Книга по работе с WinAVR и AVR Studio

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

Продолжение. Начало в №1-2/2010

Функции

Как было сказано, язык Си предоставляет в распоряжение программиста небольшой, но достаточно мощный инструментарий в виде набора операторов, которые позволяют достаточно просто решать определенные наиболее часто требуемые задачи. Однако программист наверняка будет нуждаться и в других решениях «типовых» для его алгоритма задач. Т.е. наверняка в программе найдутся определенные участки, выполняющие конкретные этапы алгоритма и при этом повторяющиеся с минимальными изменениями во многих местах. Для уменьшения таких повторов, а заодно для качественного улучшения реализации алгоритма введено понятие функции.

Функция – это особым образом оформленная последовательность операторов, заменяемая в тексте программы своим коротким эквивалентом, т.е. обращением к функции.

Функция представляет собой как бы мини-программу по обработке небольшого количества данных. Данные, передаваемые ей на обработку, получили название параметров функции, а результат их обработки назван значением функции. Для возврата результата функции служит особый оператор ***return***.

Различают ***описание*** (определение) функции, ***реализацию*** и ее ***использование*** (иначе говоря – вызов, т.е. обращение к функции). Как и переменная, функция должна быть определена до первого обращения к ней.

Шаблон определения функции следующий:

<**тип**> <**идентификатор**>([**список параметров**]);

Здесь **тип** – любой ранее определенный тип данных, **идентификатор** – это собственно «имя» описываемой функции, а **список параметров** – это разделенный запятыми список описаний «*входных*» переменных.

Описание функции напоминает описание переменной, за исключением обязательно присутствующих круглых скобок (список параметров может отсутствовать). Под типом функции подразумевается тип возвращаемого ею значения.

Реализация функции – это уже раскрытие в виде операторов алгоритма ее работы. Шаблон реализации функции следующий:

<**тип**> <**идентификатор**>([**список параметров**]) **{**
[**список описаний переменных**]
[**последовательность операторов**]
}

Начало реализации аналогично определению, однако завершается не точкой с запятой, а ограниченным фигурными скобками ***телом функции***. Тело функции состоит из необязательного объявления переменных, принадлежащих только этой функции, и необязательной последовательности операторов, реализующих алгоритм работы функции. Необязательность всех элементов лишь подчеркивает, что такое допустимо синтаксисом языка, но вовсе не здравым смыслом.

Наконец, использование функции осуществляется по следующему шаблону:

<**идентификатор**>([**список значений**]);

Здесь **идентификатор** – тот же, что был задан при описании и реализации функции, а **список значений** – это список выражений, используемых для передачи в функцию параметров. Число выражений в этом списке обязательно должно совпадать с числом параметров в определении функции. Рассмотрим все вышесказанное более подробно на примере.

Предположим, необходимо иметь функцию вычисления квадрата гипотенузы по значению катетов прямоугольного треугольника. Определимся с входными и выходными данными. Очевидно, что значения длин катетов – это входные, а квадрат гипотенузы – выходные данные. Ограничим диапазон значений входных данных типом **int**, тогда очевидно для выходного значения потребуется тип **long**. Опишем функцию **q_hyp**, которая получает на входе два значения типа **int**, и возвращает результат типа **long** (воспользуемся шаблоном описания функции):

```
long q_hyp (int katet1, int katet2);
```

Далее необходимо выполнить реализацию функции:

```
long q_hyp(int katet1, int katet2){
       long result;
       result = katet1*katet1 + katet2*katet2;
       return result;
}
```

Следует обратить внимание на новый оператор *return*, имеющий очень простой шаблон:

return [**возвращаемое значение**];

Этот оператор вызывает *немедленное* завершение работы функции с заданным возвращаемым значением, которым может быть любое выражение.

В реализации функции использована дополнительная переменная **result** типа **long**. Эта переменная описана внутри тела функции и называется ***локальной*** переменной, т.е. доступной только внутри тела этой функции. Это означает, что *вне тела функции* переменная **result** *не существует*, в противовес ***глобальным*** переменным, описанным *вне* тела функции (в тексте программы), которые существуют и доступны как внутри функции, так и в остальной части программы. Функция может использовать любые глобальные переменные, но ее локальные переменные нигде, кроме ее тела, использовать нельзя.

Как же можно использовать только что определенную функцию? Любым из следующих способов:

```
int a = 2, b = 5;
long r;
r = q_hyp(2,5);          // r будет равно 29
r = q_hyp(a, 4);         // r будет равно 20
r = q_hyp(a, b);         // r будет равно 29
r = q_hyp(5 + a, b *2); // r будет равно 149
q_hyp(a,b);              // результат функции будет проигнорирован
r = 10 * q_hyp(a, b);    // r будет равно 290
```

То есть обращение к функции эквивалентно использованию переменной соответствующего значения. Кстати, если результат функции не используется – это не является ошибкой. Самое важное: переменным, определенным в качестве входных, присваиваются значения, фактически указанные при обращении функции. Говорят, что формальным параметрам присваиваются фактические значения.

Переменные-формальные параметры функции доступны в ее теле наравне с локальными, причем любые их изменения никак не отражаются на «внешней» программе. В только что рассмотренном примере **r = q_hyp(a, b)** никаким способом функция не может изменить значения переменных **a** и **b**, даже если бы в ее теле было написано **katet1 = 100** – действует правило локальности переменной **katet1**.

Теперь рассмотрим различные допустимые варианты реализации функций.

Функция, которая не возвращает значения.

Для такого случая существует особый вид типа – ***void*** (пусто). Этот тип означает буквально «нет значения». Для описанных как **void** функций недопустимо использование оператора **return** с указанием возвращаемого значения. Обычно функции без возвращаемого значения изменяют глобальные переменные. Пример:

```
void func(int x);
```

Функция без параметров.

Такая функция может как возвращать значение, так и обойтись без этого. При ее описании, реализации и использовании просто указываются круглые скобки с ключевым словом **void** внутри. Пример:

```
int func(void);
```

Следует так же оговорить особенность оператора **return**. Как было сказано, он вызывает *немедленное* завершение функции, т.е. в следующем примере функция всегда будет возвращать значение 100 вне зависимости от значения переданных ей параметров:

```
long q_hyp (int katet1, int katet2);
```

Далее необходимо выполнить реализацию функции:

```
long q_hyp(int katet1, int katet2){
        long result;
        return 100;
        result = katet1*katet1 + katet2*katet2;
        return result;
}
```

Если функция не возвращает значения, использование оператора **return** необязательно – в этом случае функция завершится после исполнения последнего оператора ее тела.

Внутри функции могут быть определены метки для использования совместно с оператором **goto**. Это *локальные* метки, т.е. оператор **goto** может осуществить переход лишь на метку в пределах тела функции, и никуда более. *Глобальных* меток в Си не существует (см. раздел «setjmp.h – Нелокальные переходы goto» для знакомства со способом обойти это ограничение).

Рекурсия

Особый случай использования функции – это обращение к ней изнутри ее тела. Такой метод получил название рекурсивного вызова функции, или рекурсии.

Существует большое количество алгоритмов, которые чрезвычайно сложно реализуются обычным, нерекурсивным образом, в то время как с использованием рекурсии решение получается простым и красивым. Среди них, например, различные алгоритмы сортировки, поиска и т.п.

Рассмотрим пример рекурсивной функции для вычисления факториала числа:

```
long long factorial (unsigned char N){
        long long result;
        if (N != 0) result = N * factorial(N-1);
        else result = 1;
        return result;
}
```

Рекурсивный алгоритм подсчета факториала заключается в том, чтобы обращаться самому к себе, но со значением входного числа на 1 меньше текущего значения, пока входное значение не станет равно нулю. Возможно, это не самый удачный пример эффективного рекурсивного алгоритма, однако он хорошо раскрывает главную опасность рекурсии.

Дело в том, что все локальные переменные размещаются в ОЗУ, причем при каждом новом обращении к функции происходит выделение для них новой области памяти. Чем «глубже» происходит погружение в рекурсивные вызовы, тем больше расходуется памяти. Это легко может привести к полному исчерпанию доступного пространства ОЗУ во время расчета, и узнать об этом будет чрезвычайно проблематично, ведь Си для микроконтроллеров не содержит никаких средств контроля подобных ситуаций.

Поэтому, не смотря на всю привлекательность рекурсивных алгоритмов, использовать их можно лишь с большой осторожностью.

Косвенное обращение к функции

Рассмотренные способы обращения к функции являются примерами непосредственного ее вызова, точно так же, как ранее непосредственно использовались значения переменных. Но при помощи указателя можно было получить доступ к содержимому переменной и косвенным образом, аналогичный способ существует и для обращения к функции.

Очевидно, реализация этого способа требует определения *указателя на функцию*. Делается это достаточно просто, путем объявления переменной соответствующего типа:

```
int (*funcptr)(int, int);
```

Этот пример объявляет переменную **funcptr** типа *указатель на функцию*, возвращающую результат типа **int** и имеющую 2 параметра типа **int**. При объявлении указателя на функцию запись выглядит слегка шиворот-навыворот, но так уж требует Си: сначала мы указываем тип результата функции, затем обязательно в круглых скобках идентификатор (не забудьте про звездочку – признак указателя), а затем, опять в круглых скобках, список типов параметров функции.

Сделанная запись – не что иное, как объявление переменной с новым типом. Можно определить этот тип при помощи **typedef**, после чего определять указатели на функции станет легче:

```
typedef int (*t_func)(int, int);
t_func funcptr;
```

Этот пример полностью аналогичен предыдущему.

Обращение к функции по указателю происходит чуть иначе, чем обращение к переменной по указателю: в данном случае операция разыменования указателя не используется:

```
funcptr = (t_func)0x1000;
funcptr(3, 12);         // вызов функции с параметрами по абсолютному
адресу
```

В этом примере можно было бы использовать и такое присваивание значения указателю:

```
funcptr = (void *)0x1000;
```

так как тип указателя **(void *)** совместим с любым иным типом указателя.

Препроцессор

Препроцессор – это отдельное средство предварительной обработки текста программы. Препроцессор изменяет текст программы автоматически перед началом работы компилятора. Эти изменения включают:

- «слияние» строковых констант
- удаление комментариев
- замена escape-последовательностей на фактические коды символов
- выполняются текстовые макро-подстановки (см. далее)
- включение и исключение участков текста программы

Директивы препроцессора

Управление работой препроцессора осуществляется при помощи специальных ключевых слов, названных директивами препроцессора.

Директива обязательно должна начинаться с символа «**#**» в *первой позиции* строки программы.

Набор директив может быть весьма обширен, но обязательно реализуются директивы включения файлов, определения макроподстановок и условной компиляции.

#include

Директива включения файла в текст программы. Формат директивы может быть двух вариантов:

#include <имя файла>

или

#include "имя файла"

Здесь в первом случае угловые скобки – неотъемлемый атрибут директивы. **Имя файла** – это имя любого текстового файла, возможно, с указанием пути к нему.

Действие этой директивы заключается в том, что в место нее в текст программы помещается содержимое указанного файла без изменений, т.е. текст программы расширяется содержимым файла. Дальнейшая обработка препроцессором уже ведется над этим «расширенным» текстом.

Никаких проверок содержимого файла не производится.

Первый вариант директивы (с указанием файла в угловых скобках) заставляет препроцессор искать файлы в особой директории, так называемой директории стандартных библиотек.

Второй вариант ограничивает поиск той директорией, где находится обрабатываемый файл, т.е. в директории пользователя.

Наличие двух вариантов директивы позволяет программисту заменять системные библиотечные файлы собственными.

Обычно этой директивой подключаются файлы с расширением **.h** – так называемые ***заголовочные файлы***.

#define

Директива определения *макроподстановки* (*макроса*).

Директива может иметь один из двух форматов:

#define <**идентификатор**> **[значение]**

или

#define <**идентификатор**>(**псевдо-параметры**) <**тело**>

Здесь **идентификатор** – любой допустимый синтаксисом идентификатор, **значение** – любой набор символов. **Псевдо-параметры** – это список (через запятую) идентификаторов, используемых в качестве составной части при определении **тела**, представляющего собой так же любой набор символов.

Первая форма директивы получила название директивы определения ***символа*** или ***простого макроса***, а вторая – директива определения ***макроса-функции*** (из-за внешнего сходства с определением функции).

Действие директивы в любом из форматов заключается в том, что препроцессор, встретив в тексте программы определенный директивой идентификатор, осуществляет *подстановку* вместо него соответствующего значения или тела, причем подстановка осуществляется именно путем ввода текстовых символов. Это, в частности, подразумевает, что *в качестве идентификатора можно использовать даже ключевые слова языка Си*!

Рассмотрим ряд примеров.

Пусть указаны следующие директивы:

```
#define D1 7
#define D2 + 5 *
#define D3 if
```

А далее в тексте программы эти символы используются так:

```
for (i = D1; i < D1 D2 3; i++)
       D3 (i == 5) break;
```

После того, как препроцессор обработает указанные строки, будет сгенерирован следующий текст программы:

```
for (i = 7; i < 7 + 5 * 3; i++)
       if (i == 5) break;
```

То есть вместо символов **D1**, **D2** и **D3** соответственно будут подставлены их значения. Смысл полученного текста лежит на совести программиста.

При объявлении макроса-функции перед тем, как осуществить замену символа в тексте программы, препроцессор осуществляет подстановку вместо псевдо-параметров их фактические значения, и лишь затем тело макроса подставляется в текст:

```
#define foo(x) x + 5

int k = foo(45*x);
// после обработки препроцессором заменится на

int k = 45*x + 5;
```

Важно понимать, что любой макрос – это именно текстовая подстановка, т.е. в отличие от настоящей функции не происходит никакой передачи параметров, вызова и т.п. – именно поэтому псевдо-параметр макроса не имеет типа.

Из-за того, что подстановки осуществляются, начиная с первого символа (не пробельного) после идентификатора и до конца строки, следует быть осторожными с использованием однострочных комментариев и точки с запятой. Следующие примеры явно неверные:

```
#define LINE 56 // определяем константу
#define cnt(x, y) while(x[y++] != ' ');

for (i = 0; i < LINE; i ++) arr[i] = i;
count = cnt(arr, i) * 12;
```

В результате обработки препроцессором будет сформирован следующий текст (комментарии из текста удаляются):

```
for (i = 0; i < 56
count = while(arr[i++] != ' '); * 12;
```

В первом операторе из-за наличия однострочного комментария в строке определения макроса будет удалена важная часть, а во втором получится сразу две ошибки: во-первых, макрос – это не функция, и в итоге оператор присваивания приобретает недопустимый вид, а во-вторых, умножение на 12 окажется отделенным точкой с запятой из тела макроса от остальной части оператора, что, естественно, так же недопустимо.

В связи с этими особенностями макросов настоятельно рекомендуется придерживаться следующих правил:

1. В определении макроса, если это необходимо, использовать только комментарии вида /* … */

2. Заключать в круглые скобки как все тело макроса, так и входящие в него псевдопараметры

3. Не завершать тело макроса точкой с запятой

Примеры правильно определенных макросов:

```
#define LINE 56 /* числовую константу можно без скобок */
#define cnt(x,y) ( (x) * (y) ) /* скобки лишними не бывают! */
```

Если определяемый макрос не умещается на одной строке, допускается использовать символ переноса – обратная черта.

Допускается повторно определять один и тот же макрос в разных местах программы, это сопровождается предупреждением компилятора, но, в сущности, безопасно. Переопределенный макрос начинает действовать для строк программы, находящихся ниже его определения, а выше действует старое определение.

#undef

Директива, которая отменяет описание макроса, ранее сделанное директивой **#define**.

Формат директивы очень прост:

#undef <**идентификатор**>

С момента появления в тексте программы этой директивы, макрос, определяемый **идентификатором**, перестает существовать для препроцессора. Если после этого снова определить макрос с тем же идентификатором – компилятор не выдаст предупреждения.

Директивы условной компиляции

Группа директив препроцессора, позволяющих в зависимости от значения определенных символов (макросов) изменять содержимое текста программы: **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif** и **#endif**. Эти директивы *сложные*, т.е. состоят из более чем одной строки. Начинаются всегда с одной из первых трех директив, завершаются всегда директивой **#endif**. Оставшиеся директивы **#else** и **#elif** – дополнительные, могут и отсутствовать.

Эти директивы, как по синтаксису, так и по смыслу, очень близки к оператору **if** и служат для проверки определенного условия, и, в зависимости от истинности этой проверки, исключения указанных строк программы из процесса компиляции. Пояснить сказанное проще всего на примерах.

```
#if (VER > 5)
//строки, которые будут обработаны только в случае, если VER > 5
int x = 10;
#endif
```

В примере строка **int x = 10** будет включена в текст программы только в том случае, если значение ранее описанного символа **VER** будет больше 5. Если же символ **VER** не определен вообще, или его значение менее 5 – строка будет удалена препроцессором.

```
#if (VER > 5)
int x = 10;
#else
long x = 10;
#endif
```

В этом примере в текст программы будет введено определение переменной **int** x только для случаев, когда **VER > 5**, а в противном переменная x будет иметь тип **long**. (*см. следующую страницу*)

```
#if (VER > 5)
int x = 10;
#elif (VER > 3)
char x = 10;
#else
long x = 10;
#endif
```

Этот пример показывает, как изменяется тип переменной **x** в зависимости от значения **VER**:

Значение VER	Тип переменной x
6 и более	int
4 или 5	char
3 и менее	long

Количество директив **#elif** может быть любым.

Если не требуется анализировать значение символа, а достаточно лишь убедиться в том, что он определен, используется такой вариант:

```
#ifdef VER
int x = 10;
#endif
```

Если символ **VER** определен «выше по тексту» программы – определение переменной включается в текст программы, иначе – исключается.

Директива **#ifndef** позволяет что-то включить в текст программы, если символ НЕ описан ранее. Обе директивы могут содержать блок **#else** для указания альтернативного включения текста.

Не нужды упоминать, что указанные директивы могут «охватывать» любое количество строк программы, в том числе любые другие директивы препроцессора. Главное, о чем нельзя забывать: директивы препроцессора не могут использоваться для контроля содержимого переменных! То есть определить, существует или нет переменная var при помощи директивы #ifdef var можно, но будет ошибочной попытка проверить ее значение директивой #if var == 15.

Структура программы

Ранее была использована аналогия между разговорным языком и языком программирования, благодаря чему были введены понятия алфавита, лексем и т.п. Однако, «вызубрить» словарь и правила построения предложений еще не достаточно, чтобы на равных общаться с иностранцем, необходимо знать еще характерные идиомы, традиции и т.п. – все то, что создает атмосферу взаимопонимания. Аналогичная ситуация и в программировании: чтобы компилятор без проблем воспринял программу, следует придерживаться определенных правил в ее организации.

Ряд этих правил относятся к категории «неписаных», т.е. формально не обязательных, но очень и очень желательных к применению. Кроме этого существуют строгие правила, которых, к счастью немного.

Основные понятия

Рассмотрение структуры программ будет осуществлено на основе наиболее сложного, всеобъемлющего случая. Менее сложные (и более близкие к реальным любительским проектам) получаются путем соответствующего упрощения, о чем по мере изложения будет упоминаться особо.

Существует ряд устоявшихся терминов в среде программирования, которые будут использованы и в этой книге.

Проект – совокупность всех файлов, необходимых для выполнения поставленной перед программистом задачи. Включает в себя файлы исходных текстов программ, документацию, промежуточные файлы, генерируемые компилятором, и главное: результат компиляции проекта – загружаемый файл.

Загружаемый файл – результат работы компилятора, файл, содержимое которого готово для загрузки в микроконтроллер. Иногда используют неофициальный термин «прошивка».

Исходный текст программы, как правило, состоит из более чем одного файла. Каждый отдельный файл, составляющий программу, называют модулем. Модуль обычно строится из описаний переменных, типов, макросов, функций и т.п., сгруппированных по функциональному назначению. Например, программа может состоять из модуля работы с дисплеем, модуля обслуживания клавиатуры и т.д. Обязательным является наличие как минимум одного модуля – главного, который, как правило, имеет название, совпадающее с названием всего проекта.

Главный модуль использует, в том числе, определения, сделанные в других модулях. Основное его отличие в том, что он обязательно должен содержать реализацию (описание не требуется) главной функции **main()** – это зарезервированное имя для главной функции. Реализация этой функции должна быть следующей:

```
int main(void){
        <тело функции>
}
```

Главная функция автоматически получает управление после начала работы программы, т.е. она вызывается самой первой. Завершение функции **main()** равносильно прекращению любой работы. Для микроконтроллеров несущественно значение, которое функция возвращает, поэтому в теле функции нет необходимости использовать оператор **return**.

До реализации любой функции в любом модуле должны быть сделаны все необходимые объявления (описания) используемых в этой функции типов, констант, переменных и т.п. В частности, должны быть сделаны подключения файлов, в которых описаны функции из других модулей проекта. Такие файлы получили название заголовочных и традиционно имеют расширение **.h** (от англ. *header* – заголовок).

Как правило, каждый модуль имеет свой заголовочный файл. В нем должны быть описаны все функции, переменные, типы, макросы и т.п., словом, все то, что должно быть доступно в главном модуле (или любом другом, если это требуется). Реализация функций выполняется в файле с исходным текстом модуля, который имеет традиционное расширение **.c**. Файл заголовка включается директивой **#include** в соответствующий модуль и, таким образом, в этом модуле становятся доступны все объявленные возможности.

Деление содержимого исходного текста модуля – условное, т.е. не обязательное. Ничто не препятствует подключать в главный модуль сразу исходный текст вспомогательного, однако такой подход не одобряется.

Может быть так же определен заголовочный файл для всего проекта в целом. Обычно в него помещают определения макросов и символов, которые используются всеми модулями проекта. Например, описав символ DEBUG, можно при помощи директив условной компиляции изменять логику работы любого из модулей во время отладки.

Локальность и глобальность

В проекте, состоящем из нескольких файлов, изобилующих определениями переменных, функций и т.п., необходимо четко ориентироваться в том, какие описанные элементы доступны в тех или иных файлах.

Считается, что любое описание действительно только внутри блока, ограниченного фигурными скобками, где это определение реализовано. Для модуля аналогом фигурных скобок выступают начало и конец файла модуля. Любое объявление считается локальным по отношению к своему блоку.

Таким образом, все объявления, сделанные в начале модуля, «видимы» и доступны для любых функций этого модуля. Такие объявления называют глобальными для данного модуля. Итак, каждое объявление может быть локальным и глобальным одновременно, все зависит от точки отсчета.

Чтобы обеспечить видимость глобальных объявлений модуля из других модулей, необходимо вынести их в заголовочный файл, который подключить директивой **#include** в нужном модуле. Объявлять переменные в заголовочном файле – неверная практика, т.к. это может привести к многократному переобъявлению одних и тех же идентификаторов в различных модулях, т.е. всюду, где заголовочный файл будет подключен. Переменные и так по умолчанию считаются «видимыми» для других модулей, хотя для того, чтобы

воспользоваться переменной из стороннего модуля, необходимо объявить ее *внешней*.

Кроме видимости, стоит упомянуть еще о действительности значений глобальных переменных. Считается, что глобальная переменная сохраняет свое значение в пределах блока своего описания. То есть, если значение глобальной переменной ***Cnt*** будет изменено функцией ***foo()***, то это значение будет действительно и для вызванной следом функции ***fee()***.

Внешние функции и переменные

По отношению к текущему модулю все переменные и функции из других модулей считаются внешними, т.е. недоступными. Если необходимо использовать эту внешнюю переменную, ее нужно объявить в модуле с указанием ключевого слова **extern**:

```
extern int Cnt;
```

Пример показывает объявление внешней переменной **Cnt**. При компиляции такой программы компилятор зарезервирует адрес этой переменной, не определяя его конкретное значение, а компоновщик при сборке загружаемого файла будет осуществлять поиск во всех модулях объявление переменной **Cnt** и, если найдет, присвоит значение ее адресу. Таким образом, внешние переменные становятся доступны в модуле лишь после компоновки.

Для обращения к функции из другого модуля в большинстве случаев не требуется указания служебного слова **extern** – достаточно простого наличия ее объявления в подключаемом заголовочном файле, действия компоновщика при этом аналогичны рассмотренным. Иное дело, если функция определена, но исходного текста соответствующего модуля нет. Такая ситуация возможна, если вместо исходного текста модуля имеется объектный файл. В этом случае объектный файл включается в обработку компоновщиком, и для объявленной функции использование ключевого слова **extern** является обязательным.

Иными словами, слово **extern** означает, что адрес объекта должен быть найден не компилятором, а компоновщиком на основе содержимого объектных файлов, используемых для сборки проекта.

Излишне говорить, что фактический тип внешней переменной или функции должен совпадать с объявленным, а для функций еще должны совпадать типы и количество параметров. Так как на этапе компоновки уже нет возможности точно сверить соответствие фактической реализации объявлению, несоблюдение этого правила чревато крупными проблемами. Обычно, в этом случае генерируется лишь предупреждение, а не ошибка.

static-переменные

Ранее уже говорилось о сохранении значений глобальных переменных. Однако очень часто требуется, чтобы локальная переменная внутри функции сохраняла бы свое значение от вызова к вызову. Например, многие алгоритмы генерации псевдослучайных чисел для получения нового числа используют значение, возвращенное в предыдущий раз.

Возможность сохранять значение локальной переменной есть – для этого служит ключевое слово ***static*** (т.е. статическая), добавляемое к типу переменной при ее определении. Например:

```
int func(void){
        static int result = 0;
        return ++result;
}
```

При первом обращении к функции **func()** она вернет результат 1, при втором – 2, при третьем – 3 и т.д., т.е. будет возвращать порядковый номер обращений к ней. Для хранения этого номера используется локальная переменная **result**, объявленная статической.

Инициализация статической переменной происходит лишь единственный раз, при первом обращении к функции, а все прочее время эта переменная сохраняет ранее присвоенное ей значение.

Об оптимизаторе программ

Рассмотренные средства языка Си позволяют строить программы произвольной сложности. Удобство языка высокого уровня может создать ложное впечатление о «вседозволенности» или «безграничных возможностях» языка, т.е. может показаться, что любая программа будет очень качественной лишь благодаря средствам Си, а от программиста не требуется повышенных усилий при ее разработке, как, например, это необходимо при работе на ассемблере.

Увы, использование языка высокого уровня вызывает неизбежное увеличение размера итогового программного кода, повышает требовательность программы к памяти и быстродействию микроконтроллера. Для систем на базе персональных компьютеров с сотнями мегабайт доступного ОЗУ и гигагерцами тактовой частоты процессора эти проблемы, возможно, и не особенно актуальны, но не для микроконтроллеров AVR, с их единицами килобайт ОЗУ (а то и десятками байт) и весьма скоромной производительностью ядра.

Чтобы хоть как-то снизить остроту этой проблемы, все компиляторы Си содержат средства встроенной оптимизации генерируемого кода. В чем же заключается оптимизация и как она осуществляется компилятором?

Обратимся к функции, рассмотренной в разделе «static-переменные». Если задуматься, то как бы эта функция не использовалась в программе, ее можно с успехом заменить обращением к переменной **result** с автоинкрементом (если бы эта переменная была не локальной, а глобальной), и тем самым исключить лишний код и расходы стековой памяти на реализацию обращений к функции. Но программист, предположим, не посчитал нужным так поступить, и тогда в дело вступает оптимизатор компилятора, который незаметно изменяет логику программы, не нарушая ее принципиально. То есть вместо обращения к функции **func()** компилятор встраивает код **result++**, переместив при этом переменную **result** из области локальных переменных к глобальным. Для программиста все остается, как было, но фактически программа уже немного не та…

Другой способ оптимизации заключается, например, в том, что компилятор может добиться более высокой скорости выполнения цикла, заменив его на следующие друг за другом участки одинаковой функциональности, т.е. фактически аннулируя смысл оператора цикла. «Развернутый» цикл может быть больше по размеру кода, но часто оказывается оптимальнее по требованиям к ОЗУ и более быстродействующим.

Практически всегда компилятор заменяет функцию, которая используется лишь однажды в программе на эквивалентные операторы, помещенные прямо в месте вызова функции. Так же у компилятора хватает интеллекта разобраться с неизменными значениями выражений и заменить несколько операторов сразу константой результата. Пример:

```
define MAX 15
int sum, i = 10;
while (i < MAX){
        if (i == 9) break;
        i++;
        sum *= i;
}
```

Если проанализировать этот участок кода, то окажется, что несмотря на всю его сложность, значение переменной **sum** всегда будет равно нулю. Но это значение переменная и так получает по умолчанию при описании, следовательно, оператор цикла можно устранить из программы, никак не нарушив ее функционирования.

Этот пример, конечно, из разряда казусов. Однако такие казусы случаются нередко, приводя, порой, к занимательным результатам: если представить, что значение **sum** используется далее в программе, то автоматически могут быть исключены и другие участки кода:

```
if (sum > 10)
        x = x + sum;
else
        x = x * sum;
delta = (x + i * 12)/2;
```

И в этом участке лишним окажется оператор **if**, более того, значение переменной **x** так же оказывается предопределенным, и

компилятор может счесть ее вовсе лишней, исключив и другие участки, где эта переменная задействована. В общем, может оказаться, что написанная с большими усилиями программа не делает ничего… Конечно, это свидетельствует в первую очередь о недостаточно хорошо продуманном алгоритме, либо о грубых ошибках программирования – к сожалению, интеллекта никакого оптимизатора не хватит найти и исправить такие ошибки.

С другой стороны, программист сам может предпринимать меры по оптимизации своей программы. Например, один из известных способов увеличения быстродействия программы (в ущерб ее размеру) заключается в принудительной замене вызовов функций их содержимым в каждом месте, где функция используется. Такой прием получил название встраиваемых в код функций, или (более традиционный термин) ***inline***-функций.

Ключевое слово **inline**, использованное при описании функции, указывает компилятору (а точнее – его оптимизатору) при первой же возможности заменить обращения к функции явной вставкой в текст соответствующих операторов. В этом случае функция превращается в некое подобие макроса-функции. Однако само по себе ключевое слово **inline** еще *ничего не гарантирует* – компилятор все равно поступит так, как посчитает наиболее оптимальным.

Другая проблема, связанная с работой оптимизатора – это переменные, которые изменяют свое значение не в том модуле, где они описаны, а в другом, для которого они являются внешними. Например, в модуле **interface.c**, отвечающем за интерфейс с пользователем, определена переменная **key** для хранения кода нажатой клавиши, а значение в нее записывается в одной из функций модуля **keyboard.c**, отвечающего за опрос клавиатуры (в модуле **keyboard.c** переменная **key** описана как **extern**). В этом случае далеко не исключен (а точнее – закономерен) вариант, что при оптимизации модуля **interface.c** компилятор сочтет, что переменная **key** никогда не меняет своего значения, и исключит весь код, связанный с анализом ее значения. Естественно, получившаяся программа будет полностью неработоспособна.

Подобное поведение компилятора с оптимизатором породило массу мифов о «неправильных» компиляторах или ошибках в них. Однако, ошибка тут лишь одна – программиста. Чтобы оптимизатор умерил свою прыть и не трогал переменные, проанализировать поведение которых ему не по силам, программист *обязан* объявлять их как *изменяемые*, т.е. ***volatile***. В этом случае и сама переменная, и связанный с нею код не будут «испорчены» оптимизатором.

Вариантов стратегии оптимизации программы, как правило, три: достижение ***минимума размера***, достижение ***максимума скорости*** исполнения и ***универсальная***, т.е. достижения оптимального соотношения размера и производительности. Обычно программисту предоставляются средства для более «тонкой» настройки оптимизатора под свои нужды. Кроме прочего, часто предлагаются средства *отступить от стандарта* Си или «обычного поведения» операторов и стандартных функций, чтобы выиграть в чем-то другом. С такими средствами программист должен быть особо осторожен, т.к. любое отступление от стандарта может повлечь проблемы. Однако грамотное использование всех средств позволяет достичь существенного выигрыша по всем статьям.

Далее в разделе «Оптимизация» рассмотрены основные средства оптимизации, которыми располагает программист при работе с компилятором GCC для микроконтроллеров AVR

О стиле программирования

Эта глава является заключительной в кратком описании языка Си и посвящается важной части в работе программиста, которой зачастую пренебрегают. Речь пойдет о стиле программирования или, точнее, о стиле оформления программ.

Большинство программистов-одиночек разрабатывают программы в твердой уверенности, что делают это «для себя», что никто никогда не заинтересуется в содержимом исходных текстов их программ, и потому нет смысла тратить время и силы на всякие «излишества» и «украшения». Это в корне неверное предположение! Известен афоризм «всякая хорошо работающая вещь – красива», который вполне применим и к программе.

Ранее было показано, что язык Си допускает много разных способов[14] в составлении выражений, в записи операторов и т.д. Очень часто такое изобилие «инструментария» воспринимается программистом как стимул писать витиеватые программы, изобилующие огромным количеством макросов, переопределяющих едва ли не все ключевые слова языка, многократно и часто без необходимости вложенными друг в друга операторами и т.п. Все это выдается за высокий класс программиста, но фактически является лишь одним из проявлений каких-то комплексов, ничего общего с квалификацией не имеющих.

Качественная программа должна легко читаться, для чего порой комментариев в ней оказывается больше, чем собственно операторов. Кроме комментариев, существует и еще способ улучшить читаемость программного текста. Речь идет о так называемом «самодокументировании» программы. Одно время популярной была так называемая *венгерская нотация*, при которой любой идентификатор содержит в своем начале особым образом закодированный тип (например, все строки начинаются с символа **s**, указатели с **ptr**, целые числа – с **i** и т.п.), однако принципиальным можно считать лишь осмысленность любого идентификатора.

Если программа обрабатывает строки, являющиеся именами и фамилиями людей, то более удачным следует признать такие определения:

```
#define MAX_PERSONAL_COUNT 100
typedef struct{
        char name[];
        char gender[];
        char age;
} people_struct;

people_struct personal_list[MAX_PERSONAL_COUNT];
```

чем более короткий, но эквивалентный по смыслу вариант

```
struct {
        char a1[];
        char a2[];
        int a3;
} arr[100];
```

Для человека, хоть немного владеющего английским, первый вариант расскажет сам за себя: первая строка определяет максимальное количество персон, далее следует определение нового типа-структуры с полями Имя, Пол и Возраст, далее объявляется массив списка персонала. Вторая же запись понятна лишь в программистском смысле, т.е. уловить связь массивов и полей структуры с реально осуществляемыми действиями невозможно. Спустя определенный срок программист, написавший программу в первом стиле (и потративший на

[14] В этой книге изложены далеко не все особенности синтаксиса языка Си – с целью упрощения. Показан лишь необходимый минимум для начала освоения.

это немного больше времени), вспомнит все тонкости своей программы за 5 минут, в то время как второй (состряпавший свой вариант на пару часов быстрее) наверняка потратит не один день на это[15]…

Для качественного оформления текста программы обязательно следует использовать выделение иерархически связанных участков при помощи отступов – это значительно улучшает восприятие заложенного в программу алгоритма. Фигурные скобки, ограничивающие тело функций и операторов, принято размешать одним из следующих способов:

```
for (;;)
{
        // уровень операторов цикла
}
```

```
for (;;){
        // уровень операторов цикла
}
```

То есть в первом способе открывающая и закрывающая скобки оператора находятся на одном уровнем с самим оператором, но на разных строках, а тело сдвинуто вправо. Во втором случае открывающая скобка находится рядом с оператором, а закрывающая – на уровне оператора. Оба способа позволяют, проследив вертикальные уровни скобок, легко разобраться в уровне вложенности операторов:

```
for(;;)
{
        while (x)
        {
                if (x)
                {
                        a = x *2;
                }
                else
                {
                        a = x + 2;
                }
        }
}
```

```
for(;;){
        while (x){
                if (x){
                        a = x *2;
                }else{
                        a = x + 2;
                }
        }
}
```

Желательно, выбрав один раз приемлемый для себя стиль оформления программ, придерживаться его все время. Постепенно это станет привычкой и не будет восприниматься рутинной обязанностью.

Очень желательно оформлять в виде функций повторяющиеся в разных местах программы участки. Как правило, если участок повторяется 2 или более раз – его стоит оформить в виде функции. Иногда очень удобно оформлять функции не по количеству повторов, а по смыслу выполняемых действий. Например, часто бывает удобно «набросать костяк» алгоритма, обозначив его характерные этапы функциями, которые затем реализовать:

```
int main(void){
        init_all(); // инициализация всей периферии
        while(1){ // главный цикл
                display(); // обновление дисплея
                if(test_key() ){ // проверка нажатия клавиатуры
                        // если кнопка нажата – проанализировать и обработать
                        switch (get_key()){
                        case 0 : press_key0(); break
                        case 1: press_key1(); break;
                        }
                }
        }
}
```

В этом простом примере показана «заготовка» едва ли не любой программы для микроконтроллера, взаимодействующей с пользователем при помощи кнопок и дисплея. Составив такой скелет, программист может сосредоточиться на реализации уже придуманных функций ***get_key()*** – получение кода нажатой кнопки, ***display()*** – обновления содержимого дисплея и т.д.

Такой порядок разработки программы можно назвать «от общего к частностям» (иногда используют термин *нисходящее* программирование). Соответственно, при реализации функций так же можно применить этот способ. Заботиться о количестве функций и о том, что многие из них используются лишь один раз, не стоит – оптимизатор все сделает лучшим образом. Программист должен думать, а процессор – работать.

[15] Видимо, на этом основана поговорка программистов «легче сделать все с начала, чем разобраться в ранее написанном».

Продолжение в №4/2010