определяющий указатель на объект в сегменте кода

pgm_read_byte_near()

pgm_read_byte_near(address_short) – макрос, возвращающий значение байта, считанного из программной памяти по адресу address_short (в первых 64K).

pgm_read_word_near()

pgm_read_word_near(address_short) – макрос, возвращающий значение 16-битного числа, считанного из программной памяти по адресу address_short (в первых 64K).

pgm_read_dword_near()

pgm_read_dword_near(address_short) – макрос, возвращающий значение 32-битного числа, считанного из программной памяти по адресу address_short (в первых 64K).

pgm_read_byte_far()

pgm_read_byte_far(address_long) – макрос, возвращающий значение байта, считанного из программной памяти по адресу address_long (32 бита, нет ограничения в 64K).

pgm_read_word_far()

pgm_read_word_far(address_long) – макрос, возвращающий значение 16-битного числа, считанного из программной памяти по адресу address_long (32 бита, нет ограничения в 64K).

pgm_read_dword_far()

pgm_read_dword_far(address_long) – макрос, возвращающий значение 32-битного числа, считанного из программной памяти по адресу address_long (32 бита, нет ограничения в 64K).

pgm_read_byte()

pgm_read_byte(address_short) – аналог pgm_read_byte_near()

pgm_read_word()

pgm_read_word(address_short) – аналог pgm_read_word_near()

pgm_read_dword()

pgm_read_dword(address_short) – аналог pgm_read_dword_near()

Типы, вводимые модулем

Модуль вводит ряд типов, которые просто облегчают определение констант в сегменте кода. По смыслу все эти типы аналогичны определенным в inttypes.h – Целочисленные преобразования, только содержат префикс **prog_**, означающий, что соответствующий объект размещается в памяти программ. В связи с этим расшифровка значения типов не приводится.

prog_void
prog_char
prog_uchar
prog_int8_t
prog_uint8_t
prog_int16_t

prog_uint16_t
prog_int32_t
prog_uint32_t
prog_int64_t
prog_uint64_t

Функции модуля

Так как все функции не более чем аналоги string.h – Строки, вместо подробного описания будут указаны ссылки на функции для работы со строками в ОЗУ. Единственное, на что должен обратить программист, это типы передаваемых в функции параметров и возвращаемых значений: как правило, все указатели в функциях этого модуля – это указатели на объекты в сегменте кода.

memchr_P()
PGM_VOID_P memchr_P (PGM_VOID_P s, int val, size_t len)
Функция ищет в области программной памяти **s** символ, возвращая указатель на него или NULL, если не найдено. Подробности см. в memrchr()

memcmp_P()
int memcmp_P (const void *s1, PGM_VOID_P s2, size_t len)
Функция сравнивает область ОЗУ **s1** и памяти программ **s2**. Подробности см. в memcmp()

memcpy_P()
void * memcpy_P (void *dest, PGM_VOID_P src, size_t len)
Функция копирует в область ОЗу **dest** содержимое области памяти программ **src**. Подробности см. в memcpy()

memrchr_P()
PGM_VOID_P memrchr_P (PGM_VOID_P s, int val, size_t len)
Функция ищет в области программной памяти **s** символ. Подробности см. в memrchr() и memchr_P()

strcasecmp_P()
int strcasecmp_P (const char *s1, PGM_P s2)
Функция сравнивает строку s1 в ОЗУ со строкой в памяти программ s2 без учета регистра символов. См. strcasecmp()

strcat_P()
char * strcat_P (char *dest, PGM_P src)
Функция осуществляет конкатенацию строки **src** из памяти программ к строке **dest** в ОЗУ. См. strcat()

strchr_P()
PGM_P strchr_P (PGM_P s, int val)
Функция ищет символ в строке **s** в программной памяти. См. strchr()

strchrnul_P()
PGM_P strchrnul_P (PGM_P s, int val)
Функция, аналогичная strchr_P(), за исключением того, что всегда возвращает не-NULL указатель: если символ не найден, возвращается указатель на завершающий ноль строки **s**

strcmp_P()
int strcmp_P (const char *s1, PGM_P s2)
Функция аналогична strcmp(), за исключением того, что **s2** находится в программной памяти.

strcpy_P()
char * strcpy_P (char *s1, PGM_P s2)
Функция аналогична strcpy(), за исключением того, что **s2** находится в программной памяти.

strcspn_P()
size_t strcspn_P (const char *s, PGM_P reject)
Функция аналогична strcspn(), за исключением того, что **reject** находится в программной памяти.

strlcat_P()
size_t strlcat_P (char *dest, PGM_P src, size_t siz)
Функция аналогична strlcat(), за исключением того, что **src** находится в программной памяти

strlcpy_P()
size_t strlcpy_P (char *dest, PGM_P src, size_t siz)
Функция аналогична strlcpy(), за исключением того, что **src** находится в программной памяти

strlen_P()
size_t strlen_P (PGM_P s)
Функция аналогична strlen(), за исключением того, что **s** находится в программной памяти

strncasecmp_P()
int strncasecmp_P (const char *s1, PGM_P s2, size_t n)
Функция аналогична strncasecmp(), за исключением того, что **s2** находится в программной памяти

strncat_P()
char * strncat_P (char *dest, PGM_P src, size_t len)
Функция аналогична strncat(), за исключением того, что **src** находится в программной памяти

strncmp_P()
int strncmp_P (const char *s1, PGM_P s2, size_t n)
Функция аналогична strncmp(), за исключением того, что **s2** находится в программной памяти

strncpy_P()
char * strncpy_P (char *dest, PGM_P src, size_t n)
Функция аналогична strncpy(), за исключением того, что **src** находится в программной памяти

strnlen_P()
size_t strnlen_P (PGM_P src, size_t len)
Функция аналогична strnlen(), за исключением того, что **src** находится в программной памяти

strpbrk_P()
char * strpbrk_P (const char *s, PGM_P accept)
Функция аналогична strpbrk(), за исключением того, что **accept** находится в программной памяти

strrchr_P()
PGM_P strrchr_P (PGM_P s, int val)
Функция аналогична strrchr(), за исключением того, что **s** находится в программной памяти

strsep_P()
char * strsep_P (char **sp, PGM_P delim)
Функция аналогична strsep(), за исключением того, что **delim** находится в программной памяти

strspn_P()
size_t strspn_P (const char *s, PGM_P accept)
Функция аналогична strspn(), за исключением того, что **accept** находится в программной памяти

strstr_P()
char * strstr_P (const char *s1, PGM_P s2)
Функция аналогична strstr(), за исключением того, что **s2** находится в программной памяти

memmem_P()
void * memmem_P (const void *s1, size_t len1, PGM_VOID_P s2, size_t len2)
Функция аналогична memmem(), за исключением того, что **s2** находится в программной памяти

strcasestr_P()
char * strcasestr_P (const char *s1, PGM_P s2)
Функция аналогична strcasestr(), за исключением того, что **s2** находится в программной памяти

Продолжение в №2/2011