Книга по работе с WinAVR и AVR Studio

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

Продолжение. Начало в №1-12/2010; №1-2/2011

РАСПРЕДЕЛЕНИЕ ПАМЯТИ AVR-GCC

Компилятор GCC, адаптированный под архитектуру AVR, использует свои собственные соглашения о том, как распределяется память во время компиляции программы и во время ее исполнения. Знание этих особенностей позволит программисту более тонко влиять на ход своей работы, добиваясь желаемого результата, порой недостижимого без этих знаний.

Секции памяти

При компиляции программы происходит распределение результатов работы компилятора по различным областям памяти микроконтроллера, называемыми секциями (некий аналог традиционно используемым *сегментам* памяти). Компилятор размещает данные и исполняемые коды по этим секциям, а уже компоновщик затем собирает эти секции в блоки, которые в итоге предназначаются для программирования во flash-память программ микроконтроллера, EEPROM данных и т.п.

Каждая секция должна иметь символьное имя. Существует ряд предопределенных секций, и может быть создано любое количество пользовательских секций. Каждая секция характеризуется областью адресов конкретного типа памяти, внутри которой размещается ее содержимое.

В общих чертах суть использования секций можно пояснить следующим примером. Предположим, необходимо разместить какую-либо функцию в строго определенной области памяти программ (это часто бывает необходимо для микроконтроллерных систем, например для загрузчиков – см. avr/boot.h – Поддержка загрузчиков AVR). Для решения этой задачи необходимо определить пользовательскую секцию, начинающуюся с нужного адреса, а затем в тексте программы для нужной функции указать атрибут **section(«.user»)**, где «.user» – это заданное имя секции. В результате компоновщик разместит функцию как раз начиная с начала секции памяти, что и требовалось.

Разумеется, аналогичным образом можно определить секции в областях EEPROM или области ОЗУ, «привязав» таким образом переменные к конкретным адресам.

Далее рассмотрены основные стандартные секции памяти, поддерживаемые компилятором.

.text

Основная секция для сегмента кода. В нее помещаются все пользовательские функции, т.е. то, что определено в тексте программы (отсюда, очевидно, и название секции).

.data

Секция *статически проинициализированных* переменных пользователя. Если программист использует определения типа

```
char  str[] = “Это пример”;
long counter = 12345;
```

то эти переменные размещаются в секции **.data**.

Имеется возможность указать адрес начала этой секции принудительно при помощи опций компилятора **-Wl,-Tdata,<addr>**, где addr – желаемый адрес начала секции (естественно, угловые скобки не нужны). Следует помнить лишь о том, что компоновщик автоматически отнимает[38] от значения addr число 0x800000, т.е. если нужно начать секцию **.data** с адреса 0x1100, следует использовать addr=0x801100.

.bss

Секция глобальных и статических переменных, которые не инициализируются значениями, указанными пользователем, т.е. получают значение 0 по умолчанию.

[38] Так реализовано, очевидно, для совместимости с GCC, хотя реально для AVR в этом нет никакого смысла.

.noinit

Эта секция – часть секции .bss, и содержит вообще никак не инициализируемые переменные. Попытка указать атрибут размещения в этой секции для переменной, содержащей значение по умолчанию, вызовет ошибку компиляции.

Имеется возможность указать адрес начала этой секции принудительно при помощи опций компилятора **-Wl,-section-start=.noinit=<addr>**, где addr – желаемый адрес начала секции (естественно, угловые скобки не нужны). По отношению к значению addr действуют те же условности, что и для секции .data.

.init0 … .init9

Определено 10 секций начальной инициализации, являющихся частью секции .text. Номер секции определяет ее очередность, т.е. код в секции **.init1** однозначно будет выполнен раньше кода из секции **.init2**, по после кода секции **.init0**.

Важно представлять себе, что в этих секциях не используются вызовы функций, т.е. используется «плоский» код. Таким образом, функция, объявляемая в любой из этих секций, не должна использовать оператор возврата, т.к. не будет вызвана. Иначе говоря, все функции в этих секциях должны быть объявлены с атрибутом **NACKED**, и не должны вызываться из основной программы.

.init0 – самая первая секция инициализации. Код, объявленный в этой секции, будет исполнен немедленно после аппаратного сброса.

.init1, **.init3**, **.init5**, **.init7** и **.init8** – не используются по умолчанию, программист может определять в них свой код (помня о порядке их исполнения).

.init2 – в этой секции происходит инициализация указателя стека и очистка регистра ***r0*** (этот регистр используется, как вспомогательный в других инициализирующих секциях, нельзя менять его содержимое, т.к. это нарушит работу других секций).

.init4 – для микроконтроллеров с объемом ПЗУ программ более 64K в этой секции находится код, инициализирующий содержимое секции **.data**, т.е. копирующий в ОЗУ данные из памяти программ. Для всех прочих микроконтроллеров в этой секции находится код, обнуляющий секцию **.bss**.

.init6 – не используется в С-программах, но для С++ в этой секции реализуются конструкторы классов.

.init9 – это по существу переход к началу функции **main()**.

Примечание. Остается не раскрытым вопрос о том, в какой именно секции происходит инициализация секции **.data** для микроконтроллеров с объемом памяти менее 64K. Известно, что этот инициализирующий код является частью секции .text. Так же очень важно помнить, что стек инициализируется только в секции **init2**, поэтому код, размещаемый в «предыдущих» секциях не может использовать обращения к функциям.

.fini9 … .fini0

Определено 10 завершающих секций, так же являющихся частью секции .text. Эти секции содержат код (ограничения те же, что и для .init0 … .init9), последовательно (в порядке убывания номера секции) исполняющийся при завершении программы.

.fini9 – соответствует началу исполнения функции exit(). По умолчанию компилятором не используется.

.fini6 – в С-программах не используется, а в С++ в этой секции размещаются деструкторы.

.fini0 – содержит запрет прерываний и бесконечный цикл, означающий остановку программы.

.fini8, **.fini7**, **.fini5**, **.fini4**, **.fini3**, **.fini2** и **.fini1** – по умолчанию компилятором не используются, программист может размещать в них свой код.

.bootloader

Секция кода загрузчика. Адрес начала и размер этой секции должен соответствовать заданным fuse-битами параметрам микроконтроллера (обычно задается в опциях проекта и автоматически настраивается компоновщиком).

.eeprom

Секция данных EEPROM. Обычно нет необходимости работать непосредственно с этой секцией, т.к. все необходимые действия по размещению данных в EEPROM успешно осуществляются более простыми способами (см. avr/eeprom.h – Поддержка EEPROM AVR), хотя реализация этих способов все равно заключается в указании соответствующего атрибута, т.е. указания секции размещения объекта, только это скрыто от программиста.

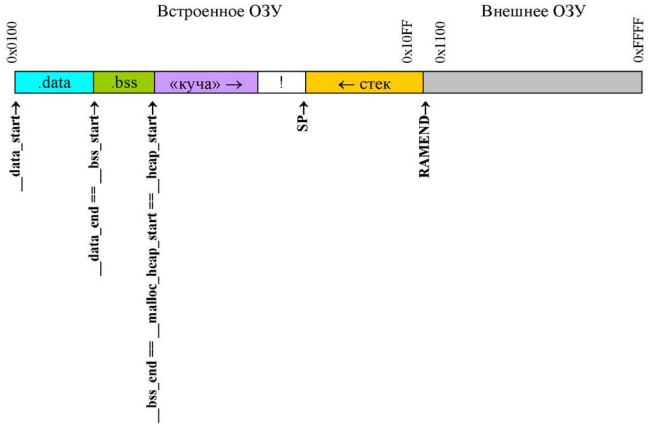
Динамическое распределение памяти

Компилятор AVR-GCC поддерживает микроконтроллеры, имеющие довольно мало ОЗУ (минимально допустимое поддерживаемое количество ОЗУ – 128 байт)[39], в то время как основой самого языка Си является активная работа с ОЗУ, которое необходимо для организации стека, динамического выделения памяти, хранения локальных переменных и т.п.

Кроме этого, в AVR нет никакой аппаратной поддержки какого-либо управления памятью, например, для контроля «пересечения» упомянутых областей ОЗУ (как это реализовано, например, для PC).

Традиционно компилятор размещает секцию .data с самого начала доступной области ОЗУ, после чего следует секция .bss. Так называемая «куча» (heap) или область динамически распределяемой памяти будет следовать сразу за .bss. Стек начинается с вершины (т.е. последней доступной ячейки памяти) и движется в сторону уменьшения адресов. При таком распределении есть гарантия, что динамически выделяемые области никогда не пересекутся со статически распределенной памятью (если, конечно, не было ошибок программиста), но нет гарантии, что область стека не пересечется с динамически распределенными областями памяти данных. Это возможно, например, при рекурсивном обращении к функциям, если функции имеют большое количество локальных переменных или если область кучи сильно фрагментирована.

Рисунок поясняет типичное распределение ОЗУ[40].



Для устройств на микроконтроллерах подобный подход непрост, т.к. нужно обеспечить минимальный размер кода, реализующего динамическое выделение памяти и контроль «границ» при обеспечении минимальной фрагментации ОЗУ, и в то же время высокое быстродействие, т.к. микроконтроллеры все еще очень медленные по сравнению с «большими» компьютерами. Программная реализация этих требований стремится к их решению, предлагая при этом некоторые средства и указывая направления для их оптимизации.

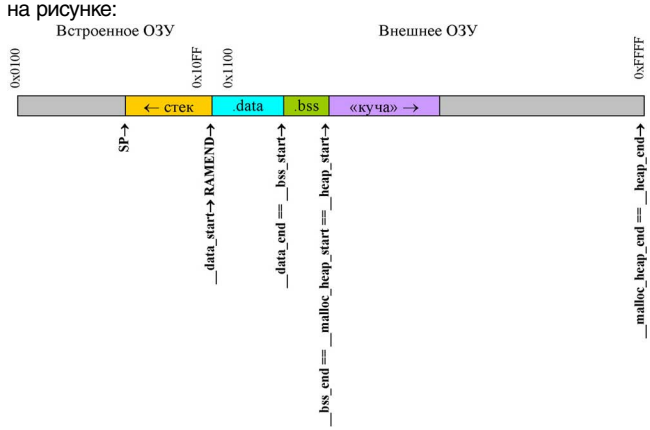
Представляется очевидным, что при наличии внешней памяти следует разместить «кучу» в ней – это однозначно позволит избежать проблем со стеком, который всегда должен быть во встроенном ОЗУ. Такой перенос не должен зависеть от места размещения .data и .bss. Разумеется, обращение к внешнему ОЗУ более медленное, нежели ко внутреннему, поэтому окончательное решение о целесообразности такого переноса следует принимать осознанно.

Кроме того, существуют ранее упомянутые (см. stdlib.h – Стандартные возможности) «настроечные» глобальные переменные, которые позволяют отрегулировать поведение функции malloc(). Инициализация этих переменных должна осуществляться до первого обращения к функции malloc(), причем следует помнить, что реализация некоторых других функций использует обращение к malloc() (см. stdio.h – Стандартный ввод-вывод), т.е. надо быть уверенным, что инициализация *действительно* происходит раньше *первого* обращения.

Переменные **__malloc_heap_start** и **__malloc_heap_end** могут использоваться для ограничения области действия функции malloc(). По умолчанию эти переменные инициализируются компоновщиком значениями так, что **__malloc_heap_start** указывает сразу на первую ячейку после .bss, а **__malloc_heap_end** устанавливается в 0, символизируя, что malloc() может выделять память вплоть до указателя стека. Для перемещения динамически распределяемой области во внешнюю память переменная **__malloc_heap_end** должна быть соответственно откорректирована. Это может быть сделано либо во время исполнения программы путем записи в переменную, либо на этапе компоновки с использованием символа **__heap_end**. Вот, например, как можно переместить целиком .data, .bss и динамическую область во внешнее ОЗУ при помощи директивы компилятора (следует помнить об особенности адресации секций при компоновке, которая была рассмотрена в предыдущем разделе):

-Wl,-Tdata=0x801100,-defsym=__heap_end=0x80ffff

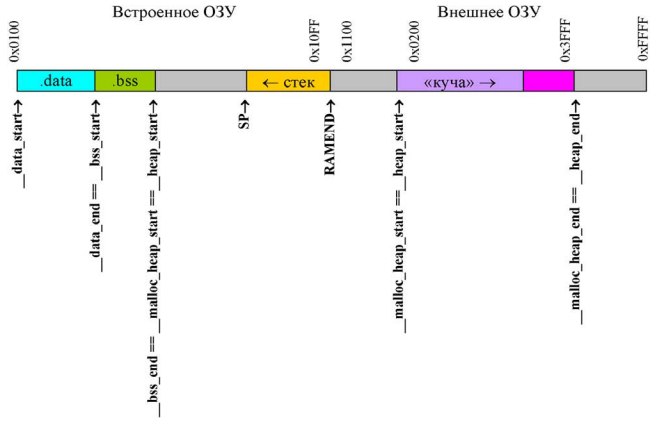
В результате распределение памяти будет таким, как показано на рисунке:



То есть во встроенном ОЗУ останется только область стека, а все остальные области окажутся во внешнем ОЗУ. Динамически распределяемая область будет простираться вплоть до адреса 0xFFFF. Другой вариант – когда требуется разместить во внешнем ОЗУ только динамическую область, оставив во встроенном стек и секции .data и .bss. В этом случае можно использовать что-то подобное такой директиве компилятора:

-Wl,-defsym=__heap_start=0x802000,-defsym=__heap_end=0x803fff

Распределение памяти в этом случае будет следующим (обратите внимание, что тут для примера специально сделано несплошное распределение областей, т.е. имеются «дыры», недоступные для динамического распределения памяти):



[39] В документации к avr-libc указана именно эта цифра – минимум 128 байт ОЗУ, однако это не мешает компилятору генерировать корректный код для микроконтроллеров с ОЗУ в 64 байта (например, для **attiny13**). Разумеется, программист должен прилагать определенные усилия для «экономии» ОЗУ в этом случае. Более того, с некоторыми усилиями WinAVR способен генерировать код для микроконтроллеров вообще без ОЗУ, например, для **attiny15**! Но это уже из области «виртуозного» программирования.

[40] Все рисунки этого раздела демонстрируют распределение адресов памяти, характерное для микроконтроллера Atmega128.

Подобное распределение с «дырами» может быть необходимо, например, в случае, когда внешнее ОЗУ физически не присутствует в определенных областях адресного пространства – ситуация вполне обычная для микроконтроллерных систем.

Примечание: разумеется, работа с внешним ОЗУ требует определенных действий по настройке аппаратной поддержки этой возможности в AVR, которые не рассматриваются в данном руководстве (изложены в соответствующих разделах документации к микроконтроллерам, поддерживающих работу с внешним ОЗУ).

Если переменная **__malloc_heap_end** содержит ноль, то функция malloc() будет пытаться выделить память так, чтобы не пересечь значение указателя стека, причем будет дополнительно зарезервировано **__malloc_margin** байтов. Т.е. между последним выделенным байтом функцией malloc() и текущим указателем стека всегда гарантируется пространство не менее **__malloc_margin** байтов. В этом случае программист должен быть уверен, что различные вызовы функций и(или) прерываний не приведут к изменению указателя стека более, чем на эту величину, иначе не исключена возможность пересечения с динамически распределяемой областью памяти. По умолчанию **__malloc_margin** содержит значение 32.

Для динамического выделения памяти используется подход, близкий к тому, который был реализован в MS DOS: связный список свободных областей. Для организации списка используется по 4 дополнительных байта, которые не являются доступными программе пользователя, но предшествуют адресу выделенной по запросу области. В этих байтах хранится признак занятости или свободности блока, ссылки на следующий и(или) предыдущий блок. Таким образом, при запросе 4 байтов фактически выделяется 8, но первые 4 используются менеджером динамической памяти, а на вторые 4 возвращается указатель. Если программа пользователя изменит содержимое первых 4-х байтов указанного промежутка, то связный список будет разрушен и работа менеджера динамической памяти станет непредсказуема. Другое следствие данного подхода: если осуществляется многократный запрос небольших участков памяти, это может быстро привести к нехватке свободного ОЗУ, за счет того, что к каждому запрошенному участку фактически прибавляется 4 дополнительных байта.

При каждом обращении к malloc() менеджер динамической памяти осуществляет просмотр списка свободных участков памяти, стремясь найти подходящий (с размером равным или большим запрошенному). Если элемент найден и его размер равен запрошенному – просто возвращается указатель на него. Если же размер найденного элемента больше требуемого, происходит разбиение его на две части: первая есть запрошенный блок, а вторая добавляется в список свободных. Если подходящего по размеру элемента не найдено, происходит попытка увеличить размер кучи, т.е. в зависимости от значения **__malloc_heap_end** происходит либо проверка на пересечение со стеком, либо на границу «кучи», после чего размер последнего элемента в списке изменяется. Если изменение (увеличение) «кучи» невозможно, malloc() возвращает NULL.

При вызове функции free() участок памяти просто добавляется в список свободных, при этом менеджер памяти всегда стремится объединить два подряд идущих свободных участка памяти в один, уменьшая тем самым количество элементов в списке и делая возможным выделение большего участка (т.е. происходит оптимизация «кучи», в определенном смысле «дефрагментация»). Подобная оптимизация происходит и при попытке изменить размер ранее выделенной области, т.е. при вызове функции realloc().

Очевидно, что все эти операции достаточно долгие, т.к. связаны с дополнительными затратами на просмотр корректировку списка элементов памяти.

Данные в сегменте кода

Любая программа на Си содержит большое количество констант, которые, в частности, используются для инициализации переменных. В силу особенностей работы компилятора получается так, что константы оказываются частью ассемблерных инструкций. Это кажется нормальным, если константа и соответствующая ей переменная имеет тип char или int, но для более «длинных» переменных это создает проблему: получается, что одни и те же данные размещаются как в памяти программ, так и в ОЗУ (куда они «перекочевывают» на этапе инициализации программы). Особенно актуальна эта проблема для определений типа этого:

```
const char str[] = 'Пример 1';
const int arr[] = {1,2,3,4,5,6,7,8,9};
```

В этих случаях будет сгенерирован автоматически исполняющийся код, который будет заносить символы строки в соответствующие ячейки ОЗУ, а так же числа 1,2,3 и т.д. в другие ячейки ОЗУ. Очевидно, это приведет к совершенно нерациональному использованию ОЗУ, которого обычно достаточно немного в микроконтроллерах.

Выход из этой ситуации – хранение констант подобного рода в памяти программ, объем которой, как правило, существенно больше ОЗУ. Однако есть сложность, накладываемая гарвардской архитектурой AVR – ОЗУ и память программ размещаются в разных непересекающихся адресных пространствах. Дело в том, что язык Си ориентирован на архитектуру фон-Неймана, т.е. разработан для случая, когда и данные и коды программы находятся в едином адресном пространстве. Поэтому требуются особые подходы, чтобы обеспечить хранение констант в памяти программ.

Во многих компиляторах для этого используются отступления от стандарта в виде новых ключевых слов, опций компилятора и т.п. Комплект WinAVR достигает этих же целей иначе.

В AVR-GCC имеется специальное ключевое слово **__attribute__**, которое позволяет определить различные дополнительные требования или условия при объявлении переменных, констант, функций и т.п. Это ключевое слово сопровождается двойными скобками, внутри которых указываются соответствующие атрибуты. В частности, для указания компоновщику и компилятору того, что описываемая переменная или константа должна размещаться в памяти программ, используется атрибут progmem.

В файле pgmspace.h (см. avr/pgmspace.h – Поддержка обращения к сегменту кода AVR) определен макрос PROGMEM, который упрощает процедуру указания этого атрибута, а так же определен ряд макросов для обращения к описанным константам. Возможно, это покажется не самым удобным способом, однако, введение нестандартного поведения в GCC – слишком сложная процедура…

Многие считают, что ключевого слова const достаточно для того, чтобы объявить константу в памяти программ, однако, это далеко не так. Это противоречит самому смыслу ключевого слова const, которое лишь сообщает компилятору, что данные не должны модифицироваться. Например, const часто используется в определении списка параметров функции, обозначая, что внутри функции содержимое этого параметра не должно модифицироваться.

Теперь о том, как же правильно осуществить хранение и обращение к константам в памяти программ.

Предположим, в программе имеется следующее объявление:

```
unsigned char mydata[11][10] =
{
        {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09},
        {0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13},
        {0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D},
        {0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27},
        {0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x30,0x31},
        {0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B},
        {0x3C,0x3D,0x3E,0x3F,0x40,0x41,0x42,0x43,0x44,0x45},
        {0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F},
        {0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59},
        {0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,0x60,0x61,0x62,0x63},
        {0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D}
};
```

Это какая-то таблица, которая будет использоваться только в качестве источника каких-то масштабных коэффициентов, т.е. содержимое массива будет неизменно. Работа с массивом будет осуществляться, например, так:

```
some = mydata[i][j];
```

Можно поместить этот массив в память программ:

```
#include <avr/pgmspace.h>

unsigned char mydata[11][10] PROGMEM =
{
        {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09},
        {0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13},
        {0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D},
        {0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27},
        {0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x30,0x31},
        {0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B},
        {0x3C,0x3D,0x3E,0x3F,0x40,0x41,0x42,0x43,0x44,0x45},
        {0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F},
        {0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59},
        {0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,0x60,0x61,0x62,0x63},
        {0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D}
};
```

После компиляции можно убедиться, что содержимое массива действительно размещено в памяти программ. Однако ранее описанная конструкция **some = mydata[i][j]** теперь не будет работать правильно! Компилятор по-прежнему ведет поиск значений в ОЗУ. Чтобы получить верное значение из теперешнего массива **mydata**, необходимо использовать макрос pgm_read_byte(), передав ему в качестве параметра адрес нужного элемента массива:

```
some = pgm_read_byte(&(mydata[i][j]));
```

Если данные имеют другую размерность – следует использовать соответствующие макросы. Теперь рассмотрим случай с хранением строк в памяти программ.

```
char* string_table[] PROGMEM =
{
    "String 1",
    "String 2",
    "String 3",
    "String 4",
    "String 5"
};
```

Будет ли верным такое определение? И да, и нет – смотря что требовалось. Если ожидалось, что в память программ попадут строки «String 1», «String 2», и т.п. – это ошибочное определение. А вот если ожидалось, что массив с адресами начала строк должен быть размещен в памяти программ – то да, это действительно верно. Но, скорее всего, ожидалось первое.

Дело в том, что атрибут в GCC действует лишь на определение переменной, но никак не на ее значение, т.е. к переменной **string_table** атрибут **progmem** будет применен, а собственно к строкам – нет. Чтобы действительно разместить строки в памяти программ, надо назначить соответствующий атрибут каждой строке отдельно:

```
char string_1[] PROGMEM = "String 1";
char string_2[] PROGMEM = "String 2";
char string_3[] PROGMEM = "String 3";
char string_4[] PROGMEM = "String 4";
char string_5[] PROGMEM = "String 5";
```

а затем воспользоваться макросом для определения массива строк в памяти программ:

```
PGM_P string_table[] PROGMEM =
{
    string_1,
    string_2,
    string_3,
    string_4,
    string_5
};
```

В результате будет получено желаемое: в памяти программ будет определен массив указателей на строки, которые так же находятся в памяти программ. Использование такого массива, как и строк в программной памяти вообще, может происходить различным образом: можно обращаться к строкам при помощи макроса pgm_read_byte() байт за байтом, а можно воспользоваться специальными функциями, ориентированными на работу с такими строками (см. avr/pgmspace.h – Поддержка обращения к сегменту кода AVR). Вот как, например, может быть реализован вывод вышеописанного массива строк:

```
void foo(void){
    char buffer[10];

    for (unsigned char i = 0; i < 5; i++)    {
        strcpy_P(buffer, (PGM_P)pgm_read_word(&(string_table[i])));

        // тут выводим строку
    }
    return;
}
```

В данном примере происходит вызов функции strcpy_P(), которая копирует в ОЗУ строку из памяти программ, адрес которой считывается из массива string_table, находящегося так же в памяти программ.

Примечание. Использование макросов для обращения к памяти программ естественно вызывает генерацию компилятором некоторого дополнительного кода, т.е. вызывает прирост объема кода программы и времени ее исполнения. Эти накладные расходы, как правило, невелики по сравнению с экономией ОЗУ, однако программист должен помнить об этом, ибо при необходимости может учесть это в программе для получения более оптимального кода, например, выделив все обращения к памяти программ в одну функцию. Всегда полезно смотреть на листинг программы.

EEPROM

Работа с EEPROM (см. avr/eeprom.h – Поддержка EEPROM AVR), входящим в состав практически всех микроконтроллеров AVR, осуществляется по принципам, сходным с обращением к памяти программ (см. Данные в сегменте кода): то есть просто объявляется проинициализированная при необходимости переменная с соответствующим атрибутом (точнее, используется макрос EEMEM для этого). Если переменная проинициализирована, то компилятор поместит соответствующие данные в отдельном файле, который затем можно использовать для программирования EEPROM микроконтроллера.

```
#include <avr/eeprom.h>

unsigned char mydata[11][10] EEMEM =
{
        {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09},
        {0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13},
        {0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D},
        {0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27},
        {0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x30,0x31},
        {0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B},
        {0x3C,0x3D,0x3E,0x3F,0x40,0x41,0x42,0x43,0x44,0x45},
        {0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F},
        {0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59},
        {0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,0x60,0x61,0x62,0x63},
        {0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D}
};

char foo(char i; char j){
return eeprom_read_byte(&(mydata[i][j]));
}
```

Для обращения к соответствующим переменным так же используются макросы-функции. К сожалению, в состав AVR-LIBC не включены функции для работы со строками, размещенными в EEPROM, однако, программист может реализовать их самостоятельно. В отличие от констант в памяти программ, переменные в EEPROM – это действительно переменные, т.е. их значение может быть изменено в процессе работы программы. Для изменения значений таких переменных используются соответствующие макросы. Запись в EEPROM – относительно долгий процесс, о чем должен помнить программист.

Продолжение в №4/2011