# Книга по работе с WinAVR и AVR Studio

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

## ПРЕДИСЛОВИЕ

Последние годы заметен среди радиолюбителей резкий всплеск интереса к конструкциям на микроконтроллерах (МК). Обусловлено это их чрезвычайной универсальностью и гибкостью при более чем демократичных ценах. Наибольшей популярностью продолжают пользоваться 8-разрядные МК, среди которых выделяются модели семейства AVR фирмы Atmel.

Но микроконтроллер без программы – это мертвый компонент, его функциональность даже меньше, чем у обычного инвертора. Процесс разработки программного обеспечения для микроконтроллера – это наиболее сложная часть конструирования, именно это вызывает наибольшие сложности у любителей. Не обладая достаточными навыками в программировании, они вынуждены заниматься лишь копированием чужих разработок. Совпадение целей и интересов у автора-разработчика и человека, решившего повторить конструкцию, происходит редко, и это порождает множество вопросов, криков о помощи на Интернет-форумах и, порой, отталкивает радиолюбителя от микроконтроллеров – дескать, легче сделать что-то на десятке микросхем жесткой логики, чем уговорить кого-то подкорректировать программу.

Но зачем кого-то уговаривать? Перефразируя известное детское стихотворение: не надо ждать, не надо звать, а надо взять и… написать! Т.е. речь идет о собственноручном написании программ для микроконтроллеров. Это не так сложно, как кажется, во всяком случае, под силу каждому технически грамотному человеку, овладевшему компьютером и электроникой, пусть даже не цифровой.

Как известно, программы разрабатываются на языках программирования, среди которых наибольшей популярностью пользуется Си. Благодаря основам, заложенным в этот язык, разработка программ для микроконтроллеров мало отличается от разработки программ для обычных компьютеров. Поэтому проще всего переключиться на микроконтроллеры смогут те, кто уже умеет хоть немного программировать для персонального компьютера. А все остальные желающие могут прочесть специальные книги, посвященные как самому языку Си, так и процессу алгоритмизации. Литературы на этот счет достаточно, равно как и литературы об аппаратном строении микроконтроллеров. А вот книг, которые целенаправленно рассматривали бы аспекты «сопряжения» языка программирования и микроконтроллера, явно недостаточно. Восполнить этот пробел хоть в какой-то мере – вот главная цель и задача этой книги.

И при восполнении упомянутого пробела сделан упор на бесплатное программное обеспечение. Дело в том, что ценность имеющихся книг, рассказывающих о программных пакетах стоимостью в сотни и даже тысячи долларов, очень сомнительна. Любителей, способных выложить такие суммы, найдется в нашей стране исчезающе мало, и даже не всякие фирмы рискнут пойти на такие траты. Выходит, что подобные книги косвенно стимулируют пиратство. В то же время есть альтернативный, абсолютно бесплатный софт, по качеству вполне соперничающий с коммерческими пакетами. Его применение абсолютно законно, и более того – часто дает лучшие результаты, нежели иные коммерческие. Но бесплатные пакеты порой грешат недостаточно качественной документацией, а то и полным ее отсутствием, тем более на русском языке. Ликвидация этого недостатка – вторая задача данной книги.

**Кому может быть полезна эта книга?** Прежде всего, тем, кто занимался «компьютерным» программированием и испытывает желание или необходимость приступить к разработке программ для микроконтроллеров, например, в связи со сменой работы. Книга будет полезна и студентам, изучающим микроконтроллеры и радиолюбителям, желающим освоить новую для себя стезю. Полезное для себя наверняка извлекут и профессионалы, желающие перейти на бесплатное программное обеспечение. Несомненно, полезной книга будет для всех, кто владеет английским языком в недостаточной мере, чтобы изучать фирменную документацию.

**Что можно найти в этой книге?** Прежде всего, практические рекомендации по установке и использованию пакета разработки программ для микроконтроллеров AVR, распространяемому бесплатно – WinAVR, а так же применению его в интеграции с бесплатной IDE фирмы Atmel – AVR Studio. Краткое описание основ языка Си, которое не может быть полноценным учебником, но должно послужить основой для первых шагов в программировании. Книга так же содержит подробный справочник по функциям, входящим в библиотеку AVR-LIBC, являющуюся частью комплекта WinAVR, руководство по разработке модулей на ассемблере для Си-программ. Достаточно подробно рассмотрен необходимый минимум команд компилятора GCC, являющегося «сердцем» WinAVR, а так же описаны некоторые утилиты сторонних разработчиков. В книге имеется ряд практических примеров программ и программных «заготовок» для решения задач, являющихся типовыми: обмен с компьютером по RS-232, организация динамической индикации, работа с ЖКИ и т.п. В книгу включен FAQ – ответы на часто задаваемые вопросы, составленный на основе Интернет-форумов и других источников.

**Чего вы не найдете в этой книге?** Главное: вы не найдете сведений об архитектуре и периферии микроконтроллеров AVR – об этом написано множество книг и повторяться нет никакого смысла. Так же в книге не рассмотрен ряд утилит и средств, входящих в комплект WinAVR – в основном, это утилиты для программирования и отладки, а так же вспомогательные утилиты (binutils), работающие в консольном режиме. Сделано это вполне осознанно: в среде Windows более привычным и удобным является графический интерфейс, который для работы с WinAVR обеспечивает AVR Studio, делая тем самым ненужными консольные средства.

**Что потребуется для того, чтобы воспользоваться информацией из этой книги?** Для работы будет необходим компьютер, желательно, подключенный к Интернет. Потребуется, скорее всего, кое-что из аппаратуры: рекомендуется один из комплектов разработчика, предлагаемых фирмой Atmel или другими фирмами. В крайнем случае, можно ограничиться наличием одного из микроконтроллеров AVR и самодельным программатором. Все необходимое для работы программное обеспечение (в том числе инсталляционные пакеты AVR Studio и WinAVR), рассматриваемые в книге примеры программ, а так же большое количество документации и свободно распространяемых любительских библиотек и разработок имеется на сайте [1].

## РАЗРАБОТКА ПРОГРАММ

Разработка программного обеспечения для микроконтроллеров AVR мало отличается от разработки любых иных программ, разве что от программиста требуется чуть более глубокие знания электроники – хотя бы на уровне понимания действия логических элементов и триггеров. Разумеется, чем лучше программист владеет электроникой, тем более качественные программы он сможет создать. Тесная связь «софта и железа» по сути требует, чтобы программист был электронщиком или же электронщик был программистом. В настоящее время обе ситуации имеют место.

Существует три более-менее устоявшихся подхода к разработке микроконтроллерных устройств:

- от схемы к программе
- от программы к схеме
- смешанный.

Деление, конечно, условное. В первом случае электронщик разрабатывает схему, представляя микроконтроллер неким «черным ящиком», т.е. устройством, которое «как-то само» выполняет определенные действия, формируя выходные сигналы в зависимости от входных. После того, как схема готова, составляется задание программисту, в котором указывается, на каких выводах контроллер какие сигналы должен получать или формировать. Порой на этом этапе требуется корректировка схемы, т.к. далеко не любой вывод контроллера способен выполнить нужную функцию. По заданию программист пишет программу, после чего следует этап отладки «в железе», т.е. в реальной схеме. После этого этапа часто следуют корректировки как схемы, так и программы. В общем, процесс итеративно повторяется до достижения требуемого результата.

При втором подходе программист получает задание на разработку программы первым. В процессе работы он сам выбирает, какой вывод МК будет выполнять то или иное действие (для чего, несомненно, он должен знать архитектуру выбранного контроллера). Когда, как минимум, сделано «распределение выводов», задание на разработку схемы получает и электронщик. Теперь уже он должен рисовать схему с учетом отсутствия свободы выбора выводов микросхемы-микроконтроллера. После того, как программа и схема готовы, следуют этапы отладки, которые не отличаются от предыдущего варианта.

Наконец, при третьем подходе программист и электронщик – это одно и то же лицо, которое по ходу составления схемы пишет и программу. В этом случае этот специалист, зная все нюансы электроники и программирования, «по ходу пьесы» подстраивает одно под другое, т.е. добивается наиболее оптимального взаимодействия всех частей. Ведь не секрет, что желания программиста и электронщика часто взаимопротивоположны, и поиск компромиссов в первых двух рассмотренных вариантах чреват лишними препираниями, в то время как третий подход все трения исключает.

Так как настоящая книга посвящена программным средствам, дальнейшее изложение будет вестись, как будто имеет место второй вариант, т.е. от программы к схеме – так, не обращая излишнего внимания на схемотехнику, проще разобраться в процессе программирования. При этом неизбежные обращения взгляда на схемотехнику будут происходить в предположении, что читатель обладает необходимыми знаниями и навыками в области электроники, т.е. он – электронщик, желающий стать программистом.

### *Основы Си*

Эта глава посвящается введению в язык Си. Это не учебник, скорее – это упрощенный краткий справочник с пояснениями и примерами. К сожалению, последовательность изложения может показаться нелогичной: это объясняется тем, что невозможно совместить краткость и простоту с логичностью и последовательностью при рассмотрении такой сложной системы, как язык программирования. Поэтому часть упоминаемых терминов получает объяснение в последующем тексте, иногда даже в другом разделе.

#### Общие сведения

Назначение языка программирования – предоставить программисту средства для изложения алгоритма решения какой-либо задачи в форме, воспринимаемой компилятором (специальной программой, служащей для «перевода» с одного языка на другой, понятный конкретной аппаратной платформе, т.е. процессору или микроконтроллеру).

Основу любого языка составляет **алфавит**, т.е. определенный набор допустимых символов, из которых затем составляются *лексемы*, т.е. элементы языка, уже означающие конкретные понятия.

Любой язык программирования – это «письменный» язык, т.е. он существует в виде символов текста. Запись средствами языка алгоритма в файле называется исходным текстом программы (или просто программой).

Алфавит языка Си составляют:

1. Буквы латинского алфавита
2. Цифры от 0 до 9
3. Специальные знаки:

```
" { } , | [ ] ( ) + - / % \ ; ' : ? < = > _ ! & # ~ ^ . *
```

Пробел не является языковым элементом, он служит лишь для разделения отдельных лексем.

Из элементов алфавита составляются лексемы:

- Идентификаторы, называемые иногда символами
- Зарезервированные (ключевые) служебные слова
- Знаки операций
- Разделители

***Идентификатор*** – это последовательность из символов латинского алфавита, десятичных цифр и символов подчеркивания, начинающаяся не с цифры. Прописные и строчные символы различаются, т.е. **Run** и **RUN** – разные идентификаторы. Длина идентификатора принципиально не ограничена, но анализируются только первые 32 символа[1], т.е. два идентификатора с одинаковыми первыми 32-символами будут считаться одинаковыми, даже если в 33-ем символе они различаются.

***Разделитель*** – это символ или несколько слитно написанных символов, которые служат для отделения одной лексемы от другой. Об одном разделителе – *пробеле* – уже было сказано, однако более важным является точка с запятой. Так же важными разделителями являются скобки[2] (обязательно в паре – открывающая и закрывающая). Разделителем является и пустая строка, т.е. символ перевода строки, а так же символ табуляции.

Существует особый вид разделителя – **комментарий**. В Си определено два вида комментария: *многострочный* и *однострочный*.

Многострочный комментарий начинается с пары символов «**/***» и завершается парой «***/**». Все символы, находящиеся между этими парами символов, игнорируются, т.е. не воспринимаются компилятором.

Однострочный комментарий начинается с пары символов «**//**» и продолжается до конца текущей строки, т.е. все символы, следующие за этой парой до конца строки, игнорируются компилятором.

Программа на языке Си представляет собой последовательность лексем и разделителей, записанных в текстовом файле. Если отдельную законченную мысль человек при письме завершает точкой, то в языке Си каждая законченная мысль (т.е. последовательность лексем, имеющая самостоятельное смысловое значение) завершается особым разделителем – точкой с запятой. Хотя нет никаких ограничений на способ записи программы, все же следует по мере возможности выделять каждую осмысленную запись программы, начиная ее с новой строки. Кроме этого есть и другие рекомендации – см. «О стиле программирования».

#### Служебные слова

Ключевое или служебное слово – это идентификатор, зарезервированный для особого использования. В Си стандартно определены следующие служебные слова[3]:

| | | | |
|---|---|---|---|
| **break** | **enum** | **long** | **struct** |
| **case** | **extern** | **public** | **switch** |
| **char** | **float** | **register** | **typedef** |
| **const** | **for** | **return** | **union** |
| **continue** | **goto** | **short** | **unsigned** |
| **do** | **if** | **signed** | **void** |
| **double** | **inline** | **sizeof** | **volatile** |
| **else** | **int** | **static** | **while** |

В сущности, этих служебных слов достаточно, чтобы реализовать любую программу. Часть из этих слов являются *операторами* (см. далее), часть используется в составе других конструкций, часть соответствует обозначению стандартных *типов*. Далее все служебные слова будут рассмотрены в соответствующем контексте по назначению.

#### Типы данных

Любой алгоритм, так или иначе, сводится к неким действиям над данными, будь то числа, буквы, звуки, сигналы и т.п. Так как язык программирования служит для реализации алгоритма, он имеет все необходимые средства для манипулирования данными. Для разделения их по видам, введено понятие *тип* данных. Так как тип не может существовать без данных, а данными в языке Си может быть очень много совершенно разных объектов, далее будем использовать термин *объект* для обозначения *абстрактных* данных.

***Тип*** – это идентификатор, обозначающий принадлежность объекта к некой категории, однозначно определяющей содержимое и набор действий, которые можно выполнить над ним. Например, все числа можно разделить на ряд категорий – целые, действительные, комплексные, отрицательные и т.п.

Так как работа процессора и микроконтроллера есть процесс цифровой обработки данных, поэтому так или иначе все разнообразие данных сводится к числам. Даже буквы есть не что иное, как числа,

[1] Значение может отличаться в зависимости от стандарта, которому соответствует компилятор.

[2] Разделителями считаются любые скобки – круглые, фигурные и квадратные, однако не всегда и не все из них могут использоваться произвольно – существует ряд ограничений, которые будут упомянуты по мере изложения основ языка.

[3] Список служебных слов может несколько отличаться в зависимости от конкретного компилятора (как правило в сторону расширения).

сопоставленные им (грубо говоря, это номер буквы по порядку в алфавите). По этим причинам в языке Си основным видом данных являются числа, которые для удобства разделены на ряд стандартных типов[4]:

**char** – символ, то есть число, занимающее один байт (т.е. 8-битное число)

**int** – целое число, занимающее 2 байта (т.е. 16-битное число)

**long** – длинное целое число, занимающее 4 байта (т.е. 32-битное число)

**long long** – двойное длинное целое число, занимающее 8 байт (т.е. 64-битное число)

**float** – действительное число, т.е. число с плавающей точкой

**double** – действительное число удвоенной точности

При помощи дополнительных служебных слов определяются разновидности типов целых чисел:

**signed** – обозначает, что целое число имеет знак (т.е. может принимать и отрицательные значения)

**unsigned** – обозначает, что целое число не имеет знака, т.е. может быть только нулем или более

**short** – обозначает, что фактический размер целого числа вдвое меньше

Таким образом, получаются следующие типы:

**unsigned char** – число от 0 до 255

**signed char** – число от -127 до 127

**unsigned int** – число от 0 до 65535

**signed int** – число от -32767 до 32764

**unsigned long** – число от 0 до 4294967295

**signed long** – число от -2147483648 до 2147483648

и т.д.

Следует отметить, что слово **signed,** как правило, нет смысла использовать, так как *по умолчанию все типы целых чисел имеют знак*.

**Примечание**: тип **char** стоит особняком – многие версии стандарта Си допускают, что этот тип *по умолчанию не имеет знака*, что является причиной многих ошибок. Обычно имеется возможность управлять компилятором, переключая тип **char** со знакового на беззнаковое по умолчанию. Но, чтобы никогда не зависеть от особенностей компилятора, рекомендуется всегда явно указывать **signed** или **unsigned** при использовании типа **char**.

Так же ключевое слово **short** используется достаточно редко[5] в программировании для микроконтроллеров.

Программист имеет возможность определять новые типы на основе уже определенных, для этого служит ключевое слово **typedef**, например:

**typedef unsigned char byte** – определение нового типа **byte** для обозначения однобайтного числа без знака. То есть сначала следует ключевое слово **typedef**, затем указывается уже определенный тип, а завершает все идентификатор нового типа.

После такого определения программист может использовать **byte** наравне с другими типами, в том числе и для нового определения типа.

Все рассмотренные типы считаются простыми типами. Кроме них, имеются и сложные, например, *массивы* (см. далее), *структуры* и *объединения* (см. далее) – они рассмотрены отдельно. Однако все сказанное относится и к ним, т.е. любой сложный тип может использоваться для определения нового типа пользователя.

**Константы**

***Константа*** – это лексема языка, представляющая какое-либо фиксированное числовое или иное значение.

Константы делятся на несколько групп: *целочисленные*, *действительные*, *символьные* и *строковые*. В зависимости от значения константы, компилятор относит ее к одному из известных типов данных, выделяя для нее соответствующее количество байтов памяти.

**Целочисленные константы** представляют собой запись чисел в одной из допустимых систем счисления: десятичной, двоичной[6], восьмеричной и шестнадцатеричной. Система счисления определяется компилятором следующим образом: если лексема, определенная как константа, начинается с символов «0b» – принимается, что это запись числа в двоичной системе; если лексема начинается с символов «0x» – принимается шестнадцатеричное представление числа. Восьмеричная система подразумевает, что число всегда начинается с *нуля*. Если же ни один из рассмотренных признаков не обнаружен – принимается десятичная система.

Вот пример записи константы (числа 307) в разных системах счисления:

**307** – десятичная запись

**0b100110011** – двоичная запись

**0463** – восьмеричная запись

**0x133** – шестнадцатеричная запись.

По значению константы компилятор определяет наименьший подходящий для нее тип данных. Однако часто необходимо задавать константы так, чтобы они заранее соответствовали конкретному типу, например, нужно определить константу **long** равную **2**. Если просто написать **2**, то компилятор посчитает, что для константы достаточно типа **char**, и будет его и использовать (далее будет показано, к чему это может привести). Чтобы принудительно указать размер константы, используются специальные ***суффиксы***, приписываемые к константе справа:

**L** – для обозначения **long** (можно и прописную букву «**l**» использовать)

**U** – для обозначения **unsigned** (можно и прописную букву «**u**» использовать).

Суффиксы могут использоваться по отдельности или вместе в любых комбинациях. Таким образом, ранее упомянутая константа должна быть записана в виде **2L** или **2ul**.

Константы **действительного типа** определяются компилятором по наличию точки в записи лексемы или по наличию символа «**e**» (или «**E**»), отделяющего мантиссу[7] от показателя степени числа 10. Вот пример констант этого типа:

**66.2 .1 0. 2.14e-7**

Количество байтов, выделенное компилятором для хранения константы, в этом случае определяется формой представления чисел с плавающей точкой, т.е. может быть разным для разных компиляторов. При помощи служебного слова ***sizeof*** можно получить константу, равную количеству байтов для хранения константы:

**sizeof**(**3**) будет равно 1 (1 байт);

**sizeof**(**3L**) будет равно 4 (4 байта),

и т.д.

**Символьная константа** – это разновидность целочисленной константы, служащей для определения кода алфавитно-цифрового символа. Для обозначения такой константы используется запись символа, окруженного апострофами:

**'F' '5' '.'**

Очевидно, что далеко не любой символ можно ввести таким образом. Снять проблему позволяет особый символ «обратная черта» (или **back-slash**): этот символ (называется **escape**-символ) позволяет ввести *неотображаемые* символы, символы «апостроф», «кавычки», «вопрос» и «обратная черта», а так же любой другой символ, указав его код.

Определение символа при помощи **escape** называется escape-последовательностью[8]:

| *Escape-последовательность* | *Код символа* | *Наименование символа* |
|---|---|---|
| **\a** | 0x07 | Звуковой сигнал |
| **\b** | 0x08 | «Забой» (возврат на символ) |
| **\f** | 0x0c | Перевод страницы |
| **\n** | 0x0a | Перевод строки |
| **\r** | 0x0d | Возврат каретки |
| **\t** | 0x09 | Горизонтальная табуляция |
| **\v** | 0x0b | Вертикальная табуляция |
| **** | 0x5c | Обратная черта |
| **\'** | 0x27 | Апостроф |
| **\"** | 0x22 | Кавычки |
| **\?** | 0x3f | Знак вопроса |
| **\000** | 000 | Любой восьмеричный код символа |
| **\0x00** | 0x00 | Любой шестнадцатеричный код символа |

[4] Приведены типы, являющиеся стандартными в версии Си для микроконтроллеров AVR

[5] Ключевое слово short имеет важную роль в системах, для которых тип int не является 16-битным числом (например, для PC-компьютеров). В этом случае тип char соответственно может быть 16-битным, и тогда для получения однобайтного типа необходимо использование short char.

[6] Двоичные константы определены для GNU-расширений Си.

[7] Подразумевается, что читатель знаком с показательной формой записи действительных чисел.

[8] Escape-последовательности ведут свое начало с тех времен, когда работа велась через консольные текстовые терминалы – эти последовательности управляли положением «курсора» на терминале или печатающем устройстве.

При вводе символов по их кодам в **escape**-последовательности следует помнить, что код не может быть больше 255 (в десятичном представлении).

Наконец, **строковая константа** – это последовательность символов, заключенная в кавычки. Среди символов строковой константы могут быть и **escape**-последовательности. Если введены подряд несколько строковых констант, разделенных лишь пробелами, табуляциями, переводами строки или комментариями – они объединяются в одну строковую константу. Длинные строки можно переносить со строки на строку при помощи символа переноса – той же самой «обратной черты».

Есть несколько особенностей слияния и переноса строк, связанных с обработкой пробелов:

1. Если происходит слияние строк – результирующая строка будет содержать последовательно все символы обеих строк (это очевидно).

2. Если строка переносится при помощи символа переноса, то в результирующую строку будет включены все пробелы от последнего символа до символа переноса и все пробелы от начала следующей строки до константы.

Вот примеры, поясняющие сказанное:

```
"простая строковая константа"
"слияние строк "              "123"
"перенос строки               \
123"
"еще один вариант\
                              переноса строки"
"вариант\
переноса"
```

Компилятор преобразует эти константы в следующие:

```
"простая строковая константа"
"слияние строк 123"
"перенос строки               123"
"еще один вариант                                переноса строки"
"вариантпереноса"
```

Важной особенностью строковых констант заключается в том, что компилятор всегда добавляет к явно введенным программистом символам еще один – символ завершения строки, код которого 0x00 (так называемый **null**-символ или *завершающий ноль*). Этот символ никак не виден в тексте программы, но обрабатывается компилятором и программой.

**Переменные**

В процессе работы любая программа не только использует константы, т.е. фиксированные значения, но и получает, изменяет и выводит данные различных типов, т.е. оперирует изменяющимися числами (как было сказано ранее, все данные, в конце концов, есть числа).

Изменяемые величины хранятся в ОЗУ, каждое отдельное число в отдельных ячейках. Для обозначения таких ячеек введена особая лексема – *переменная*.

***Переменная*** – это идентификатор, обозначающий строго определенную область памяти, содержащую данные конкретного типа. Для определения любых переменных не допускается использовать идентификаторы, совпадающие с ключевыми словами языка или уже определенными ранее (например, введенных программистом типов).

Из определения переменной следует ее неразрывная связь с одним из определенных типов, т.е. переменные могут быть типа **char**, **int**, **float** и т.д. В программе каждая переменная должна быть обязательно определена, т.е. описана. Описание переменной осуществляется по следующему шаблону[9]:

<**тип**> <**список идентификаторов**>;

Здесь **тип** – это один из ранее определенных типов, а **список идентификаторов** – это один или более идентификаторов, разделенных *запятыми*. Завершать определение должна традиционная *точка с запятой*. Запятая – это особый разделитель, используемый для разделения элементов списка, т.е. таких элементов, которые должны восприниматься компилятором как некая неделимая группа.

[9] Здесь и далее, если иное не оговорено, используется традиционный формат описания шаблонов выражений: в угловых скобках указываются обязательные элементы, в квадратных – опциональные, все прочие символы вне скобок являются обязательными.

```
unsigned long var;
int a, b, c;
char ch;
```

Пример демонстрирует определение переменной **var**, имеющей тип «беззнаковое длинное целое», трех переменных **a**, **b** и **c** типа «целое число со знаком» и одну переменную **ch** типа «символ».

Встретив определение переменных, компилятор резервирует необходимое количество ячеек памяти, которые затем будут использованы в программе по усмотрению программиста. Использование неописанных переменных в программе *не допускается*. По умолчанию каждой из описанных переменных сразу присваивается *нулевое* значение – говорят «*переменная проинициализирована значением по умолчанию*» (инициализация по умолчанию характерна не для всех переменных, о чем будет сказано далее). Далее будет показано, как можно инициализировать переменные другими значениями.

Программист должен знать, что память для хранения переменных выделяется в порядке их описания, т.е. для вышеприведенного примера в ОЗУ сразу после 4-го байта переменной **var** будут следовать 2 байта переменной **a** и т.д. Порядок байтов, составляющих переменную, определяется платформой, для которой компилируется программа. Для микроконтроллеров AVR принято, что многобайтные переменные хранятся в памяти «от младшего байта к старшему», т.е. первым размещается младший байт (наименее значащие биты числа), а затем все более старшие, вплоть до последнего (наиболее значащие биты числа).

При описании переменных могут дополнительно использоваться ключевые слова ***static***, ***register***, ***extern*** и ***volatile***. Эти слова могут располагаться как до определения типа переменной, так и после него, однако обязательно до указания идентификатора. Если переменная объявлена с использованием слова **register**, это означает, что компилятор поместит ее в один из доступных регистров микроконтроллера. Не стоит надеяться, что регистровые переменные каким-то образом помогут создать более быстродействующую программу: для большинства случаев результат будет скорее обратный, хотя в некоторых случаях положительный эффект может быть.

Ключевое слово **volatile** – очень важное. Дословно оно означает «изменяемая», что на первый взгляд излишне для переменной, само наименование которой означает изменение значения. Однако, дело тут в другом. Компилятор в процессе своей работы анализирует написанный программистом текст программы и старается убрать из него те части, которые явно не несут никакого смысла (подробнее об этом см. в главе «Об оптимизаторе программ»). В частности, компилятор может исключить все участки программы, которые обрабатывают переменные, не изменяющие (с точки зрения компилятора) значения. Разумеется, далеко не всегда это соответствует замыслам программиста. Ключевое слово **volatile**, использованное при объявлении переменной, укажет компилятору, что эта переменная все-таки изменяется, пусть даже и неизвестным компилятору способом. Можно считать, что **volatile** есть синоним слова «неприкосновенный», т.е. объявленную таким образом переменную нельзя трогать при оптимизации.

Назначение остальных ключевых слов будет рассмотрено далее.

**Массивы**

***Массив*** – это последовательно хранимый набор нескольких «пронумерованных» переменных *одинакового* типа, называемых элементами массива.

Для определения массива используется следующий шаблон:

<**тип**> <**идентификатор**>[<**размер**>];

Здесь **тип** – как и ранее, любой из определенных типов, обозначающий тип каждого элемента массива, **идентификатор** – это собственно имя переменной-массива, а **размер** (который может и отсутствовать) – это константа, ограничивающая количество элементов массива.

Обратите внимание на то, что ***квадратные скобки*** должны присутствовать всегда, независимо от того, задается ли размер или нет – это признак массива.

Из определения массива следует, что, во-первых, все элементы массива находятся в памяти друг за другом, а во-вторых, имеют одинаковый тип. Чем же это отличается от обычного последовательного определения нескольких однотипных переменных?

```
int a[5];              // массив a[] из 5-и элементов типа int
int a1, a2, a3, a4, a5; // 5 отдельных переменных типа int
```

Отличие принципиальное. Определенные во второй строке переменные независимы друг от друга, для обращения к каждой из них программист обязан использовать уникальный идентификатор. У массива же идентификатор единственный, а для обращения к его элементам используется *индекс*, т.е. номер элемента: **a[0]** – первый элемент массива, **a[4]** – пятый элемент. Обратите внимание, что нумерация элементов массива начинается с нуля. Преимущество массивов перед отдельными переменными бросается в глаза на простом примере. Если представить себе строку символов в виде массива и в виде нескольких отдельных символьных переменных, сразу становится ясно, что работа со строкой уже из десятка символов становится неразумно сложной, в то время как каждый их элементов массива-строки может быть обработан одним и тем же способом.

В качестве указателя размера массива при описании может использоваться любая целочисленная константа, а при обращении – константа или переменная соответствующего типа. В Си никак не контролируется допустимость индекса при обращении к элементу массива, т.е. обращение к **a[12]** будет воспринято компилятором, как безошибочное, однако результат в этом случае не предсказуем.

Рассмотренный пример массива – *одномерный*, т.е. его можно представить в виде «линейки» с делениями – элементами. Однако нередко требуется оперировать массивами-«плоскостями», «кубами» и более многомерными[10] «фигурами». Такая возможность предусмотрена в Си: *многомерный* массив описывается так же, как и одномерный, только количество квадратных скобок (с возможно указанным размером «измерения») соответствует числу измерений:

```
int a1[5];             // 1-мерный массив из элементов int
int a2[][];            // 2-мерный массив элементов int
                       // с неопределенными размерами
int a3[3][3][3];       // 3-мерный массив элементов int
                       // с определенными размерами
```

Подобные записи «читаются» слева направо, например, третья строка примера читается так: *массив из трех элементов, каждый из которых представляет собой массив из трех элементов, элементами которого являются массивы из трех элементов типа* ***int***. «Трехмерный массив» звучит куда проще, но это неудобное описание помогает понять, как именно следует обращаться к многомерному массиву:

```
a3[0] – это массив двумерный
a3[0][0] – это массив одномерный
a3[0][0][0]– это элемент массива (не массив!) типа int
```

Кстати, фактически многомерные массивы – это лишь форма сокращенной (и, возможно, более удобной) записи одномерного массива, размер которого определяется произведением размеров по его «измерениям». Т.е. рассмотренный массив **a3** можно представить в виде одномерного массива с размером **27** элементов. Именно как одномерный и хранится в памяти многомерный массив, причем «развертка измерений» происходит по измерениям справа налево, т.е. самым первым в памяти будет элемент **a3[0][0][0]**, затем **a3[0][0][1]**, затем **a3[0][0][2]**, затем **a3[0][1][0]** и т.д.

Как и любые переменные, массивы в момент описания инициализируются значением по умолчанию. Кроме того, как и для любой переменной, для объявления массива возможно использование ключевых слов ***extern***, ***static*** и ***volatile***.

**Структуры и объединения**

Часто требуется объединить несколько переменных или типов данных (одинаковых или разных) в некую группу. Например, обрабатывая список сотрудников, удобно объединить в одну группу фамилию, имя и отчество сотрудника, и хранить такие группы в массиве. Это, с одной стороны, позволило бы при подсчете общего количества сотрудников просто считать такие группы, но с другой стороны, в любой момент можно получить имя любого из сотрудников по его номеру. Для подобной группировки введено понятие *структуры*.

***Структура*** – это особая форма слияния нескольких переменных, называемых полями структуры, в единое целое.

Для определения структуры используется ключевое слово **struct**. Шаблон определения структуры следующий:

**struct** {<**поле 1**>; <**поле 2**>; … ; <**поле N**>;};

Здесь в фигурных скобках перечисляются *поля* структуры, которые есть ничто иное, как объявления переменных. Обратите внимание, на точки с запятой, завершающие определение каждого поля структуры. Структура – это разновидность ***типа***, поэтому определение переменной типа структура осуществляется аналогично ранее рассмотренному объявлению простых переменных:

```
struct {int a; long b; char c;} st;
```

Этот пример показывает, как объявляется переменная **st** типа структура с полями **a** (типа **int**), **b** (типа **long**) и **c** (типа **char**). Обращение к переменным-полям структуры осуществляется через так называемую *точечную нотацию*, когда поле структуры отделяется от ее идентификатора точкой: **st.a** позволит обратиться к полю, а переменной **st**, а **st.c** – к полю ее полю **c**.

Ничто не препятствует объявить массив таких структур:

```
struct {int a; long b; char c;} st[5];
```

Этот пример определяет массив **st**, состоящий из 5-и элементов, каждый из которых имеет поля **a**, **b** и **c** соответствующих типов. Точечная нотация продолжает действовать и при обращении к элементам массива структур: **st[1].a** – поле **a** второго элемента массива, **st[4].c** – поле **c** последнего элемента.

Так же ничто не препятствует определить структуру как тип, и использовать для определения новых переменных уже этот тип:

```
typedef struct { int a; long b; char c;} strc;
strc st;
strc arr_st[5];
```

Язык Си считается самым низкоуровневым языком среди высокоуровневых. Под этим следует понимать, что язык позволяет получить возможность практически так же манипулировать данными, как и ассемблер. Одной из таких «низкоуровневых» возможностей являются *объединения*.

По виду описания ***объединение*** (определяемое ключевым словом **union**) очень похоже на ***структуру***:

```
union { int a; long b; char c;} uni;
```

Однако, принципиальное отличие от структуры в том, что ***все поля объединения физически соответствуют одной и той же области памяти***! Размер области памяти, выделяемой компилятором, определяется так, чтобы поместилось поле наибольшего размера, т.е. для вышеприведенного примера будет выделено 4 байта для переменной **uni**. При этом при обращении к **uni.b** будет задействована вся эта область, при обращении к **uni.a** - только первые 2 байта этой области, а **uni.c** позволит оперировать лишь первым байтом. То есть налицо обращение к одной и той области памяти, как к переменным разного типа.

Объединение, так же как и структура, может использоваться в описании нового типа, быть элементом массива и полем структуры. Действует правило: все, что уже определено, может быть использовано. Разумеется, все равнее сказанное в отношении переменных, применимо и к структурам и объединениям.

**Литература, Интернет-ссылки**

1. http://www.arv.radioliga.com/

[10] Вспоминается шутка математиков: «представьте себе проекцию семимерного куба на пятимерную плоскость»…

**Продолжение в №2/2010**