Проектирование микропроцессорных систем и логических контроллеров

1. Назначение и применение стека в программах для микроконтроллеров.
2. Подпрограммы-процедуры и подпрограммы-функции.
3. Структурированное программирование. Структуры линейные, с ветвлением, циклические.
4. Программы многофайловые и многомодульные. Программа-монитор.
5. Программирование на языке ASM-51: порядок создания программы для микроконтроллера; символы языка.
6. Идентификаторы в языке ASM-51: ключевые слова.
7. Идентификаторы в языке ASM-51: встроенные и определяемые имена.
8. Программирование на языке ASM-51: числа и литеральные строки.
9. Директивы языка программирования ASM-51: equ, set, db, dw.
10. Директивы языка программирования ASM-51: org, using; управляющие команды: debug, include.
11. Использование сегментов в языке программирования ASM-51: абсолютные сегменты.
12. Использование сегментов в языке программирования ASM-51: перемещаемые сегменты.
13. Программирование на языке С-51: структура программы, символы языка и управляющие последовательности.
14. Программирование на языке С-51: идентификаторы, ключевые слова, простые и составные ограничители, числовые и текстовые строковые константы.
15. Типы данных языка программирования С-51 и их объявление.
16. Категории типов данных: целые, с плавающей запятой, перечислимые.
17. Программирование на языке С-51: указатели.
18. Программирование на языке С-51: массивы.
19. Программирование на языке С-51: структуры.
20. Программирование на языке С-51: объединения.
21. Программирование на языке С-51: определения новых типов и инициализация данных.
22. Программирование на языке С-51: выражения, операнды и операции.
23. Программирование на языке С-51: преобразования типов при выполнении выражений.
24. Программирование на языке С-51: операции вычисления адреса и разадресации; поразрядные логические операции.
25. Программирование на языке С-51: мультипликативные и аддитивные операции, операции сдвига, операция последовательного вычисления.
26. Программирование на языке С-51: условная операция, инкрементирование и декрементирование, простое и составное присваивания; приоритеты операций.

27. Программирование на языке С-51: операторы (пустой, «выражение», составной).
28. Программирование на языке С-51: условные операторы (if, switch).
29. Программирование на языке С-51: операторы цикла (for, while, do - while).
30. Программирование на языке С-51: операторы перехода (break, continue, return, goto).
31. Программирование на языке С-51: определения и вызов функций.

Красные - вопросы для билетов на 3.

4 билет: 4,15 вопросы
7 билет: 8,13 вопросы
5 билет: 19,26 вопросы
16 билет: 16,31 вопросы
9 билет: 9,27 вопросы
2 билет: 22,2 вопросы

младшие курсы неблагодарные какахи и это не шутка. какой год говорят спасибо за то, чего у нас не было, за то что мы сами делали :)

1 Назначение и применение стека в программах для микроконтроллеров

Стек – часть программного пространства данных, в которой хранится информация для временного использования. (адреса возврата из подпрограмм хранятся в области памяти, называемой стеком)

Логически доступ к этим ячейкам памяти организован так, чтобы считывание последнего записанного адреса возврата производилось первым, а первого — последним.

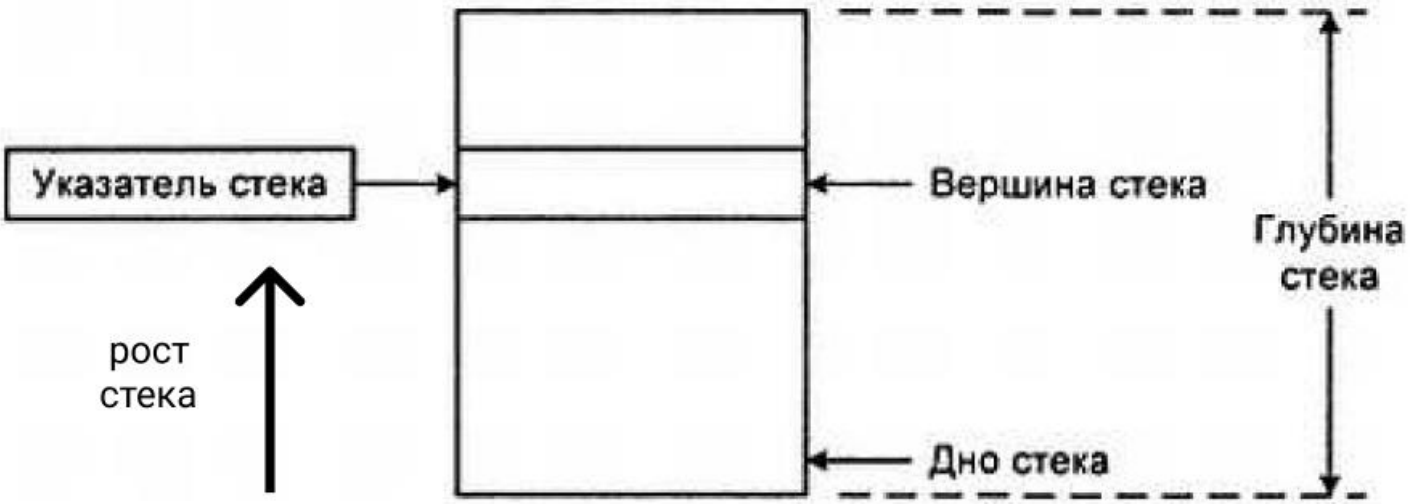


Рисунок 1.1 – Организация стека в памяти данных микропроцессора

Адресация ячеек стека осуществляется с использованием специального регистра, называемого указателем стека (SP). Он хранит адрес ячейки, в которую можно записать следующие данные. Ячейка памяти, в которую в данный момент может быть записан адрес возврата из подпрограммы, называется вершиной стека. Ее адрес всегда хранится в указателе стека. Количество ячеек памяти, выделенных для стека, называется глубиной стека. Глубина стека должна быть рассчитана как минимум на кол-во вложенных подпрограмм (она устанавливается при помощи записи начального адреса вершины стека в регистр SP). Обычно глубина стека устанавливается один раз после включения питания в процедуре инициализации контроллера.

При занесении информации в стек содержимое указателя стека увеличивается (стек растет вверх), поэтому стек размещается в самой верхней части памяти данных.

Чтобы пользоваться стеком, надо установить его глубину. Для того чтобы установить глубину стека 28 байт, необходимо вычесть из адреса максимальной ячейки внутренней памяти микроконтроллера глубину стека и записать полученное значение в указатель стека SP:

```
DnoSteka EQU 127          ; для AT89c51 размер внутренней памяти равен 128 байтам
MOV SP, #DnoSteka – 28    ; установить глубину стека 28 байт
```

Для работы со стеком в систему команд микропроцессоров введены специальные команды – PUSH и POP. PUSH заносит в стек значения регистров и ячеек памяти, которые надо сохранить на время, а POP – перед выходом из подпрограммы восстанавливает эти сохраненные данные.

POP - чтение из стека, PUSH - запись в стек.

```
Podprogramma:
PUSH PSW        ; Сохранить содержимое
PUSH ACC        ; используемых в
PUSH R0         ; подпрограмме регистров
…               ; Текст подпрограммы
POP R0          ; Восстановить содержимое
POP ACC         ; используемых в
POP PSW         ; подпрограмме регистров
RET             ; вернуться из подпрограммы
```

Я бы еще это порекомендовал прочесть

2 Подпрограммы-процедуры и подпрограммы-функции

Подпрограммы предназначены для выполнения определенных действий над находящимися внутри микросхемы периферийными устройствами, внешними устройствами, подключенными к выводам микросхемы микроконтроллера, или числами, хранящимися в его внутренней памяти.

В любом случае, с точки зрения программы, операции производятся над переменными. Переменные могут быть локальными (доступными только из подпрограммы) или глобальными (доступными из любого места программы).

Если подпрограмма осуществляет действия над глобальными переменными или выполняет определенный набор действий, то такая подпрограмма называется **процедурой**. Эта подпрограмма может осуществлять управление какими-то устройствами или осуществлять какие-либо вычисления. Если производятся вычисления, то результат помещается в глобальную переменную для того, чтобы этим результатом могла воспользоваться другая подпрограмма или основная программа. Пример фрагмента программы управления последовательным портом, написанного на языке высокого уровня С-51:

```
…
G_Per = 56;              // Занести передаваемое число в глобальную переменную
PeredatByte ();          // Передать это число
…
G_Per = 37;              // Занести передаваемое число в глобальную переменную
PeredatByte ();          // Передать это число
…

void PeredatByte(void)   // Подпрограмма передачи байта через последовательный порт
{
do; while (TI == 0);     // Если предыдущий файл передан,
SBUF = G_per;            // то передать очередной байт
}
```

Часто подпрограмма должна выполнять действия над каким-либо числом, значение которого неизвестно в момент написания программы. Это число можно передать через глобальную переменную, как в приведенном выше примере подпрограммы. Однако намного удобнее использовать подпрограмму с параметрами.

Параметры подпрограммы – локальные переменные подпрограммы, начальные значения которым присваиваются в вызывающей программе или подпрограмме. В алгоритмическом языке С-51 параметры подпрограммы записываются в скобках после ее имени. Пример вызова такой подпрограммы:

```
…
PeredatByte {56};
```

```
…
PeredatByte {57};
…

void PeredatByte{char byte}// подпрограмма передачи байта через последовательный порт
{
do; while (TI == 0);        // Если предыдущий байт передан,
SBUF = byte;                // то передать очередной байт
}
```

Часто требуется передавать результат вычислений из подпрограммы в основную программу. Для этого можно воспользоваться подпрограммой-функцией.

Подпрограмма-функция – подпрограмма, которая возвращает вычисленное значение. Пример использования подпрограммы-функции: Y = sin (x);

Использование подпрограмм-функций позволяет приблизить текст программы к математической записи выражений, которые необходимо вычислить, а также увеличивает наглядность программ и, в результате, повышает скорость написания и отладки программного обеспечения.

Подпрограммы-функции обычно возвращают значения простых типов, таких как байт, слово или целое. Однако при помощи указателя можно возвращать и значения более сложных типов, таких как массивы переменных, структуры или строки. Например:

```
y=sin(x);                       // sin - это имя подпрограммы-функции
if (rus(c)) SvDiod = Gorit;     // rus - это имя подпрограммы-функции
```

Выполнение вызова функции происходит следующим образом:

1. Вычисляются выражения в списке выражений и подвергаются обычным арифметическим преобразованиям. Затем тип полученного фактического аргумента сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо формируется сообщение об ошибке. Число выражений в списке выражений должно совпадать с числом формальных параметров. Если в прототипе функции указано, что ей не требуются параметры, а при вызове они указаны, формируется сообщение об ошибке.
2. Происходит присваивание значений фактических параметров соответствующим формальным параметрам.
3. Управление передается на первый оператор функции.

Рассмотрим пример использования указателя на функцию в качестве параметра функции вычисляющей производную от функции cos(x):

```
float proiz (float x, float dx, float (*f)(float x));
float fun (float z);
```

```
int main ()
{
float x;                          //  точка вычисления производной
float dx;                         //  приращение
float z;                          //  значение производной
scanf("%f, %f", &x, &dx);         //  ввод значений x и dx
z = proiz(x,dx,fun);               //  вызов функции
printf("%f",z);                   //  печать значения производной
}
float proiz (float x, float dx, float (*f)(float z))// вычисляющая производную функция
{
float xk, xk1;
xk = fun (x);
xk1 = fun (x+dx);
return (xk1/xk-1e0)*xk/dx;
}
float fun (float z)       // функция от которой вычисляется производная
{
return (cos(z));
}
```

Для вычисления производной от какой-либо другой функции можно изменить тело функции fun или использовать при вызове функции proiz имя другой функции. В частности, для вычисления производной от функции cos(x) можно вызвать функцию proiz в форме z=proiz(x,dx,cos);, а для вычисления производной от функции sin(x) в форме z=proiz(x,dx,sin);.

3 Структурированное программирование. Структуры линейные, с ветвлением, циклические

3.1 Линейная структура

3.1.1 C-51

```
void ProchitatPort (void)          // Подпрограмма чтения порта
{
…                                  // Это подпрограмма-заглушка
}
void VklychitIndicator (void)      // Подпрограмма включения индикатора
{
…                                  // Это подпрограмма-заглушка
}
void main (void)                   // Главная программа
{
ProchitatPort ();                  // Прочитать порт
VklychitIndicator ();              // Включить индикактор
}
```

3.1.2 ASM-51

```
…                                  ; Главная программа
call Deistvie1                     ; Подпрограмма 1
call Deistvie2                     ; Подпрограмма 2
Deistvie1:                         ; Определение подпрограммы 1
…                                  ; Это подпрограмма-заглушка
ret                                ; Возврат из подпрограммы
Deistvie2:                         ; Определение подпрограммы 2
…                                  ; Это подпрограмма-заглушка
ret                                ; Возврат из подпрограммы
```

3.2 Структура с ветвлением (выполнением по условию)

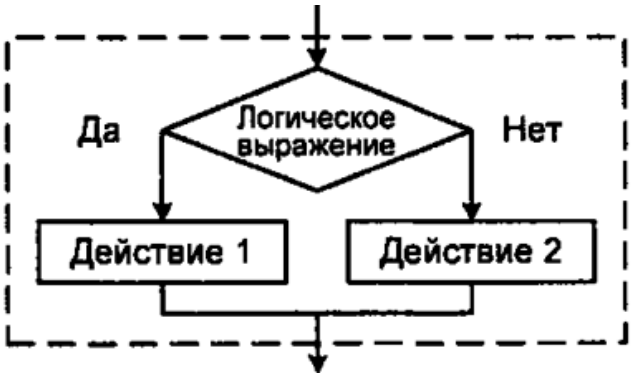


Рисунок 3.1 – Структура с ветвлением

3.2.1 C-51

```
void Deistvie1 (void)              // Подпрограмма 1
{
```

```
…                                   // Это подпрограмма-заглушка
}
void Deistvie2 (void)
{
…                                   // Это подпрограмма-заглушка
}
...
if (PrinyatByte)                    // Выражение
Deistvie1 ();                       // Реализация ветви 1
else
Deistvie2 ();                       // Реализация ветви 2
```

3.2.2 ASM-51

```
…                                   ; Главная программа
jb PrinyatByte, ElseUsl             ; Условное выражение
call Deistvie1                      ; Реализация ветви 1
jmp EndUsl                          ; Конец её реализации
ElseUsl:                            ; Реализация ветви 2
Call Deistvie2                      ;
EndUsl:                             ; Конец реализации ветвей

Deistvie1:                          ; Определение подпрограммы 1
…                                   ; Это подпрограмма-заглушка
ret                                 ; Возврат из подпрограммы
Deistvie2:                          ; Определение подпрограммы 2
…                                   ; Это подпрограмма-заглушка
ret                                 ; Возврат из подпрограммы
```

3.3 Циклическая структура

3.3.1 С проверкой после тела цикла

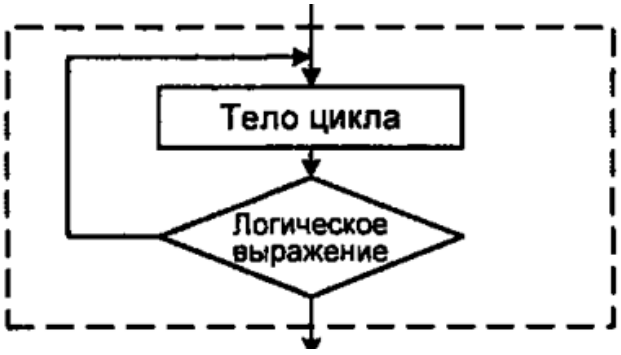


Рисунок 3.2 – Циклическая структура с проверкой после тела цикла

3.3.1.1 С-51

```
void TeloCikla (void)               // Определение подпрограммы
{
…                                   // Это подпрограмма-заглушка
```

```
}
…
do
TeloCikla ();                    // Реализация тела цикла
while (1);                       // Проверка (в данном случае – бесконечно)
```

3.3.1.2 ASM-51

```
...                              ; Главная программа
Nachalo:                         ; "Место" до тела цикла (рис. 3.2)
call TeloCikla                   ; Реализация тела цикла
jb PrinyatByte, Nachalo          ; Условное логическое выражение
…
TeloCikla:                       ; Определение подпрограммы
…                                ; Это подпрограмма-заглушка
ret                              ; Возврат из подпрограммы
```

3.3.2 С проверкой до тела цикла

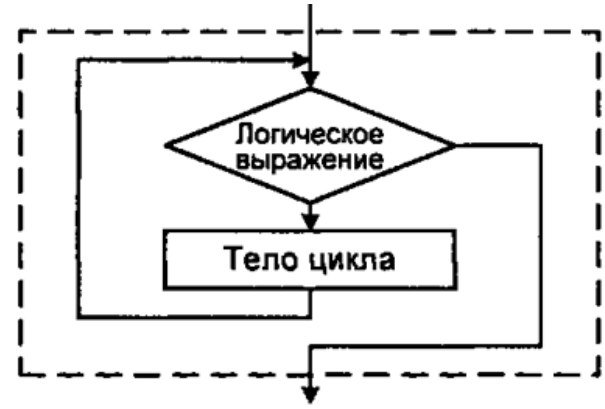


Рисунок 3.3 – Циклическая структура с проверкой до тела цикла

3.3.2.1 С-51

```
void TeloCikla(void)             // Определение подпрограммы
{
…                                // Это подпрограмма-заглушка
}
…                                // Главная программа
while(KnNaj=1)                   // Условное логическое выражение
TeloCikla():;                    // Реализация тела цикла
```

3.3.2.2 ASM-51

```
...                              ; Главная программа
Nachalo:                         ; "Место" до тела цикла (рис. 3.3)
jb KnNaj, KonCikla               ; Проверка условия завершения цикла
Call TeloCikla                   ; Тело цикла
```

```
sjmp Nachalo                ; Возврат в "место" до тела цикла (рис. 3.3)
KonCikla:                   ; Конец цикла
…
TeloCikla:
…                           ; Это подпрограмма-заглушка
ret                         ; Возврат из подпрограммы
```

4 Программы многофайловые и многомодульные. Программа-монитор

Все то, что выделено желтым цветом, просто для ознакомления. Так сказать комментарии автора.

4.1 Многофайловые программы

Самым простым способом соединения нескольких файлов в одну программу является использование директивы (include) включения текстового файла.

При использовании директивы include в исходный текст программы добавляется содержимое включаемого файла, и только после этого производится трансляция исходного текста в исполняемый код программы. Иными словами, содержимое главного файла программы и включаемых в него файлов объединяются препроцессором транслятора во временном файле, и только после этого производится трансляция полученного временного файла в исполняемый код микроконтроллера.

В отдельные файлы выделяются, как правило, описания внутренних регистров микроконтроллера и переменных, связанных с выводами микросхемы микроконтроллера.

Рассмотрим примеры:

```
#include <global.h>     // Добавление файла определений переменных,
                        // связанных с выводами микросхемы
#include <reg51.h>      // Добавление файла описаний
                        // регистров специальных функций микроконтроллера
#include <IO.h>         // Добавление файла с подпрограммами, осуществляющими
                        // ввод и вывод данных в микросхему микроконтроллера
…                       // Остальная часть программы (C-51)

$include (stdio.asm)    ; Включение файла с функциями стандарт. ввода-вывода
$include (reg51.inc)    ; включение файла с описаниями
                        ; регистров специальных функций микроконтроллера
...                     ; остальная часть программы (ASM-51)
```

Содержимое файла IO.h является примером использования отдельного файла для хранения функций, осуществляющих ввод и вывод данных. Такое использование включаемых файлов позволяет разделить программу по функциям.

В файле REG51.h объявляются переменные, связанные с регистрами специального назначения микроконтроллера 89c51. Они должны использоваться в любой программе, работающей с микроконтроллером 89c51.

4.2 Многомодульные программы

Часть программы, которая может быть отдельно оттранслирована, называется программным модулем. Оттранслированный программный модуль сохраняется в виде отдельного файла в объектном формате, где, кроме машинных команд, сохраняется информация об именах переменных, адресах команд, требующих модификации при

объединении модулей в единую программу, и отладочная информация. Одновременно с объектным файлом может создаваться листинг этого модуля.

Использование нескольких модулей при разработке программы увеличивает скорость трансляции и, в конечном итоге, скорость написания программы. Однако объявления переменных и имен подпрограмм внешних модулей загромождают исходный текст модуля. Кроме того, при использовании чужих модулей трудно объявить переменные и подпрограммы без ошибок, поэтому объявления переменных, констант и предварительные объявления подпрограмм хранят во включаемых файлах, которые называются файлами-заголовками. Правилом хорошего тона считается при разработке программного модуля сразу же написать для него файл-заголовок, который может быть использован программистами, работающими с вашим программным модулем.

Из всего рассмотренного ранее понятно, что большую программу можно разделить на части. Остается открытым вопрос – как разделять единую программу на части? А с помощью подпрограмм.

Подпрограммы обычно выносят в отдельные модули, при этом стараются собрать в одном модуле подпрограммы, решающие подобные задачи.

- Использование нескольких файлов позволяет разбить исходный текст программы на несколько частей, каждая из которых реализует свою независимую задачу.
- Удобнее всего в отдельные файлы выносить подпрограммы, которые должны быть построены таким образом ,чтобы их связь с основной программой была минимальной.
- Разбираться с короткими файлами ,реализующими одну или несколько связанных между собой функций ,намного легче, чем работать с одним большим файлом;
- Различные части программы могут быть написаны разными программистами, которым намного легче работать со своей программой .оформленной в виде отдельного файла.

Нерешенные проблемы

1. Программа-транслятор работает со всем исходным текстом целиком, ведь она соединяет все файлы перед трансляцией вместе. Поэтому время трансляции исходного текста программы получается значительным. В то же самое время программа никогда не переписывается целиком. Обычно изменяется только небольшой участок программы. В связи с этим значительные затраты времени на компиляцию представляются необоснованно большими;
2. Максимальное количество имен переменных и меток бывает ограничено программой-транслятором и может быть исчерпано при написании исходного текста программы, если он достаточно велик;
3. Различные программисты, участвующие в создании программного продукта, могут назначать одинаковые имена для своих переменных и при попытке соединения таких файлов в единую программу обычно возникают проблемы.

4.3 Программа-монитор

Разработка программ для микропроцессоров происходит во многом иначе, чем для универсального компьютера. При выполнении программы на универсальном компьютере ее запуск, взаимодействие с внутренними и внешними устройствами или человеком берет на себя операционная система. Программа, написанная для микроконтроллера, должна решать все эти задачи самостоятельно. Программа универсального компьютера когда-нибудь запускается и завершается. Программа, управляющая микроконтроллером, запускается при включении устройства и не завершает свою работу, пока не будет выключено питание.

Рассмотрим программу для микроконтроллера, которая будет выполнять поставленные задачи. Назовем эту программу «монитор», поскольку она «наблюдает» за использованием ресурсов системы.

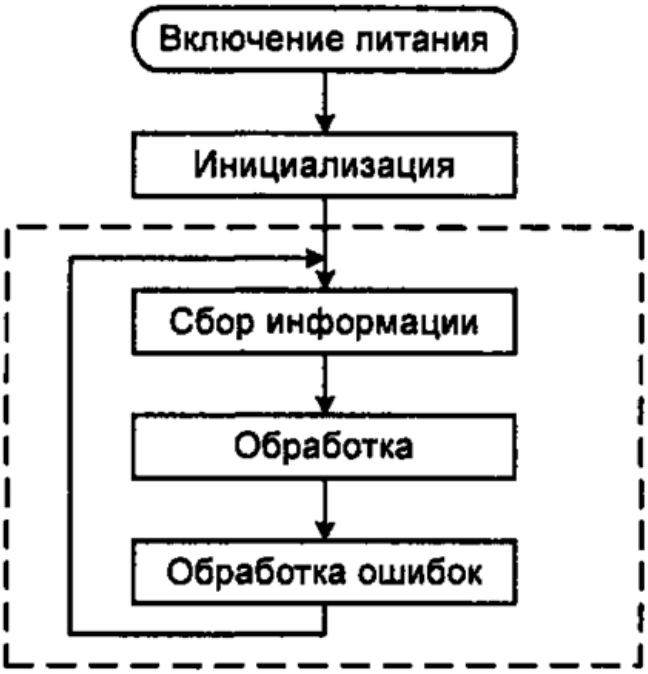


Рисунок 4.1 – Схема программы-монитор

После включения питания эта программа должна настроить микросхему микроконтроллера для выполняемой разрабатываемым устройством задачи. Для этого она должна запрограммировать определенные выводы микросхемы микроконтроллера на ввод или вывод информации, включить и настроить внутренние таймеры и т. д. Этот блок алгоритма программы-монитора называется инициализацией процессора. Инициализация микроконтроллера выполняется только один раз. Повторно она может потребоваться только при сбоях в работе микроконтроллера. Устранение таких сбоев производится аппаратным сбросом микроконтроллера.

Основная часть программы, реализующая алгоритм работы устройства, начинает выполняться после инициализации микроконтроллера. При этом необходимо понимать, что если в устройствах, не содержащих программно-управляемых компонентов, ввод, обработка и вывод информации производятся различными аппаратными блоками, то при выполнении программы эти же действия производятся последовательно одним и тем же

устройством — микропроцессором. Для выполнения каждой задачи обычно пишется отдельная подпрограмма. То есть при программной реализации устройства подпрограмма выполняет те же функции, что и отдельный блок при схемотехнической реализации устройства.

Для того чтобы работали все написанные подпрограммы, они включаются в один бесконечный цикл. Это эквивалентно периодическому запуску аппаратных блоков. Соединению между блоками соответствует взаимодействие частей программы, осуществляемое при помощи глобальных переменных. В этом же цикле обычно предусматривается блок обработки ошибок. Его предназначение сообщать оператору (пользователю) о непредвиденной ситуации, такой как неправильный ввод с клавиатуры или неправильные данные, полученные от подключенного к микроконтроллеру устройства.

```
void Init (void)                    // Подпрограмма инициализации микроконтроллера
{
/* По мере написания программы здесь будут добавляться операторы, настраивающие
элементы внутренней структуры микроконтроллера на необходимый режим работы*/
}
void SborInf (void)                 // Подпрограмма опроса клавиатуры
{}
void ObrabInf (void)                // Подпрограмма обработка информации
{
// Здесь обычно осуществляется переключение режимов работы устройства
}
void Obraboshib (void)              // Подпрограмма обработка ошибок
{
//Здесь  обычно осуществляется индикация ошибочного ввода информации с клавиатуры
}
void main (void)                    // Основная программа
{
Init();                             // Вызов подпрограммы инициализации МК
while (1)                           // Бесконечный цикл работы МК
{
Sborinf ();                         // Вызов подпрограммы
ObrabInf ();                        // Вызов подпрограммы
ObratOshib ();                      // Вызов подпрограммы
}                                   // Конец бесконечного цикла while (1)
}
```

5 Программирование на языке ASM-51: Порядок создания программы для микроконтроллера. Символы языка

5.1 Порядок создания

В большинстве случаев, независимо от выбранного языка программирования, предлагается следующая методика разработки программ (с учетом специфики языка ассемблер):

1. Этап постановки и формулировки задачи:

- изучение предметной области и сбор материала в проблемно-ориентированном контексте;
- определение назначения программы, выработка требований к ней представление требований в формализованном виде;
- формулирование требований к представлению исходных данных и выходных результатов;
- определение структуры входных и выходных данных;
- формирование ограничений и допущений на исходные и выходные данные.

2. Этап проектирования:

- формирование «ассемблерной» модели задачи;
- выбор метода реализации задачи;
- разработка алгоритма реализации задачи;
- разработка структуры программы в соответствии с выбранной моделью памяти.

3. Этап кодирования:

- уточнение структуры входных и выходных данных и определение ассемблерного формата их представления;
- программирование задачи;
- комментирование текста программы и составление предварительного описания программы.

4. Этап отладки и тестирования:

- составление тестов для проверки правильности работы программы;
- обнаружение, локализация и устранение ошибок в программе, выявленных в тестах;
- корректировка кода программы и ее описания.

5. Этап эксплуатации и сопровождения.

Традиционно у существующих реализаций ассемблера нет интегрированной среды, подобной интегрированным средам *Turbo Pascal*, *Turbo C* или *Visual C++*,

поэтому для выполнения всех функций по вводу кода программы, ее трансляции, редактированию и отладке необходимо использовать отдельные служебные программы. Большая часть их входит в состав специализированных пакетов ассемблера.

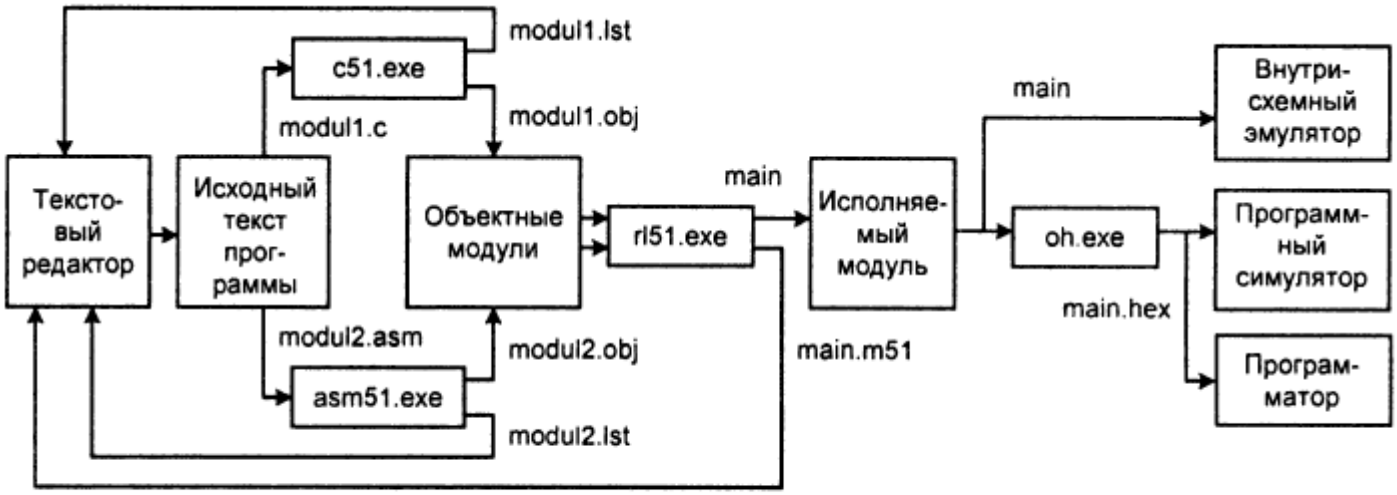


Рис. 8.1. Схема процесса написания программы на языке программирования ASM-51

Объектный модуль - набор машинных кодов и таблиц, адресов, переменных и признаков отладочных процессов

rl51.exe – линкёр/редактор связей – все метки, переменные в адреса

oh.exe превращает исполняемый модуль в загружаемый, создаёт служебные команды для программатора.

На *первом шаге* с использованием любого текстового редактора вводится код программы, т.е. создается исходный файл. Основным требованием к текстовому редактору является то, чтобы он не вставлял посторонних символов (спецсимволов редактирования). Файл должен иметь расширение ***.asm*** (для ассемблера) и ***.c*** (для С-51).

Второй шаг – трансляция программы. На этом шаге формируется объектный модуль, который включает в себя представление исходной программы в машинных кодах и некоторую другую информацию, необходимую для отладки и компоновки его с другими модулями.

Третий шаг разработки программы - создание исполняемого (загрузочного) модуля или компоновка программы. Главная цель этого шага - преобразовать код и данные в одном или нескольких объектных файлах (исходные модули могут быть написаны на одном или нескольких языках) в их перемещаемое выполняемое изображение. Результатом работы компоновщика является создание загрузочного файла с расширением ***exe***. После этого операционная система может загрузить такой файл в память и выполнит его.

Получение исполняемого модуля, т.е. фактическое устранение синтаксических ошибок, еще не гарантирует того, что программа будет хотя бы запускаться, не говоря уже о правильности работы. Поэтому обязательным этапом процесса разработки является отладка. На этом этапе выполняется контроль правильности функционирования как отдельных участков кода, так и всей программы в целом.

Структура программы в целом такова, что слева отводится место для поля метки. Метка записывается английскими буквами и арабскими цифрами, начинается метка с буквы. признаком метки является знак "**:**", которое следует за последним символом метки, пробел при этом не допускается. Метка может относиться к пустой строке, тогда следующая непустая строка будет привязана к метке.

M1:
M2:
Primer: MOV A, R0

Метка может относится и к присвоению переменных, это нужно, когда программа на языке С включает в себя подпрограмму на языке ASM.

Поле операций: мнемоническое обозначение команды и 1, 2 или 3 операнда.

После метки в этой же строке может следовать символ "**;**", отделяющий содержимое комментария. Комментарии не переводятся в машинный код, а хранятся в исходном тексте. Длина текста программы ASM ограничена транслятором. Транслятор должен перевести текст программы в машинный код, составить таблицу меток, таблицу адресов ссылок, заменить метки на эти адреса во всем тексте, проверить корректность перехода по меткам, а также корректность условий перехода.

5.2 Символы, используемые в языке Assembler

Символы исходной программы представляют собой подмножество таблиц символов ASCII для DOS и ANSI для WINDOWS. В исходном тексте программы, написанном на языке программирования PL/M-51 допустимо использование следующих символов:

- буквы,
- знаки
- цифры.

В качестве *букв* воспринимаются латинские буквы верхнего и нижнего регистра:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z.

Ниже приведен перечень *цифр*:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Наименования знаков и их обозначение приведено в таблице 1:

Наименование	Обозначение
Номер	#
Знак денежной единицы	$
Апостроф	'

круглая скобка левая	(
круглая скобка правая	)
Звездочка	*
Плюс	+
Запятая	,
Минус	-
Точка	.
дробная черта	/
Двоеточие	:
Точка с запятой	;
Меньше	<
Равно	=
больше	>
вопросительный знак	?
коммерческое эт	@

Знаки, комбинации знаков (<>, >=, <=), а также символы интервала являются разделителями конструкций языка. До и после знака - разделителя в любой конструкции языка могут быть вставлены символы интервала.

ASCII символы, не входящие в перечень основных символов алфавита языка, считаются дополнительными. Эти символы могут использоваться для пояснений в исходном тексте программы, а также для определения символьных констант.

Из символов формируются идентификаторы и числа.

6 Идентификаторы в языке ASM-51: ключевые слова

Идентификатор это символическое обозначение объекта программы. В качестве идентификатора может быть использована любая последовательность букв и цифр. При этом в качестве буквы может быть использована любая буква латинского алфавита, а также вопросительный знак (?) и знак "нижнее подчеркивание" (_). Идентификатор может начинаться только с буквы! Это позволяет отличать его от числа. В идентификаторах, язык программирования ASM-51 различает буквы верхнего и нижнего регистров.

Количество символов в идентификаторе ограничено длиной строки (255 символов). Транслятор различает идентификаторы по первым 31 символам.

Примеры идентификаторов: ADD5, FFFFH, ?, ALPHA_1.

В языке программирования ASM-51 имеются три категории идентификаторов:

- ключевые слова;
- встроенные имена;
- определяемые имена.

Ключевое слово является определяющей частью оператора языка ассемблера. Значения ключевых слов языка ассемблера ASM-51 не могут быть изменены или переопределены в программном модуле каким-либо образом. Ключевому слову не может быть назначено имя- синоним. Ключевые слова могут быть написаны буквами как верхнего, так и нижнего регистров. То есть ключевое слово MOV и ключевое слово mov полностью эквивалентны.

В языке ASM-51 имеются следующие категории ключевых слов:

- инструкции;
- директивы;
- вспомогательные слова;
- операции.

Инструкции по форме записи совпадают с мнемоническими обозначениями команд микроконтроллеров семейства MCS-51 и совместно с операндами, составляют команды микроконтроллера. Список инструкций:

ACALL (вызов подпрограммы), ADD(арифметическое сложение), ADDC (складывает содержимое аккумулятора A с содержимым байта-источника), AJMP (безусловный переход), ANL (логическое ИЛИ), CALL(вызов), CJNE (сравнить и перейти, если не равны первый и второй операнд), CLR (сбросить битовый или байтовый аккумулятор), CPL (инверсия бита), DA (десятичная коррекция двоично-десятичных чисел), DEC (уменьшить на 1 целочисленное), DIV (арифм. деление), DJNZ (вычитание 1 из указанной во втором байте команды ячейки памяти и переход к вычисляемому по третьему байту команды адресу, если результат вычитания не равен 0), INC (целочисленное увеличение на 1), JB (перейти, если установлен бит на линии), JBC (если бит установлен, перейти и сбросить), JC (Если бит переноса равен единице, то производится переход к вычисляемому по второму байту команды адресу. В противном случае выполняется следующая команда), JMP (складывает 8- битовое содержимое

аккумулятора без учета знака с 16- битовым указателем данных (DPTR) и загружает полученный результат в счетчик команд), JNB (Если указанный во втором байте команды бит равен нолю, то производится переход к вычисляемому по третьему байту команды адресу), JNC (Если бит переноса равен нолю, то производится переход к вычисляемому по второму байту команды адресу. В противном случае выполняется следующая команда.), JNZ (перейти если не 0), JZ (перейти если 0), LCALL (вызов распростр. на все строки программы), LJMP (длинный переход, на 2^16 адресов) , MOV (переменная, указанная во втором операнде, копируется в ячейку, указываемую первым операндом), MOVC (загружает аккумулятор константой из памяти программ.), MOVX (пересылает данные между аккумулятором и байтом внешней памяти.), MUL (перемножает целые восьмибитовые беззнаковые числа, хранящиеся в аккумуляторе и регистре), NOP (Кроме программного счетчика не изменяет ни одного регистра, на флаги не воздействует.), ORL (выполняет операцию побитового "логического ИЛИ" между указанными переменными), POP (Считывает содержимое ячейки внутренней памяти, на которую указывает регистр указателя стека, после этого содержимое указателя стека уменьшается на 1), PUSH (Содержимое указателя стека увеличивается на 1), RET (возврат из подпрограммы, последовательно выгружает старший и младший байты счетчика команд из стека, уменьшая содержимое указателя стека на 2), RETI (возврат из подпрограммы обслуживания прерывания.), RL (сдвиг содержимого аккумулятора влево), RLC (сдвиг содержимого аккумулятора влево через флаг переноса), RR (сдвиг содержимого аккумулятора вправо), RRC (сдвиг содержимого аккумулятора вправо через флаг переноса) SETB (установить бит), SJMP (команда выполняет безусловный относительный короткий переход по указанному адресу), SUBB (команда вычитает указанную переменную вместе с флагом переноса из содержимого аккумулятора, результат помещается в аккумулятор), SWAP (команда осуществляет обмен между младшей (биты 0..3) и старшей (биты 4..7) тетрадами аккумулятора), XCH (команда загружает аккумулятор содержимым указанной переменной, в то же самое время первоначальное содержимое аккумулятора заносится по указанному адресу), XCHD (команда осуществляет обмен между младшей (биты 0..3) тетрадой (тетрада это четырехбитная переменная) аккумулятора, где обычно хранится двоично-десятичная цифра с тетрадой ячейки внутреннего ОЗУ), XRL (выполняет операцию побитового "исключающее логического ИЛИ" между указанными переменными. Результат сохраняется в приемнике).

Директивы совместно с вспомогательными словами определяют действия в программе, которые должны быть выполнены ассемблером в процессе преобразования исходного текста программы в объектный код. В языке программирования ASM-51 используются:

Директивы: BIT, BSEG(позволяет определить абсолютный сегмент во внутренней памяти данных с битовой адресацией по определённому адресу. Эта директива не назначает имени сегменту, то есть объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут AT не используется, то начальный адрес сегмента

предполагается равным нулю. Использование битовых переменных позволяет значительно экономить внутреннюю память программ микроконтроллера.), CODE, CSEG(позволяет определить абсолютный сегмент в памяти программ по определенному адресу. Эта директива не назначает имени сегменту, то есть объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут AT не используется, то начальный адрес сегмента предполагается равным нулю.), DATA, DB(используется для занесения в память программ однобайтных констант), DBIT(битовые переменные в памяти данных могут быть назначены при помощи этой директивы резервирования битов), DS(байтовые переменные в памяти данных могут быть назначены при помощи этой директивы резервирования памяти), DSEG(позволяет определить абсолютный сегмент во внутренней памяти данных по определённому адресу. Предполагается, что к этому сегменту будут обращаться команды с прямой адресацией. Эта директива не назначает имени сегменту, то есть объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут AT не используется, то начальный адрес сегмента предполагается равным нулю.), DW(позволяет заносить в память программ двухбайтные числа), END, EQU(позволяет назначать имена переменных и констант. Теперь можно назначить переменной адрес в одном месте и пользоваться идентификатором переменной во всей программе), EXTRN, IDATA, ISEG(позволяет определить абсолютный сегмент во внутренней памяти данных по определённому адресу. Напомню, что внутренняя память с косвенной адресацией в два раза больше памяти с прямой адресацией. Эта директива не назначает имени сегменту, то есть объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут AT не используется, то начальный адрес сегмента предполагается равным нулю.), NAME, ORG(предназначена для записи в счетчик адреса сегмента значения своего операнда. То есть при помощи этой директивы можно разместить команду (или данные) в памяти микроконтроллера по любому адресу.), PUBLIC(Для того, чтобы редактор связей мог осуществить связывание модулей в единую программу, переменные и метки, объявленные по крайней мере в одном из модулей как EXTRN, в другом модуле должны быть объявлены как доступные для всех модулей при помощи директивы PUBLIC . Пример использования директивы PUBLIC на языке программирования ASM-51:

```
PUBLIC BufInd, Parametr
PUBLIC Podprogr, ?Podprogr?Byte
```

), RSEG(После определения имени сегмента можно использовать этот сегмент при помощи директивы rseg. Использование сегмента зависит от области памяти, для которой он предназначен. Если это память данных, то в сегменте объявляются байтовые или битовые переменные. Если это память программ, то в сегменте размещаются константы или участки кода программы.), SEGMENT(позволяет определить имя сегмента и область памяти, где будет размещаться данный сегмент памяти. Для каждой области памяти определено ключевое слово:

- data - размещает сегмент во внутренней памяти данных с прямой адресацией;
- idata - размещает сегмент во внутренней памяти данных с косвенной адресацией;
- bit - размещает сегмент во внутренней памяти данных с битовой адресацией;
- xdata - размещает сегмент во внешней памяти данных;
- code - размещает сегмент в памяти программ;

), SET(Если требуется в различных местах программы назначать одному и тому же идентификатору различные числа, то нужно пользоваться директивой set.), USING(При использовании прерываний критичным является время, занимаемое программой, обработчиком прерываний. Это время можно значительно сократить, выделив для обработки прерываний отдельный банк регистров. Выделить отдельный банк регистров можно при помощи директивы USING. Номер банка используемых регистров указывается в директиве в качестве операнда.), XDATA, XSEG(позволяет определить абсолютный сегмент во внешней памяти данных по определённому адресу. Эта директива не назначает имени сегменту, то есть объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут AT не используется, то начальный адрес сегмента предполагается равным нулю.).

Вспомогательные слова: AT, BIT(размещает сегмент во внутренней памяти данных с битовой адресацией), BITADDRESSABLE, CODE(размещает сегмент в памяти программ), DATA(размещает сегмент во внутренней памяти данных с прямой адресацией), IDATA(размещает сегмент во внутренней памяти данных с косвенной адресацией), INBLOCK, INPAGE, NUMBER, PAGE(начало сегмента с адреса, кратного 256), UNIT, XDATA(размещает сегмент во внешней памяти данных).

Операции выполняются ассемблером в процессе вычисления выражений на этапе трансляции исходного текста программы для определения конкретного числа, которое используется в команде. Перечень операций, использующихся языком программирования ASM-51: AND(логическое И), EQ, GE, GT, HIGH, LE, LOW, LT, MOD(вычисление остатка от целочисленного деления), NE, NOT(побитовая инверсия операнда), OR(логическое ИЛИ), SHL, SHR, XOR("исключающее или").

7 Идентификаторы в языке ASM-51: встроенные и определяемые имена

7.1 Встроенные имена

Имя	Регистр
A	Аккумулятор
R0-R7	8-разрядный рабочий регистр текущего банка рабочих регистров
AR0-AR7	Адреса 8-разрядных рабочих регистров текущего банка рабочих регистров
DPTR	16-разрядный регистр-указатель данных
PC	16-разрядный счетчик команд
C	Флаг переноса
AB	Регистровая пара, состоящая из аккумулятора A (старшая часть) и регистра B (младшая часть)

Ниже представлены пояснения от Федотова Ю.А.

Все идентификаторы используемые здесь не должны использоваться в качестве имен переменных задаваемых пользователем.

Для 16-разрядных регистров операции HIGH и LOW нужны для того чтобы можно было выделить старший или младший байт.

В командных обращениях к командному счетчику (PC) не предусмотрено. Т.е. таких команд обращений на прямую к программному счетчику (командному счетчику (PC)) нет. Поэтому можно только при необходимости считать адрес, но тогда надо будет сначала скопировать (MOV).

А вот DPTR используется очень часто, поскольку, для того чтобы задать адрес внешней памяти, надо будет выйти за пределы 8-разрядного 2-ого слова. В самой команде надо писать MOV DPTR,#число. Сам транслятор преобразовав это 10-ое число запишет старший байт в старшую часть, а младший байт в младшую часть этого указателя данных (DPTR).

7.2 Определяемые имена

Определяемые имена – имена, которые задает пользователь:

- метки;
- внутренние и внешние переменные адресного типа;
- внутренние и внешние переменные числового типа;
- имена сегментов;
- названия программных модулей.

8 Программирование на языке ASM-51: числа и литеральные строки

В языке программирования ASM-51 используются целые беззнаковые числа, представленные в двоичной, восьмеричной, десятичной и шестнадцатеричной формах записи. Для определения основания системы счисления используется суффикс (буква, следующая за числом):

- B – двоичное число (0, 1);
- Q – восьмеричное число (0, 1, 2, 3, 4, 5, 6, 7);
- D – десятичное число (0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
- H – шестнадцатеричное число (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

Для десятичного числа суффикс может отсутствовать. Количество символов в числе ограничено размером строки, однако значение числа определяется по модулю 2^16 (т. е. диапазон значений числа находится в пределах от 0 до 65535).

Примеры записи чисел:

011101b, 1011100B, 735Q, 456o, 256, 0fah, 0CBH.

Число всегда начинается с цифры. Это необходимо для того, чтобы отличать шестнадцатеричное число от идентификатора. Например:

ADCH — идентификатор; 0ADCH — число.

Часто бывает удобно выполнить некоторые вычисления для того, чтобы получить число. При этом, если поместить в текст программы предварительно вычисленное на калькуляторе значение, то может возникнуть вопрос: откуда взялось это значение. Лучше ввести формулу расчета и сами значения непосредственно в исходный текст программы. Язык программирования ASM-51 позволяет выполнять беззнаковые операции над числами. В таких выражениях допустимо использовать следующие арифметические операции: "+" суммирование, "-" вычитание, "*" умножение, "/" деление, "mod" вычисление остатка от целочисленного деления.

В языке программирования ASM-51 также определена одноместная операция изменения знака — «минус» (-). Для нее требуется только один операнд — тот, которому она предшествует.

Часто требуется выполнять операции в определенном порядке, отличающемся от принятого по умолчанию. Для изменения порядка выполнения операций можно воспользоваться скобками. Более того, использование скобок в ряде случаев повышает наглядность программы и, тем самым, уменьшает время ее отладки.

Кроме арифметических операций в выражениях допустимо использование следующих логических операций: "not" побитовая инверсия операнда, "and" логическое «И», "or" логическое «ИЛИ», "xor" «исключающее ИЛИ» (суммирование по модулю два), функции выделения старшего «HIGH» и младшего «LOW» байта 16-разрядного числа.

Пример использования выражений языка программирования ASM-51 для определения числовой константы приведен ниже.

```
;--------------Настройка  таймер T0-------------------------------------

Int T0:

mov TMOD, #00000001b                ; Настройка таймера на 1 режим работы
mov TH0, #HIGH(-(F_ZQ/12)*10-2)     ; Настроить таймер
mov TL0, #LOW(-(F_ZQ/12)*10-2)      ; на период 10мс
reti
```

, где LOW - выделение младшего байта из числа в скобках, HIGH - выделение старшего байта из числа в скобках, F_ZQ - константа 12 МГц.

Часто операнд используется для представления символов на экране дисплея. В этом случае для определения его значения удобнее воспользоваться не числом, а литеральной константой.

Литеральная константа заключается в апострофы: 'a', 'W'.

```
mov SBUF, #'Б'   ;Передать по последовательному порту ANSI код буквы 'Б'
```

Часто на экране дисплея приходится отображать не одну букву, а целые фразы. Их удобно запоминать в памяти программ, а затем передавать на дисплей при помощи специальной подпрограммы. Для записи фраз в памяти программ можно воспользоваться **литеральными строками**, для ввода которых в память программ удобно воспользоваться директивой db:

```
Nadp: DB 'Ошибка в блоке 5'
```

В этом случае каждый символ строки заменяется отдельным байтом и запоминается в ПЗУ памяти программ. Начало строки обязательно помечается при помощи метки. Для увеличения наглядности программы следует содержание надписи отобразить в имени использованной метки.

9 Директивы языка программирования ASM-51: equ, set, db, dw

Директивы – это особые инструкции, которые записаны в тексте программы и выполняются до трансляции программы. Они используются для указания компилятору некоторых особенностей обработки кода при компиляции.

Директива **equ** позволяет назначать имена для констант и значения адресов для переменных. При использовании этой директивы можно назначить идентификатору переменной адрес в одном месте программы и пользоваться идентификатором этой переменной во всей программе. **Пример:**

```
FuncSet equ 0x20
_8bit equ 0x10
DispDat equ P0                      ; Шина данных ЖКИ
RDS equ P1_2                        ; Чтение команды ЖКИ
RW equ P1_1                         ; Сигнал выбора записи/чтения
E equ P1_0                          ; Строб синхронизации ЖК
mov DispDat, #(FuncSet or _8bit)    ; Выставить на шине данных
setb E                              ; команду установки функции
clr E                               ; и выдать стробирующий сигнал
```

Директива **set**. Если требуется в различных местах программы назначать одному и тому же идентификатору различные значения, то нужно пользоваться директивой **set**. **Она используется точно так же, как директива equ, поэтому иллюстрировать это примером не будем.** Константы, назначаемые директивами **equ** и **set**, могут быть использованы только в качестве операнд команд микроконтроллера. В то же время достаточно часто требуется работа с таблицей констант, такой как таблица перекодировки, таблицы элементарных функций или синдромы помехоустойчивых кодов. Такие константы используются не на этапе трансляции для формирования машинного кода инструкций, а при исполнении программы. Они заносятся в память программ микроконтроллера. Для помещения значений констант в память программ микроконтроллера используются директивы **db** и **dw** (direct byte/word).

Директива **db** используется для занесения в память микроконтроллера однобайтных констант. **Пример:**

```
Decod:
        mov  DPTR, #TabDecod
        movc A,@A+DPTR
        ret
TabDecod: ;-abcdefg
        DB 01111110b    ;символ “0”
        DB 00110000b    ;символ “1”
```

Расположение сегментов в семисегментном индикаторе

```
DB 01101101b    ;символ “2”
DB 01111001b    ;символ “3”          a
DB 00110011b    ;символ “4”         ---
DB 01011011b    ;символ “5”       f|   |b
DB 01011111b    ;символ “6”         ---g
DB 01110000b    ;символ “7”       e|   |c
DB 01111111b    ;символ “8”         ---
DB 01111011b    ;символ “9”          d
```

В директиве **db** можно задавать сразу несколько констант, разделенных запятыми. Можно одновременно использовать все системы счисления, но обычно имеет смысл снабдить каждую константу комментарием, как это сделано в предыдущем примере (картинка выше). Так программа становится более понятной и её легче отлаживать.

Эта же директива позволяет легко размещать в памяти строки, которые в дальнейшем потребуется высвечивать на встроенном дисплее или экране дисплея универсального компьютера, подключенного к разрабатываемому устройству через какой-либо интерфейс.

Пример использования директивы **db** для занесения строк в память программ микроконтроллера:

```
mov R7,#(EndNadp-NadpSvjazUst)   ; занести количество символов в строке
mov DPTR,#NadpSvjazUst           ; подготовиться к передаче первого символа

PrdSledSmv:
clr A
movc A,@A+DPTR                   ; считать очередной символ надписи
inc DPTR

call PrdSmv                      ; передать очередной символ надписи, и
djnz R7,PrdSmv                   ; если это был последний символ надписи,
ret                              ; выйти из подпрограммы

NadpSvjazUst: db “связь установлена”,10,13
EndNadp:
```

Директива **dw** позволяет заносить в память программ двухбайтные числа. В ней, как и директиве db, можно записывать несколько чисел, разделенных запятой. **Пример:**

```
0016  0001        264        dw 1,2, 0abh, “a”, ”QW”
```

```
0018  0002
001A 00AB
001C 0061
001E 5157

LOC  OBJ     LINE     SOURCE

0000 78FF      1      MOV R0,#255         ;обнулить 255 ячеек памяти
0002  E4       2      clr A
0003           3      ClrOZU:
0003 F6        4      mov @R0,A    ;обнулить очередную ячейку памяти
0004 D8FD      5      djnz R0,ClrZOU          ;если все ячейки обнулены
               6                    ;то продолжить выполнение программы
```

10 Директивы языка программирования ASM-51: org, using; управляющие команды: debug, include

Директива **ORG** предназначена для записи в счетчик адреса сегмента значения своего операнда. То есть при помощи этой директивы можно поместить команду (или данные) в памяти микроконтроллера по произвольному адресу.

Пример:

```
reset:
        LJMP main
;Перезагрузка таймера-----------------------------------------------------
        ORG 0bh ;Вектор прерывания от таймера
        LGMP IntT0
;Перезагрузка таймера -----------------------------------------------------
        ORG 23h ;Вектор прерывания последовательного порта
        LGMP IntSPORT
IntT0:
        mov TL0, #LOW(-(F_ZQ/12)*10-2)   ;Настроить таймер
        mov TH0, #HIGH(-(F_ZQ/12)*10-2)  ;на период 10мс
        reti
;Начало основной программы микроконтроллера----------------------
main:
        mov SP,#VershSteka ;настроить указатель стека на вершину стека
        call init                  ;настроить микроконтроллер
;-------------------------------------------------------------------------------
```

Необходимо отметить, что при использовании этой директивы возможна ситуация, когда программист приказывает транслятору разместить новый код программы по уже написанному месту, поэтому использование этой директивы допустимо только в крайних случаях. Обычно это использование векторов прерываний **(Вектор прерываний** – адрес процедуры обработки **прерываний**.Адреса размещаются в специальной области памяти доступной для всех подпрограмм. **Вектор прерывания** содержит 4 байта: старшее слово содержит сегментную составляющую адреса процедуры обработки исключения, младшее — смещение.**)**

Директива **USING** (указывает, что же будет использоваться). При использовании прерываний критичным является время, занимаемое программой, обработчиком прерываний. Это время можно значительно сократить, выделив для обработки прерываний отдельный **банк регистров.** Выделить отдельный банк регистров можно при помощи директивы USING. Номер банка используемых регистров указывается в директиве в качестве операнда. При этом реальное включение банка регистров производится записью необходимой константы в регистре PSW(набор битов).

Банк регистров:

Несмотря на то, что это самое маленькое адресное пространство из рассматриваемых, оно устроено наиболее сложным образом.

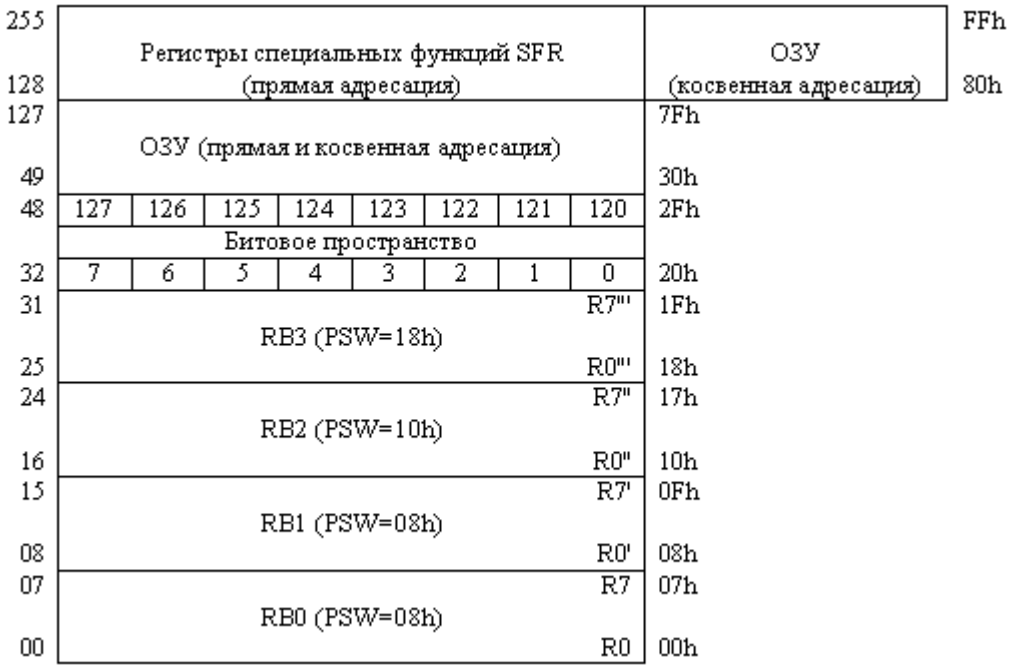


Рисунок 1-Адресное пространство внутренней памяти данных.

Внутреннее ОЗУ данных предназначено для временного хранения информации, используемой в процессе выполнения программы, и занимает 128 младших байт, с адресами от 000h до 07Fh для микроконтроллеров 8051, 8031, КР1816ВЕ31, КР1816ВЕ51, КР1816ВЕ751 КР1830ВЕ31, КР1830ВЕ51, КР1830ВЕ751 или 256 восьмиразрядных ячеек, с адресами от 000h до 0FFh для всех остальных микроконтроллеров семейства.

Регистры специальных функций занимают адреса внутренней памяти данных с 080h по 0FFh. Так как адреса регистров специальных функций совпадают со старшими адресами внутреннего ОЗУ данных, то имеются особенности при использовании этих адресов внутренней памяти данных.

Система команд микроконтроллера позволяет обращаться к ячейкам внутренней памяти данных при помощи прямой и косвенно-регистровой адресации. При обращении к ячейкам памяти с адресами 0-127 использование любого из этих видов адресации будет производить выборку одной и той же ячейки памяти.(В ячейках памяти с адресами с 32-47 может осуществляться чтение и запись, как байтовое, так и битовое). При обращении к ячейкам ОЗУ с адресами 128-256 следует воспользоваться косвенно-регистровой адресацией. Учитывая, что работа со стеком ведётся при помощи косвенной адресации, то имеет смысл размещать в этой области памяти **стек**(нужен для того, чтобы в нем хранить что то временно).Ячейки памяти, в которых хранятся адреса возврата из подпрограмм называются **стеком. Стек**-это особая память(организован как “последний вошел первый вышел”). Если же требуется обратиться к регистрам специальных функций, то нужно использовать прямую адресацию.

Например:

```
MOV A, 80h   ;Скопировать сигналы с внешних ножек порта P0 в аккумулятор
```

```
MOV R0, #80h ;Скопировать в аккумулятор содержимое
MOV A, @R0   ;ячейки внутреннего ОЗУ  с адресом 80h
```

(Прямая битовая адресация используется для обращения к отдельно адресуемым 128 битам, расположенным в ячейках с адресами 20H-2FH, и к отдельно адресуемым битам регистров специального назначения. **Например**:

```
D220    SETB 20h ;использована прямая битовая адресация
C215    CLR  15h ;использована прямая битовая адресация
```

Косвенно-регистровая адресация используется для обращения к ячейкам внутреннего ОЗУ данных. В качестве регистров-указателей адреса используются регистры R0, R1 выбранного банка регистров. **Например**:

```
E6    MOV A,@R0 ;
F7    MOV @R1,A ;)
```

Регистры общего назначения позволяют писать самые эффективные программы. У микроконтроллеров семейства MCS-51 программирующему инженеру доступны восемь регистров. Более того, в этом семействе микроконтроллеров есть целых четыре набора (банка) регистров с именами RB0 - RB3. Банк регистров состоит из восьми восьмиразрядных регистров с именами R0, R1, …, R7. Несколько банков регистров служат для организации независимой работы нескольких параллельно выполняемых программ. Переключение банков регистров производится при помощи двух особых бит регистра слова состояния программы PSW (RS0 и RS1). Если организация нескольких параллельных потоков обработки данных не нужна, то можно пользоваться только нулевым банком регистров, включающимся автоматически после включения питания и сброса микроконтроллера, остальные ячейки памяти использовать как обычное ОЗУ.

Все четыре банка регистров объединены с 32 младшими байтами внутреннего ОЗУ данных (см. рисунок 1). Так как физически регистры и ячейки внутреннего ОЗУ объединены, то команды программы могут обращаться к регистрам, используя их имена R0-R7 **(регистровая адресация)**:

```
MOV A, R0   ;Скопировать содержимое регистра R0 в аккумулятор
MOV R7, A   ;Скопировать содержимое аккумулятора в регистр R7
```

или используя их адрес во внутренней памяти данных **(прямая байтовая адресация):**

```
MOV A, 0   ;Скопировать содержимое нулевой ячейки ОЗУ в аккумулятор
MOV 7, A   ;Скопировать содержимое аккумулятора в седьмую ячейку ОЗУ
```

Регистровая команда работает быстрее и она предпочтительнее.

Пример использования директивы USING для подпрограммы обслуживания прерываний от таймера 0:

```
_code segment code
        CSEG AT 0bh ;вектор прерываний от таймера 0
        jmp IntT0

        rseg _code
        USING 2
IntT0:
        push PSW ;сохранить содержимое слова состояния МК
        push ACC ;сохранить содержимое аккумулятора
        mov PSW, #00010000b ;включить второй банк регистров
        mov TL0, #LOW(-(F_ZQ/12)*10-2) ;настроить таймер
        mov TH0, # HIGH(-(F_ZQ/12)*10-2) ;на период 10мс
        pop ACC ;восстановить содержимое аккумулятора
        pop PSW ;восстановить содержимое слова состояния МК
        reti
```

PUSH-этой командой записываем в стек;

POP-сдвигается в другую сторону.

Управляющие команды (нужны для транслятора, что нужно делать с файлами) записываются со знака $, и должен стоять в крайней левой позиции.

Управляющие команды

Примеры управляющих строк:

```
$PRINT(A:\PROG.LIST) OBJECT(PROG.OBJ)
$LIST DEBUG XREF
```

Команда include

```
$INCLUDE (REG51.INC)
```

Команды list/nolist

```
$NOLIST   //Запретить создание листинга включаемого файла
$INCLUDE (REG51.INC)
$LIST  //Разрешить создание листинга дальнейшего текста
```

$PRINT-направить файл;

$INCLUDE-подключение к дальнейшему тексту;

$LIST-создать отладочный листинг для программного файла XREF;

$NOLIST-помогает управлять работой транслятора

Команда include Это, пожалуй, наиболее часто используемая команда языка программирования ASM-51. Она позволяет включать в состав программы участки текста из другого файла. Это удобно при многофайловом написании программы, например, для того, чтобы вынести описания внутренних регистров микроконтроллера в отдельный файл. Пример использования команды include для включения файла описания внутренних регистров микроконтроллера 89c51 выглядит следующим образом:

```
$INCLUDE (REG51.INC)
```

При использовании этой команды все содержимое включаемого файла помещается в выходной листинг программы, в результате чего становится трудно читать этот листинг. Поэтому в состав команд языка программирования ASM-51 включены команды **list/nolist**.

Команды list/nolist позволяют включать и выключать листинг исходного текста соответственно. При активной команде nolist в файл листинга будут помещаться только сообщения об ошибках. Пример запрета размещения содержимого включаемого файла в листинге программы будет выглядеть следующим образом:

```
$NOLIST //Запретить создание листинга включаемого файла

$INCLUDE (REG51.INC)

$LIST //Разрешить создание листинга дальнейшего текста
```

Команда debug/nodebug позволяет помещать в объектный модуль отладочную информацию (имена и местоположение переменных, меток и операторов) или запрещать размещение отладочной информации в объектном модуле.

11 Использование сегментов в языке программирования ASM-51: абсолютные сегменты

При трансляции программы по частям возникает вопрос, как с этими частями работать. Иначе говоря, дает о себе знать проблема сегментов.

Отметим, что даже когда мы не задумываемся о сегментах, в программе (в памяти программы) присутствуют два сегмента: сегмент кода и сегмент данных. Если внимательно присмотреться к программе, то можно обнаружить, что кроме кодов команд в памяти программ хранятся константы, т.е. в памяти программ микроконтроллера располагаются, по крайней мере, два сегмента: код и данные. Чередование кода и данных может привести к нежелательным последствиям. Вследствие каких-либо причин данные могут быть случайно выполнены в качестве машинных команд или наоборот коды машинных команд могут быть восприняты и обработаны как данные. Другой случай, в памяти данных, например, если отвести ячейки под СТЕК, определив дно СТЕКА с адреса какой-нибудь ячейки, и начать заполнять ячейки переменными, начиная с 0-ой и т.д до тех пор пока не дойдем до ячейки, отведенной под СТЕК, может получиться так, что следующая переменная запишется в ячейку для СТЕКА, для избежания подобного следует располагать дно СТЕКА на самый верхний незанятый адрес.

Может возникнуть ситуация, когда нужно использовать и байтовые переменные и битовые переменные, тогда байтовые переменные следует разместить в ячейках, начиная с 48 по 127, битовые - с 32 по 47 (рис.11.1).

Поэтому следует разделять на сегменты память данных и память программ.

Рисунок 11.1 – Внутренняя память данных MCS-51

Перечисленные выше причины приводят к тому, что желательно явным образом выделить, по крайней мере, четыре сегмента в памяти программ:

1) кода;

2) переменных;

3) стека;

4) констант.

И лучше, если эти сегменты будут перемещаемыми. Тогда редактор связей сможет автоматически скомпоновать программу наилучшим способом.

Пример размещения сегментов в адресном пространстве памяти программ и внутренней памяти данных приведен на рис. 11.2.

четыре сегмента:

1) кода;

2) переменных;

3) стека;

4) констант.

Рисунок 11.2 – Использование сегментов в ASM-51

На этом рисунке видно, что при использовании нескольких сегментов переменных во внутренней памяти данных редактор связей может разместить меньший из них на месте неиспользованных банков регистров. Под сегмент стека обычно отводится вся область внутренней памяти, не занятая переменными. Это позволяет создавать программы с максимальным уровнем вложенности подпрограмм. Сегмент переменных, расположенный на рис. 11.2 во внешней памяти данных, при использовании современных микросхем, таких как AduC842, может находиться и в ОЗУ, расположенном на кристалле микроконтроллера.

Наиболее простой способ определения сегментов — это использование абсолютных сегментов памяти. При этом способе распределение памяти ведется вручную точно так же, как это делалось при использовании директивы EQU. В этом случае начальный адрес сегмента жестко задается программистом, и он же следит за тем, чтобы сегменты не перекрывались друг другом в памяти микроконтроллера.

Использование абсолютных сегментов позволяет более гибко работать с памятью данных, т. к. теперь байтовые переменные в памяти данных могут быть назначены при помощи директивы резервирования памяти DS, а битовые переменные при помощи директивы резервирования битов DBIT.

Для определения абсолютных сегментов памяти используются следующие директивы:

- BSEG — абсолютный сегмент в области битовой адресации;
- CSEG — абсолютный сегмент в области памяти программ;
- DSEG — абсолютный сегмент в области внутренней памяти данных;
- ISEG — абсолютный сегмент в области внутренней памяти данных с косвенной адресацией;
- XSEG — абсолютный сегмент в области внешней памяти данных.

Директива BSEG позволяет определить абсолютный сегмент во внутренней памяти данных с битовой адресацией (битовые ячейки по адресам байтов с 32-ой по 47-ой – рис. 11.1) по определенному адресу. Эта директива не назначает имени сегменту, т. е. объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут AT не используется, то начальный адрес сегмента предполагается равным нулю. Использование битовых переменных позволяет значительно экономить внутреннюю память программ микроконтроллера.

Пример использования директивы BSEG для объявления битовых переменных приведен в листинге 8.24.

Листинг 8.24. Пример использования директивы BSEG перед определениями битовых переменных

```
BSEG AT 8                ; Сегмент начинается с 8-го бита
Rejind    DBIT 1         ; Флаг режима индикации
```

```
RejPriem   DBIT 1          ; Флаг режима приема
Flag       DBIT 1          ; Флаг общего назначения
```

(Начиная с 8-го бита в битовой области размещаем битовые переменные: в 8-ом бите (0-ой бит 33-го байта) - переменная Rejind, в 9-ом бите - RejPriem, в 10-ом бите - Flag)

Директива CSEG позволяет определить абсолютный сегмент в памяти программ по определенному адресу. Директива не назначает имени сегменту, т. е. объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут at. Если атрибут AT не используется, то начальный адрес сегмента предполагается равным нулю. Пример использования директивы CSEG для размещения подпрограммы обслуживания прерывания от таймера 0 приведен в листинге 8.25.

Листинг 8.25. Пример использования директивы CSEG для размещения подпрограммы обслуживания прерывания

```
; Перезагрузка таймера ----------------------------------------------------------------
 CSEG AT 0bh            ;Вектор прерывания от таймера 0, 0bh-начало сегмента.
IntT0:
mov TL0,  #LOW(-(F_ZQ/12)*10-2)      ;Настроить таймер
mov TH0,  #HIGH(-(F_ZQ/12)*10-2)     ;на период 10мс
reti
```

(по адресу 0b размещается команда mov TL0; по адресу 0c размещается команда mov TH0; по адресу 0d размещается команда reti; LOW - выделение младшего байта из числа в скобках, HIGH - выделение старшего байта из числа в скобках, F_ZQ - константа 12 МГц)

Директива DSEG позволяет определить абсолютный сегмент во внутренней памяти данных, работающей с прямой адресацией (с 0 по 127 ячейку), по определенному адресу. Предполагается, что к этому сегменту будут обращаться команды с прямой адресацией. Эта директива не назначает имени сегменту, т. е. объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут at не используется, то начальный адрес сегмента предполагается равным нулю. Пример использования директивы DSEG для объявления байтовых переменных приведен в листинге 8.26.

Листинг 8.26. Пример использования директивы DSEG перед определением байтовых переменных

```
DSEG AT 20h     ; Разместить в битовом пространстве микроконтроллера
                ;(для возможности одновременно битовой и байтовой адресации)
RejInd DS 1     ; Переменная, отображающая состояние программ
                ; обслуживания аппаратуры
Rejim DS 1      ; Переменная, отображающая режимы работы
Massiv DS 10    ; Десятибайтовый массив
```

(директивы: DS – заносит в память байтовые числа, DW – заносит в память 2-байтовые, DB – битовые. Начало сегмента с ячейки 20h (32ая ячейка); в 32-ой ячейке –

RejInd; в 33-ей ячейке – Rejim; а в 34-ой по 43-ю будут находится элементы массива Massiv)

Последний пример связан с примером, приведенным в листинге 8.24. То есть команды, изменяющие битовые переменные RejInd, RejPriem или Flag, одновременно будут изменять содержимое переменной Rejim, и наоборот команды, работающие с переменной Rejim, одновременно изменяют содержимое флагов RejInd, RejPriem или Flag. Такое объявление переменных позволяет написать наиболее эффективную программу управления контроллером и подключенными к нему устройствами.(Мы можем изменять как целый байт, так и отдельные биты этого байта это демонстрируют листинги 8.24 и 8.26)

Директива ISEG позволяет определить абсолютный сегмент во внутренней памяти данных с косвенной адресацией по определенному адресу (ячейки с 0 по 255).

Напомню, что адресное пространство внутренней памяти с косвенной адресацией в два раза больше адресного пространства памяти с прямой адресацией. Именно в этой области памяти размещается стек. Директива ISEG не назначает имени сегменту, т. е. объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут AT не используется, то начальный адрес сегмента предполагается равным нулю. Пример использования директивы ISEG для объявления байтовых переменных приведен в листинге 8.27.

```
ISEG AT 80h      ; Разместить сегмент в диапазоне адресов, совмещенных с SFR
Bufer DS 10      ; Десятибайтовый массив
Stack DS 118     ; Стек
```

(Разместить сегмент, начиная со 128-ой ячейки; 10 ячеек памяти, начиная со 128-ой будут отведены под десятибайтовый массив Bufer, Проблема примера: отведенный сегмент залазит в область памяти, отведенную под регистры специальных функций.)

Директива XSEG позволяет определить абсолютный сегмент во внешней памяти данных(внутренняя адресация 8-ми разрядная, внешняя может быть 16-ти разрядная) по определенному адресу. Эта директива не назначает имени сегменту, т. е. объединение сегментов из различных программных модулей невозможно. Для определения конкретного начального адреса сегмента применяется атрибут AT. Если атрибут AT не используется, то начальный адрес сегмента предполагается равным нулю. До недавнего времени использование внешней памяти не имело смысла, т. к. это значительно увеличивало габариты и цену устройства. Однако в последнее время ряд фирм стал размещать на кристалле значительные объемы ОЗУ, доступ к которому осуществляется как к внешней памяти. Так как эта директива применяется так же, как DSEG, то отдельный пример приводиться не будет.

Чтобы можно было использовать 16 разрядную адресацию используется 16-битный регистр DPTR. Пример:

```
MOVX A, @DPTR;Скопировать число из внешней ячейки памяти с адресом,
;хранящемся в DPTR, в аккумулятор
```

Использование абсолютных сегментов позволяет облегчить работу программиста по распределению памяти микроконтроллера для различных переменных. Однако в большинстве случаев абсолютный адрес переменной нас совершенно не интересует. Исключение составляют только регистры специальных функций. Так зачем же вручную задавать начальный адрес сегментов?

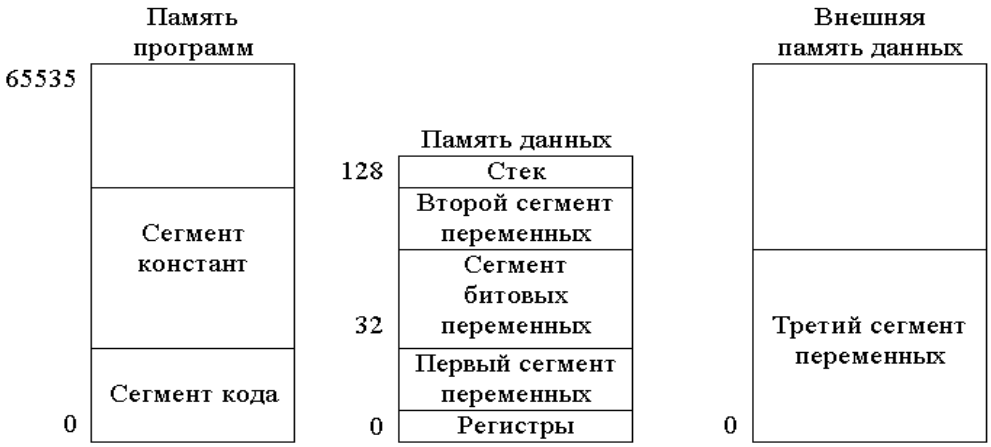
Одна из ситуаций, когда нас не интересует начальный адрес сегмента, — это программные модули. Как уже говорилось ранее, в программные модули обычно выносятся подпрограммы. Естественно, что конкретные адреса, по которым будут находиться эти подпрограммы в адресном пространстве микроконтроллера, нас тоже мало интересуют.

Если абсолютные адреса переменных или участков программ не интересны, то можно воспользоваться перемещаемыми сегментами.

12 Использование сегментов в языке программирования ASM-51: перемещаемые сегменты

В программе присутствует два сегмента: сегмент кода программы и сегмент данных. Кроме кодов команд в памяти программ хранятся константы, то есть в памяти программ микроконтроллера располагаются, по крайней мере, два сегмента: программа и данные. Чередование программы и констант может привести к нежелательным последствиям. Вследствие каких-либо причин данные могут быть случайно выполнены в качестве программы или наоборот программа может быть воспринята и обработана как данные.

Пример разбиения памяти программ и памяти данных на сегменты:



На примере размещения сегментов в адресном пространстве памяти программ и внутренней памяти данных видно, что при использовании нескольких сегментов переменных во внутренней памяти данных редактор связей может разместить меньший из них на месте неиспользованных банков регистров. Под сегмент стека обычно отводится вся область внутренней памяти, не занятая переменными. Это позволяет создавать программы с максимальным уровнем вложенности подпрограмм. Сегмент переменных, расположенный во внешней памяти данных, при использовании современных микросхем, таких как AduC842, может находиться в ОЗУ, расположенном на кристалле микроконтроллера.

Перемещаемые сегменты памяти

Если абсолютные адреса переменных или участков программ не интересны, то можно воспользоваться перемещаемыми сегментами. Имя перемещаемого сегмента задается директивой segment.

Директива segment позволяет определить имя сегмента и область памяти, где будет размещаться данный сегмент памяти. Для каждой области памяти определено ключевое слово:

- data – размещает сегмент во внутренней памяти данных с прямой адресацией;
- idata – размещает сегмент во внутренней памяти данных с косвенной адресацией;

- bit – размещает сегмент во внутренней памяти данных с битовой адресацией;
- xdata – размещает сегмент во внешней памяти данных;
- code – размещает сегмент в памяти программ;xx

Директива rseg. После определения имени сегмента можно использовать этот сегмент при помощи директивы rseg. Использование сегмента зависит от области памяти, для которой он предназначен. Если это память данных, то в сегменте объявляются байтовые или битовые переменные. Если это память программ, то в сегменте размещаются константы или участки кода программы. Пример использования директив segment и rseg для объявления битовых переменных:

```
_data segment idata
  public VershSteka, buferKlav
;Определение переменных---------------------------
 rseg  _data
buferKlav:  ds 8
VershSteka:
  End
```

Рисунок 6. Пример использования директив segment и rseg для объявления байтовых переменных

В этом примере объявлена строка buferKlav, состоящая из восьми байтовых переменных. Кроме того, в данном примере объявлена переменная VershSteka, соответствующая последней ячейке памяти, используемой для хранения переменных. Переменная VershSteka может быть использована для начальной инициализации указателя стека для того, чтобы отвести под стек максимально доступное количество ячеек внутренней памяти. Это необходимо для того, чтобы избежать переполнения стека при вложенном вызове подпрограмм.

Объявление и использование сегментов данных в области внутренней или внешней памяти данных не отличается от приведенного примера за исключением ключевого слова, определяющего область памяти данных.

Еще один пример использования директив segment и rseg приведен на рисунке 7. В этом примере директива segment используется для объявления сегмента битовых переменных.

```
_bits segment bit
public knIzm,strVv
; Определение битовых переменных ------------------------------------
rseg _bits
knIzm:   dbit 1
strVv:    dbit 1
end
```

Рисунок 7. Пример использования директив segment и rseg для объявления битовых переменных

Наибольший эффект от применения сегментов можно получить при написании основного текста программы с использованием модулей. Обычно каждый программный модуль оформляется в виде отдельного перемещаемого сегмента. Это позволяет редактору связей скомпоновать программу оптимальным образом. При использовании абсолютных сегментов памяти программ пришлось бы это делать вручную, а так как в процессе написания программы размер программных модулей постоянно меняется, то пришлось бы вводить защитные области неиспользуемой памяти между программными модулями.

Пример использования перемещаемых сегментов в исходном тексте программы приведен на рисунке 8.

```
_code segment code

; Старт программы----------------------------------------------
CSEG AT  0   ; Вектор рестарта процессора
reset:
jmp main
; Начало основной программы микроконтроллера ------------------------
rseg _code
main:
MOVX @DPTR,A
mov  SP,#VershSteka      ; Настроить указатель стека на вершину стека
call    init                    ; Настроить микроконтроллер
;-----------------------------------------------------------------
```

Рисунок 8. Пример использования директив segment и rseg в программном модуле

В этом примере приведен начальный участок основной программы микроконтроллера, на который производится переход с нулевой ячейки памяти программ. Использование такой структуры программы позволяет в любой момент времени при необходимости использовать любой из векторов прерывания, доступный в конкретном микроконтроллере, для которого пишется эта программа. Достаточно разместить определение этого вектора с использованием директивы cseg.

В приведенном примере использовано имя перемещаемого сегмента _code. Оно было объявлено в самой первой строке исходного текста программы. Конкретное имя перемещаемого сегмента может быть любым, но как уже говорилось ранее оно должно отображать ту задачу, которую решает данный конкретный модуль.

13 Программирование на языке С-51: структура программы, символы языка и управляющие последовательности

Каждая программа, написанная на языке программирования С-51, состоит из одного или более модулей. Каждый модуль записывается в отдельном файле и компилируется отдельно.

В модуле помещаются операторы, составляющей программы. Эти операторы выполняют необходимые действия, а также объявляют константы или переменные. Операторы, выполняющие действия, обязательно должны быть помещены в функции. Все подпрограммы, независимо от того, возвращают они значения или нет, называются функциями. Исполнение программы всегда начинается с функции с именем main (т. е. в простейшем случае достаточно написать только эту функцию).

Функция начинается с заголовка, в который входит тип возвращаемого значения, имя функции и круглые скобки, внутри которых объявляются параметры функции. Параметр — это определяемая функцией переменная, которая принимает передаваемый функции аргумент. Во всех функциях, которые ничего не возвращают, вместо типа возвращаемого значения указывается ключевое слово void. Исполняемые операторы, составляющие тело функции, заключаются в фигурные скобки.

Все переменные и константы обязательно должны быть объявлены до первого использования.

В исходном тексте программы, написанной на языке программирования С-51, используется часть ASCII- или ANSI-символов. Множество символов, используемых в языке программирования С, можно разделить на следующие группы:

1) Символы, используемые для ключевых слов языка С и идентификаторов: прописные и строчные буквы латинского алфавита A-Z, a-z, символ подчеркивания _, арабские цифры 0-9.
2) Прописные и строчные буквы русского алфавита. Эти буквы могут быть использованы в комментариях к исходному тексту программы и строковых константах.(А-Я, а-я)
3) Специальные символы языка С-51. Эти символы используются для записи вычисляемых выражений, а также для передачи компилятору определенного набора инструкций.

Символ	Наименование	Символ	Наименование
,	запятая	)	круглая скобка правая
.	точка	)	круглая скобка левая
;	точка с запятой	}	фигурная скобка правая
:	двоеточие	{	фигурная скобка левая

?	вопросительный знак	<	меньше
`	апостроф	>	больше
!	восклицательный знак	]	квадратная скобка правая
|	вертикальная черта	[	квадратная скобка левая
/	дробная черта	#	номер
\	обратная черта	%	процент
~	тильда	&	амперсант
*	звездочка	^	исключающее ИЛИ
+	плюс	=	равно
-	минус	“	кавычки

4) Управляющие и разделительные символы. К этой группе символов относятся: пробел, символы табуляции, перевода строки, возврата каретки, новой страницы и новой строки. Символы-разделители отделяют друг от друга лексические единицы языка, к которым относятся ключевые слова, константы, идентификаторы и т. д. Последовательность разделительных символов рассматривается компилятором как один символ (последовательность пробелов).
5) Набор специальных символов используется для создания управляющих последовательностей. Они нужны для управления устройствами ввода вывода. Управляющая последовательность может быть представлена следующими способами: \ и буква/цифра или своим 16-м кодом (обязательно 3-х разрядным, если написать 2-х разрядным, то транслятор и программа не воспримут его как управляющую последовательность).

Управляющие последовательности ООО и хННН (здесь о обозначает восьмеричную цифру; н обозначает шестнадцатеричную цифру) позволяют представить символ из кодовой таблицы ASCII или ANSI как последовательность восьмеричных или шестнадцатеричных цифр соответственно.

Управляющая последовательность	Наименование	Шестнадцатеричный код
\a	звонок	007
\b	возврат на шаг	008
\t	горизонтальная табуляция	009
\n	переход на новую строку	00A

\v	вертикальная табуляция	00B
\r	возврат каретки	00D
\f	новая страница	00C
\”	кавычки	022
\`	апостроф	027
\0	ноль-символ	000
\\	обратная дробная черта	05C
\000	восьмеричный код ASCII или ANSI-символов	000
\xHHH	шестнадцатеричный код	HHH

14 Программирование на языке С-51: идентификаторы, ключевые слова, простые и составные ограничители, числовые и текстовые строковые константы

Наименьшей единицей операторов С-51 является лексическая единица. Каждая из лексических единиц относится к одному из классов:

- идентификаторы;
- ключевые слова;
- простые ограничители (все специальные символы, кроме _, являются простыми ограничителями);
- составные ограничители (они образуются посредством определенных комбинаций двух спецсимволов, а именно: =>(стрелка, отношение),**(возведение в степень),:=(присваивание), /=(неравенство),<<(сдвиг влево),>>(сдвиг вправо), <=(меньше или равно), >=(больше или равно), /*(начало комментария), */(конец комментария), !=(не равно));
- числовые константы;
- текстовые строковые константы;

Ключевые слова, идентификаторы и числовые константы должны обязательно отделяться друг от друга. Если между двумя идентификаторами, числовыми константами или ключевыми словами не может быть указан простой или составной ограничитель, то в качестве разделителя между ними должен вставляться символ пробела. Для улучшения читабельности программы вместо одного символа пробел может использоваться несколько символов пробела.

14.1 Идентификаторы

Идентификаторы в языке программирования С-51 используются для определения имени переменной, подпрограммы, символической константы или метки оператора. Длина идентификатора может достигать 255 символов, но транслятор различает идентификаторы только по первым 31 символам.

Возникает вопрос - а зачем тогда нужен такой длинный идентификатор? Ответ: для создания "говорящего" имени подпрограммы или переменной, которое может состоять из нескольких слов. Например:

```
ProchitatPort();          // Прочитать порт
VklychitIndikator();      // Включить индикатор
```

В приведенном примере подпрограмма ProchitatPort выполняет действия необходимые для чтения порта, а подпрограмма VklychitIndikator выполняет действия, необходимые для зажигания индикатора. Намного легче прочитать действие подпрограммы непосредственно из имени подпрограммы, чем лазить каждый раз в алгоритм программы или искать исходный текст подпрограммы для того чтобы в

очередной раз разобраться - что же она выполняет? Для этого при объявлении имени подпрограммы можно потратить количество символов и большее чем 31.

То же самое можно сказать и про имена переменных. Например:

```
sbit ReleVklPitanija = 0x80;      // К нулевому выводу порта P0 подключено реле
sbit svDiod = 0x81;               // К первому выводу порта P0 подключен светодиод
sbit DatTemperat = 0x82;          // Ко второму выводу порта P0 подключен датчик температуры
```

В приведённом примере каждой ножке микроконтроллера назначается переменная с именем, отображающим устройство, подключённое к этой ножке. В результате при чтении программы не потребуется каждый раз обращаться к принципиальной схеме устройства каждый раз, как только производится операция записи или чтения переменной, связанной с портами микроконтроллера. В качестве идентификатора может быть использована любая последовательность строчных или прописных букв латинского алфавита и цифр, а также символов подчёркивания '_'. Идентификатор может начинаться только с буквы или символа '_', но ни в коем случае с цифры. Это позволяет программе-транслятору различать идентификаторы и числовые константы. Строчные и прописные буквы в идентификаторе различаются. Например: идентификаторы abc и ABC, A128B и a128b воспринимаются как разные.

Идентификатор создается при объявлении переменной, функции, структуры и т.п. после этого его можно использовать в последующих операторах разрабатываемой программы. Следует отметить важные особенности при определении идентификатора:

1. Идентификатор не должен совпадать с ключевыми словами, с зарезервированными словами и именами функций из библиотеки компилятора языка С.
2. Следует обратить особое внимание на использование символа подчеркивание (_) в качестве первого символа идентификатора, поскольку идентификаторы, построенные таким образом, могут совпадать с именами системных функций или переменных, в результате чего они станут недоступными.

Никто не запрещает объявлять идентификатор, совпадающий с именами функций из библиотек компилятора языка С. Однако после объявления такого идентификатора уже нельзя обратиться к функции с таким же именем.

Примеры правильных идентификаторов: A; XYR_56; OpredKonfigPriem; Byte_Prinjat; SvdiodGorit.

14.2 Ключевые слова

Ключевые слова – это зарезервированные слова, которые используются для построения операторов языка.

Список ключевых слов:

_at_(следом за ним	alien	auto	bdata	bit	break	case

идёт абсолютный адрес для сегмента)						
char(представления одиночного символа или для объявления литеральных строк)	code(размещение переменной в ПЗУ)	compact	const	continue	data	default
do	double	else	enum(Переменная, которая может принимать значение из некоторого списка значений)	extern	far	float
for	goto(переход к метке)	idata	if	int	interrupt	large
long	pdata	_priority_	reentrant	register	return	sbit
sfr	sfr16	short	signed	sizeof(определяет размер области памяти)	small	static
struct	switch	typedef	_task_	union	unsigned(нулевой бит как часть числа)	using
void(откладывает определение типа, на который	volatile	while(зацикливание программы)	xdata			

ссылается указатель)						

Ключевые слова не могут быть использованы в качестве идентификаторов.

14.3 Ограничители

Ограничитель — это один из следующих специальных символов из набора основных символов:

& ' () * + , - . / : ; < = > !

или один из следующих *составных ограничителей,* представляющих собой пару специальных символов:

=> .. ** := /= >= <= << >> <>

Каждый специальный символ является простым ограничителем, за исключением тех случаев,когда он встречается в составном ограничителе, в комментарии, в строковом, символьном или числовом литералах.

Остальные формы лексем описаны в других разделах данной главы.

Каждая лексема должна располагаться в одной строчке, поскольку конец строчки — разделитель. Символы кавычки, решетки, подчеркивания и два соседних дефиса не являются ограничителями, но могут входить в лексемы в качестве ее частей.

Лексема (лексический элемент) —это ограничитель, идентификатор (который может быть зарезервированным словом), числовой литерал, символьный литерал, строковый литерал или комментарий. Результат выполнения программы зависит только от конкретной последовательности лексем, исключая возможные комментарии.

Наименования составных ограничителей даны ниже:

=>(стрелка, отношение), **(возведение в степень), :=(присваивание), /=(неравенство), <<(сдвиг влево), >>(сдвиг вправо), <=(меньше или равно), >=(больше или равно), /*(начало комментария), */(конец комментария).

14.4 Константы

Константы предназначены для введения чисел в состав выражений операторов языка программирования С. В отличие от идентификаторов, всегда начинающихся с буквы, константы всегда начинаются с цифры. В языке программирования С-51 разделяют четыре типа констант:

1. целые знаковые и беззнаковые константы,

2. константы с плавающей запятой,

3. символьные константы и литеральные строки.

Целочисленные константы могут быть записаны как восьмеричные, десятичные или шестнадцатеричные числа в зависимости от того, какая система счисления удобнее для представления константы. Константа может быть представлена в десятичной, восьмеричной или шестнадцатеричной форме. При выполнении вычислений обычно пользуются десятичными константами. При работе с ножками микроконтроллера или передаче двоичных данных удобнее пользоваться двоичными числами или их более короткой формой записи — восьмеричными или шестнадцатеричными числами.

Десятичная константа состоит из одной или нескольких десятичных цифр, причем первая цифра не может быть нулем (иначе число будет воспринято как восьмеричное).

Восьмеричная константа состоит из обязательного нуля и одной или нескольких восьмеричных цифр (среди цифр должны отсутствовать цифры восемь и девять, так как эти цифры не входят в восьмеричную систему счисления). Если константа содержит цифру, недопустимую в восьмеричной системе счисления, то константа считается ошибочной.

Шестнадцатеричная константа начинается с обязательной последовательности символов 0x или 0X и содержит одну или несколько шестнадцатеричных цифр (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)

Примеры целых констант:

Десятичная константа	Восьмеричная константа	Шестнадцатиричная константа
16	020	0x10
127	0117	0x2B
240	0360	0xF0

Если требуется сформировать отрицательную целую константу, то используют знак "-" перед записью константы (который будет называться унарным минусом). Например: -0x2A, -088, -16.

Каждой целой константе присваивается тип, определяющий преобразования, которые должны быть выполнены, если константа используется в выражениях. Тип константы определяется следующим образом:

- десятичные константы рассматриваются как знаковые числа, и им присваивается тип int (целая) или long (длинная целая) в соответствии со значением константы. Если константа меньше 32768, то ей присваивается тип int в противном случае long.

- восьмеричным и шестнадцатеричным константам присваивается тип int, unsigned int (беззнаковая целая), long или unsigned long в зависимости от значения константы согласно ниже

Диапазон 16-х констант	Диапазон 8-х констант	Тип
0x0 - 0x7	0 - 07	char
0x0 - 0x7FFF	0 - 077777	int
0x8000 - 0xFFFF	0100000 - 0177777	unsigned int
0x10000 - 0x7FFFFFFF	0200000 - 017777777777	long
0x80000000 - 0xFFFFFFFF	020000000000 - 037777777777	unsigned long

Иногда требуется с самого начала интерпретировать константу как длинное целое число. Для того чтобы любую целую константу определить типом long, достаточно в конце константы поставить букву "l" или "L". Пример:

5l, 6l, 128L, 0105L, OX2A11L.

Примеры синтаксически недопустимых целочисленных констант:

12AF - шестнадцатеричная константа не имеет символов 0x в начале константы, поэтому по умолчанию для нее принимается десятичная система счисления, но тогда в ней присутствуют недопустимые символы.

0x2ADG - символ G недопустим при записи шестнадцатеричных чисел.

Константа с плавающей запятой - это десятичное число, представленное в виде действительного числа с десятичной запятой и порядком числа. Формат записи константы с плавающей запятой:

[цифры].[цифры] [E|e [+|-] цифры].

Число с плавающей запятой состоит из целой и дробные части и (или) порядка числа. Для определения отрицательного числа необходимо сформировать константное выражение, состоящее из знака минуса и положительной константы. В языке программирования С-51, в отличие от стандартного языка С, константы с плавающей точкой представляются с одинарной точностью (имеют тип float). Для определения отрицательного числа необходимо записать константное выражение, состоящее из знака минуса и положительной константы. Например:

115.75, 1.5E-2, -0.025, .075, -0.85E2

Символьная константа - представляется ASCII или ANSI символом, заключенном в апострофы. Управляющая последовательность тоже может быть использована в символьных константах. При этом она рассматривается как одиночный символ. Значением символьной константы является числовой код символа. Примеры символьных констант:

' ' – пробел , 'Q' – буква Q , '\n' – символ новой строки, '\' – обратная дробная черта, '\v' - вертикальная табуляция .

Символьные константы имеют тип int и при преобразовании типов дополняются знаком. Символьные константы используются обычно при управлении микроконтроллерным устройством от клавиатуры. Пример использования символьной константы на языке программирования С-51 приведён ниже:

if (NajKn=='p') VklUstr();

В этом примере если в переменной NajKn содержится код, соответствующий букве "p", то будет выполнена подпрограмма VklUstr.

Строковые константы. Если символьные константы используются обычно при вводе информации с клавиатуры, то при отображении информационных сообщений обычно используются целые строки символов. В строке допускается использование пробелов. Строковая константа (литерал или литеральная строка) - последовательность символов (включая строковые и прописные буквы русского и латинского а также цифры) заключенные в кавычки ("). Например: "Школа N 35", "город Тамбов", "YZPT КОД".

Отметим, что все управляющие символы, кавычка ("), обратная дробная черта (\) и символ новой строки в литеральной строке и в символьной константе представляются соответствующими управляющими последовательностями. Каждая управляющая последовательность представляет собой один символ. Например, при печати литерала "Школа \n N 35" его часть "Школа" будет напечатана на одной строке, а вторая часть "N 35" на следующей строке.

Символы литеральной строки обычно хранятся в памяти программ, но могут храниться и в памяти данных. В конец каждой литеральной строки компилятором добавляется нулевой символ, который можно записать как: "\0". Именно этот символ и является признаком конца строки.

Литеральная строка рассматривается как массив символов (char[ ]). Отметим важную особенность: число элементов массива равно числу символов в строке плюс 1, так как нулевой символ (символ конца строки) также является элементом массива. Все литеральные строки рассматриваются компилятором как различные объекты. Одна литеральная строка может выводиться на дисплей как несколько строк. Такие строки разделяются при помощи обратной дробной черты и символа возврата каретки \n. На одной строке исходного текста программы можно записать только одну литеральную строку. Если необходимо продолжить написание одной и той же литеральной строки на следующей строке исходного текста программы, то в конце строки исходного текста можно поставить обратную строку. Например исходный текст:

"строка неопределенной \

длины"

полностью идентичен литеральной строке:

"строка неопределенной длины".

Однако более удобно для объединения литеральных строк использовать символ (символы) пробела. Если в программе встречаются два или более литерала, разделенные только пробелами или символами табуляции, то они будут рассматриваться как одна литеральная строка. Этот принцип можно использовать для формирования литералов, занимающих более одной строки.

15 Типы данных языка программирования С-51 и их объявление

Описание переменных в языке программирования С имеет огромнейшее значение, т. к. именно оно в большинстве случаев определяет объем программы. Обычно большой объем загрузочного модуля программы вызван неправильным объявлением переменных в исходном тексте программы. Обращение к внутренним регистрам микроконтроллеров и внешним ресурсам разрабатываемого устройства тоже производится при помощи заранее объявленных переменных. В языке программирования С-51 любая переменная должна быть объявлена до первого использования этой переменной в программе.

Синтаксические диаграммы являются формальным и одновременно наглядным представлением правил, составленных из терминальных и нетерминальных имен. Для того чтобы сделать описание синтаксиса более компактным, часть определений нетерминальных имен будет приводиться в словесной форме.

Терминальными называются имена, входящие в текст программы так, как они написаны. Например, терминальными именами являются символьные или строковые константы. В синтаксических правилах такие терминальные имена заключаются в апострофы и кавычки соответственно.

Нетерминальные имена — это понятия, которые выводятся при помощи синтаксических правил с использованием других нетерминальных и терминальных имен, а также символов.

Объявление переменной в этом языке представляется в следующем виде:

[*Спецификатор класса памяти*] *Спецификатор типа*

[*Спецификатор типа памяти*] *Описатель* ['=' *Инициатор*]

[, *Описатель* ['=' *Инициатор*]]…

Описатель — идентификатор простой переменной либо более сложная конструкция с квадратными скобками, круглыми скобками или звездочкой (набором звездочек).

Спецификатор типа — одно или несколько ключевых слов, определяющих тип объявляемой переменной. В языке С-51 имеется стандартный набор типов данных, используя который, можно сконструировать новые (уникальные) типы данных. Перечень стандартных типов данных С-51 приведен в табл. 15.1.(Пример - float A=2.12)

Инициатор — задает начальное значение или список начальных значений, которое (которые) присваивается переменной при объявлении. (При объявлении переменной ей можно присвоить начальное значение при помощи инициатора, который начинается со знака "=": char screw = 12; char crew = 'V')

Спецификатор класса памяти — определяется одним из ключевых слов языка С-51: *auto, bit, extern, register, sbit, sfr, sfrl6 static*, и указывает, каким образом и в какой области памяти микроконтроллера будет распределяться память под объявляемую переменную, с одной стороны, а с другой — область видимости этой переменной, т. е. из каких программных модулей можно будет к ней обратиться.

Таблица 15.1

Типы данных	Размер		Область значений
	битов	байтов	
bit	1		От 0 до 1
signed char	8	1	От -128 до +127
unsigned char	8	1	От 0 до 255
enum	8/16	1/2	От -32768 до +32767
signed short	16	2	От -32768 до +32767
unsigned short	16	2	От 0 до 65535
signed int	16	2	От -32768 до +32767
unsigned int	16	2	От 0 до 65535
signed long	32	4	От -2147483648 до 2147483647
unsigned long	32	4	От 0 до 4294967295
float	32	4	От ±1.175E-38 до ±3.403E+38
sbit	1	0	От 0 до 1
sfr	8	1	От 0 до 255
sfr16	16	2	От 0 до 65535

Спецификатор типа памяти — определяется одним из шести ключевых слов языка C-51: *code, data, idata, bdata, xdata, pdata*, и указывает, в какой области памяти микроконтроллера будет размещена переменная.

Компилятор C51 обеспечивает следующие расширения ANSI-стандарта языка программирования C, необходимые для программирования микроконтроллеров MCS-51:

- области памяти;
- типы памяти;
- модели памяти;
- описатели типа памяти;
- описатели изменяемых типов данных;

- битовые переменные и данные с битовой адресацией;
- регистры специальных функций;
- указатели;
- атрибуты функций.

Типы данных *bit*, *sbit*, *sfr* и *sfri6* являются расширением языка программирования С-51 для поддержки процессора 8051. Они не описаны стандартом ANSI, поэтому к ним нельзя обращаться при помощи переменных-указателей.

16 Категории типов данных: целые, с плавающей запятой, перечислимые

Основные типы данных определяют с использованием следующих ключевых слов.

Для целых типов данных: *bit, sbit, char, int, short, long, signed, unsigned, sfr, sfrl6.*

Для типов данных с плавающей запятой: *float.*

Переменная любого типа может быть объявлена как неизменяемая. Это достигается добавлением ключевого слова const к спецификатору типа.

Объекты с квалификатором const представляют собой данные, используемые только для чтения, т. е. этой переменной в ходе выполнения программы не может быть присвоено новое значение. Отметим, что если после слова const отсутствует спецификатор типа, то подразумевается спецификатор типа int. Если ключевое слово const стоит перед объявлением составных типов (массив, структура, объединение, перечисление), то это приводит к тому, что каждый элемент также будет немодифицируемым, т. е. значение ему может быть присвоено только один раз.

Примеры использования ключевого слова const:

```
const float A=2.128E-2;
const B=286;        //подразумевается const int B=286
```

Отметим, что переменные со спецификатором класса памяти размещаются во внутреннем ОЗУ. Неизменяемость контролируется только на этапе трансляции. Для размещения переменной в ПЗУ лучше воспользоваться спецификатором типа памяти code.

16.1 Целочисленный тип данных

Для определения данных целочисленного типа используются различные ключевые слова, которые определяют диапазон значений и размер области памяти, выделяемой под переменные (табл. 9.7).

Тип	Размер памяти в битах	Размер памяти в байтах
bit	1	
char	8	1
unsigned char	8	1
int, short	16	2
unsigned int, unsigned short	16	2
long	32	4
unsigned long	32	4
sbit	1	

sfr	8	1
sfr16	16	2

Таблица 9.7 – спецификаторы целочисленных типов данных

Отметим, что ключевые слова signed и unsigned необязательны. Они указывают, как интерпретируется старший бит объявляемой переменной, т. е. если указано ключевое слово unsigned, то нулевой бит интерпретируется как часть числа, в противном случае нулевой бит интерпретируется как знаковый.

При отсутствии ключевого слова unsigned целочисленная переменная считается знаковой. В том случае, если спецификатор типа состоит из ключевого типа signed или unsigned и далее следует идентификатор переменной, то она будет рассматриваться как переменная типа int. Например:

```
unsigned int n;     // Беззнаковое шестнадцатиразрядное число n
unsigned int b;
int c;              // подразумевается signed int c
unsigned d;         // подразумевается unsigned int d
signed f;           // подразумевается signed int f
```

Отметим, что модификатор типа char используется для представления одиночного символа или для объявления литеральных строк. Численное значение объекта типа char соответствует ANSI-коду записанного символа (размером 1 байт).

Отметим также, что восьмеричные и шестнадцатеричные константы также могут иметь модификатор unsigned. Это достигается указанием префикса и или и после константы, константа без этого префикса считается знаковой.

Например:

```
0xA8C       // int signed;
017861      // long signed;
0xF7u       // int unsigned;
```

16.2 Числа с плавающей запятой

Для переменных, представляющих число с плавающей запятой используется модификатор типа float. Спецификатор double тоже допустим в языке программирования С-51, но он не приводит к увеличению точности результата.

Величина со спецификатором типа float занимает 4 байта. Из них 1 бит отводится для знака, 8 битов для избыточной экспоненты и 23 бита для мантиссы. Отметим, что старший бит мантиссы всегда равен 1, поэтому он явным образом в битовом представлении числа не указывается, в связи с этим диапазон значений переменной с плавающей точкой равен от ±1.175494E-38 до ±3.402823E+38.

Пример объявления переменной:

```
float f, a, b;
```

16.3 Переменные перечислимого типа

Переменная, которая может принимать значение из некоторого списка значений, называется переменной перечислимого типа или перечислением (enum). Использование такого вида переменной эквивалентно применению целочисленного знакового значения типа char или int. Это означает, что для переменной перечислимого типа будет выделен один или два байта в зависимости от максимального значения используемых этой переменной констант. В отличие от переменных целого типа, переменные перечислимого типа позволяют вместо безликих чисел использовать имена констант, которые более понятны и легче запоминаются.

Например, вместо использования чисел 1, 2, 3, 4, 5, 6, 7 можно использовать Названия Дней Недели: Poned, Vtorn, Sreda, Chetv, Pjatn, Subb, Voskr. При этом каждой константе будет соответствовать конкретное число.

Использование имен констант приведет к более понятной программе. Более того, транслятор отслеживает правильность использования констант и при попытке использования константы, не входящей в объявленный заранее список, выдает сообщение об ошибке.

Переменные enum-типа могут использоваться в индексных выражениях и как операнды в арифметических операциях и в операциях отношения.

Например:

If(rab_ned == SUB) dejstvie = rabota [rab_ned];

При объявлении перечисления определяется тип переменной перечисления и определяется список именованных констант, называемый списком перечисления. Значением каждого имени этого списка является целое число. Объявление перечислимой переменной начинается с ключевого слова enum и может быть представлено в двух формах:

"enum" [*Имя типа перечисления*] '{' *Список констант*'}' *Имя1* [',' *Имя2*…];

"enum" *Имя типа перечисления Описатель* [',' *Описатель*..];

В первом формате имена и значения констант задаются в Списке констант. Необязательное Имя типа перечисления — это идентификатор, который представляет собой тип переменной, соответствующий списку констант. За списком констант записывается Имя одной или нескольких переменных.

Список констант содержит одну или несколько конструкций вида:

Идентификатор ['=' *Константное выражение*]

Каждый *Идентификатор* — это имя константы. Все идентификаторы в списке констант оператора enum должны быть уникальными. Если константе явным образом не присваивается Константное выражение (чаще всего это число), то первому идентификатору присваивается значение 0, следующему — значение 1 и т. д.

Пример объявления переменной rab_ned и типа week для переменных, совместимых с переменной rab_ned, выглядит следующим образом:

```
enum week {SUB = 0,   // константе SUB присвоено значение 0
```

```
        VOS = 0,     // константе VOS присвоено значение 0
        POND,        // константе POND присвоено значение 1
        VTOR,        // константе VTOR присвоено значение 2
        SRED,        // константе SRED присвоено значение 3
        HETV,        // константе HETV присвоено значение 4
        PJAT         // константе PJAT присвоено значение 5
        } rab ned;
```

Идентификатор, связанный с *Константным выражением*, принимает значение, задаваемое этим *Константным выражением*. Результат вычисления *Константного выражения* должен иметь тип int и может быть как положительным, так и отрицательным. Следующему идентификатору в списке, если этот *Идентификатор* не имеет своего *Константного выражения*, присваивается значение, равное константному выражению предыдущего идентификатора плюс 1. Использование констант должно подчиняться следующим правилам:

- объявляемая переменная может содержать повторяющиеся значения констант;
- идентификаторы в списке констант должны быть отличны от всех других идентификаторов в той же области видимости, включая имена обычных переменных и идентификаторы из других списков констант;
- Имена типов перечислений должны быть отличны от других имен типов перечислений, структур и объединений в этой же области видимости;
- значение может следовать за последним элементом списка констант перечисления.

Во втором формате для объявления переменной перечислимого типа используется готовый тип переменной, уже объявленный ранее. Например: enum week rabl;

К переменной перечислимого типа можно обращаться при помощи указателей. При этом необходимо заранее определить тип переменной, на которую будет ссылаться указатель. Это может быть сделано, как описывалось выше или при помощи оператора typedef. Например:

```
typedef enum {   SUB = 0,    // константе SUB присвоено значение 0
                 VOS = 0,    // константе VOS присвоено значение 0
                 POND,       // константе POND присвоено значение 1
                 VTOR,       // константе VTOR присвоено значение 2
                 SRED,       // константе SRED присвоено значение 3
                 HETV,       // константе HETV присвоено значение 4
                 PJAT        // константе PJAT присвоено значение 5
        } week;
```

Этот оператор не объявляет переменную, а только определяет тип переменной, отличающийся от стандартного. В дальнейшем этот тип может быть использован для объявления переменных и указателей на переменные.

17 Программирование на языке С-51: указатели

Указатель — это переменная, которая может содержать адрес другой переменной. Указатель может быть использован для работы с переменной, адрес которой он содержит. Для инициализации указателя (записи в него начального адреса) можно использовать идентификатор переменной, при этом в качестве идентификатора может выступать имя переменной, массива, структуры, литеральной строки.

При объявлении переменной-указателя необходимо определить тип объекта данных, адрес которых будет содержать эта переменная, и идентификатор указателя с предшествующей звездочкой (или группой звездочек).

Формат объявления указателя: *Спецификатор типа* [*Модификатор*] '*' *Описатель*.

Спецификатор типа задает тип объекта и может быть любого основного типа, структуры или объединения (об этих типах данных будет сказано ниже).

Примеры объявления указателей:

```
unsigned int * a;   /* переменная a представляет собой указатель на целочисленную беззнаковую переменную */
float * x;          /* x может указывать на переменную с плавающей точкой */
char * fuffer;      /* объявляется указатель с именем fuffer, который может указывать на символьную переменную */
```

Задавая вместо спецификатора типа ключевое слово void, можно отсрочить определение типа, на который ссылается указатель. Переменная, объявляемая как указатель на тип void, может быть использована для ссылки на объект любого типа. Однако для того, чтобы можно было выполнить арифметические и логические операции над указателями или над объектами, на которые они указывают, необходимо при выполнении каждой операции явно определить тип объектов. Такие определения типов могут быть выполнены с помощью операции приведения типов.

```
void *addres;       /* Переменная addres объявлена как указатель на объект любого типа. Поэтому ей можно присвоить адрес любого объекта */
addres = &nomer;    /* (& — операция вычисления адреса). */
```

Однако, как было отмечено выше, ни одна арифметическая операция не может быть выполнена над указателем, пока не будет явно определен тип данных, на которые он указывает. В данном примере используется операция приведения типа (float *) для преобразования типа указателя address к типу float. Затем оператор ++ отдает приказ перейти к следующему адресу переменной с таким же типом.

```
(float *)address ++;
```

В качестве модификаторов при объявлении указателя могут выступать ключевые слова const, data, idata, xdata, code. Ключевое слово const указывает, что указатель не может быть изменен в программе.

17.1 Указатели общего вида

Указатели общего вида объявляются точно так же, как указатели в стандартном языке программирования С. Для того чтобы не зависеть от типа памяти, в которой может быть размещена переменная, для указателей общего вида выделяется 3 байта. В первом байте указывается вид памяти переменной, во втором байте — старший байт адреса, в третьем — младший байт адреса переменной. Указатели общего вида могут быть использованы для обращения к любым переменным независимо от типа памяти микроконтроллера. Многие библиотечные функции языка программирования С-51 используют указатели этого типа, поскольку в этом случае совершенно неважно, в какой именно области памяти размещаются переменные.

17.2 Специализированные указатели

В объявлениях специализированных указателей всегда включается модификатор памяти. Обращение всегда происходит к указанной области памяти, например:

```
char data *str;          // указатель на строку во внутренней памяти данных data
int xdata *numtab;       // указатель на целое во внешней памяти данных xdata
long code *powtab;       // указатель на длинное целое в памяти программ code
```

Поскольку модель памяти определяется во время компиляции, специализированным указателям не нужен байт, в котором указывается тип памяти микроконтроллера. Поэтому программа с использованием типизированных указателей короче и будет выполняться быстрее по сравнению с программой, использующей указатели общего вида. Специализированные указатели могут иметь размер в 1 байт (указатели на память idata, data, bdata и pdata) или в 2 байта (указатели на память code и xdata).

18 Программирование на языке С-51: массивы

При обработке данных достаточно часто приходится работать с рядом переменных одинакового типа (и описывающих одинаковые объекты). В этом случае эти переменные имеет смысл объединить одним идентификатором. Это позволяют сделать массивы.

Массивы — это группа элементов одинакового типа (char, float, int и т. п.). Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве. Объявление массива имеет два формата:

Спецификатор типа Имя [*Константное выражение*];

Спецификатор типа Имя [];

Имя — это идентификатор массива.

Спецификатор типа задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа void.

void - не имеет числовой размерности поэтому не может быть элементом массива.

Константное выражение в квадратных скобках задает количество элементов массива — размерность массива. При объявлении массива размерность может быть опущена в следующих случаях:

- массив инициализируется при объявлении;
- массив объявлен как формальный параметр функции;
- массив объявлен как внешняя переменная, явно определенная в другом файле.

В языке С-51 определены только одномерные массивы, но поскольку элементом массива в свою очередь тоже может быть массив, то таким образом можно определить и многомерные массивы. Они формализуются списком размерностей массива, следующих за идентификатором массива, причем каждое константное выражение размерности массива заключается в свои квадратные скобки.

Каждое константное выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит две размерности, трехмерного — три и т. д.

К каждому конкретному элементу массива можно обратиться при помощи его индекса (порядкового номера). Отметим, что в языке С первый элемент массива всегда имеет индекс, равный 0. Например:

```
A[0] = 78; //Записать число 78 в первый элемент массива A
int a [2] [3]; /*Определить двумерный массив, представленный в виде матрицы
                a[0] [0] a[0] [1] a[0] [2]
                a[1] [0] a[1] [1] a[1] [2] */
float b[10]; /* Определить массив из 10 элементов типа float */
int w[3] [3] = { {2,3,4}
                 {3,4,8}
                 {1,0,9}};
```

В последнем примере определен и инициализирован двумерный массив w[3] [3], состоящий из трех трехэлементных строк. Списки, выделенные в фигурные скобки,

соответствуют строкам массива и используются для инициализации (присваивания начальных значений) элементов массива.

В случае отсутствия скобок инициализация будет выполнена неправильно.

Для работы с литеральными строками в языке программирования С используются массивы символов, например: char str[] = «объявление символьного массива";

Следует учитывать, что размер символьного массива всегда на один элемент больше числа символов в строке, т. к. последний из элементов массива является управляющей последовательностью '\0', являющейся признаком конца строки. В примере использовано неявное задание длины массива символов. Это стало возможным, т.к. массиву сразу присваивается конкретное значение. При программировании микроконтроллеров семейства MCS-51 такое задание массива может привести к неоправданному расходу внутренней памяти данных, поэтому лучше воспользоваться размещением строки в памяти программ:char code str[] = «объявление массива символов";

19 Программирование на языке С-51: структуры

Работа с массивами облегчает понимание и написание программы, когда для обозначения похожих элементов используется один идентификатор. Однако в ряде случаев приходится обрабатывать разнородные элементы, описывающие один объект. В этом случае вместо массива используется структура.

Структура — это составной объект, в который входят элементы — называемые членами или полями — любых типов, за исключением функций, а также типа void или неполного типа. В отличие от массива, который является однородным объектом, структура может быть неоднородной.

В простейшем случае тип структуры определяется записью вида:

"struct" '{' *Список описаний* '}' [*Идентификатор*]

В структуре обязательно должно быть указано хотя бы одно поле. Определение полей структур имеет следующий вид:

Тип данных Описатель';'

где *Тип данных* указывает тип поля структуры для объектов, определяемых в *Описателях*. В простейшей форме описатели представляют собой идентификаторы переменных или массивы (массивы тоже могут входить в состав структуры).

Пример объявления структур:

```
struct
{int year;          // Поле структуры, в котором хранится год
char moth,          // Поле структуры, в котором хранится месяц
char day;           // Поле структуры, в котором хранится день
}date1, date2;      // Переменные, обозначающие две различные даты
```

В примере каждая из двух переменных — date1, date2 — состоит из трех полей: year, moth, day.

Доступ к отдельным полям структуры осуществляется с помощью указания имени структуры и следующего через точку имени поля, например:

```
st2.id=5;                  //Присвоить значение полю id структуры st2
st1.name="Иванов";         //Присвоить значение полю name структуры st1
st1_node.data=1985;        //Присвоить значение полю data структуры st1
st[1].name="Иванов";       //Занести студента Иванова в журнал
st[1].id=2;                //поместить его в журнал под вторым номером
st[1].age=23;              //Занести в журнал его возраст
```

Битовые поля

Элементом структуры может быть битовое поле, обеспечивающее доступ к отдельным битам памяти. Вне структур битовые поля объявлять нельзя. Нельзя также организовывать массивы битовых полей и нельзя применять к полям операцию определения адреса. В общем случае тип структуры с битовым полем задается в следующем виде:

"struct" '{' «unsigned» *Идентификатор1* ':' *Длина поля1*'

"unsigned" *Идентификатор2* ':' *Длина поля2*' '}'

Длина поля задается целым выражением или константой. Эта константа определяет число битов, отведенное соответствующему полю. Поле нулевой длины обозначает выравнивание на границу следующего слова.

Пример:

```
struct { unsigned a1 : 1;
        unsigned a2 : 2;
        unsigned a3 : 5;
        unsigned a4 : 2; } primer;
```

20 Программирование на языке С-51: объединения

Объединение – составной объект, в который могут входить элементы любых типов данных, за исключением функций, однако, в отличие от структуры, для всех элементов выделяется одна и та же область памяти, т.е. они перекрывают друг друга.

Соответствующая объединению память определяется памятью, необходимой для выделения на наиболее длинного его элемента, таким образом, при использовании элемента меньшей длины остаётся незадействованная память. Запись производится с первой ячейки памяти объединения.

Применение:

- интерпретация представления переменной одного типа в виде нескольких переменных другого типа;
- размещение переменных различного типа в одной и той же области памяти.

Объявление объединения:

union {
тип данных element_1;
тип данных element_2;
……
тип данных element_z;
} имя объединения;

Обращение к элементу объединения то же, как и у структуры: имя объединения.element_i.

Примеры

1. Интерпретация представления переменной одного типа в виде нескольких переменных другого типа.

Допустим, необходимо отправить данные типа integer через последовательный порт на некоторое устройство.

```
void main ()                    // основная программа
{                               //
union {                         // начало объявления объединения с именем madness
      int IKEA;                 // необходимые для отправки данные типа int
      char JOPA [4];            // необходимые для отправки данные в виде массива char
      } madness;                // конец объявления объединения с именем madness
madness.IKEA = brexit ();       // по выполнении функции brexit элемент объединения
                                // IKEA получает некое значение
decadance (madness.JOPA [0]);// отправка 1-го байта данных через функцию decadance
decadance (madness.JOPA [1]);// отправка 2-го байта данных через функцию decadance
decadance (madness.JOPA [2]);// отправка 3-го байта данных через функцию decadance
decadance (madness.JOPA [3]);// отправка 4-го байта данных через функцию decadance
}                               //
```

2. Размещение переменных различного типа в одной и той же области памяти.

Допустим, к МК подключено 100 датчиков по типу Master-Slave, каждый из которых отправляет свои данные в различных типах данных разного размера, а задача МК – просто отобразить эти данные на дисплее. Очевидно, что использование объединения наиболее экономный способ в плане памяти МК, поскольку вместо использования $\sum_1^{100}$ ■ [размерность данных $i-$ го датчика] байт мы используем [размерность самого большого элемента] байт.

```
void main ()                    // основная программа
{                               //
union {                         // начало объявления объединения с именем Nicht
        long Sensor_1;          // объявление 1-го элемента (8 байт)
        char Sensor_2;          // объявление 2-го элемента (1 байт)
        float Sensor_3;         // объявление 3-го элемента (4 байта)
        int Sensor_4 [2];       // объявление 4-го элемента (8 байт)
        ……                      // объявление ещё 95 элементов
        bit Sensor_100;         // объявление 100-го элемента (1 бит)
        } Nicht;                // конец объявления объединения с именем Nicht (8 байт)
Nicht.Sensor_1 = get (1);       // получение данных с 1-го датчика
send (Nicht.Sensor_1);          // отправка данных с 1-го датчика на дисплей
Nicht.Sensor_2 = get (2);       // получение данных со 2-го датчика
send (Nicht.Sensor_2);          // отправка данных 2-го датчика на дисплей
……                              // опустим работу 97 датчиками
Nicht.Sensor_100 = get (100);   // получение данных с 100-го датчика
send (Nicht.Sensor_100);        // отправка данных 100-го датчика на дисплей
}                               //
```

21 Программирование на языке С-51: определения новых типов и инициализация данных

21.1 Определение новых типов

Кроме определения переменных различных типов, имеется возможность заранее объявить тип переменной, а затем воспользоваться им при определении переменных. Использование при определении переменной заранее объявленного типа позволяет сократить определение, избежать ошибок при определении переменных в разных местах программы и добиться полной идентичности определяемых переменных.

Это можно сделать двумя способами. Первый способ – указать имя типа при объявлении структуры, объединения или перечисления, а затем использовать это имя в объявлении переменных и функций. Второй – использовать для объявления типа ключевое слово typedef. Последний способ предпочтителен, т.к. использование его приводит к более наглядным и понятным программам.

При объявлении с ключевым словом typedef идентификатор, стоящий на месте описываемого объекта, является именем вводимого в рассмотрение типа данных, и далее этот тип может быть использован для объявления переменных.

Любой тип может быть объявлен с использованием ключевого слова typedef, включая типы указателя, функции или массива. Имя с ключевым словом typedef для типов указателя, структуры, объединения может быть объявлено прежде, чем эти типы будут определены, но в пределах видимости объявителя.

Примеры объявления и использования новых типов:

```
typedef float (*MATH)();        // MATH – новый тип, представляющий собой
                                // указатель на функцию, возвращающую float
MATH sin;                       // sin – указатель на функцию, возвращающую float
// float (*sin)(); – эквивалентная двух строкам выше запись
typedef char YEAH [286];        // YEAH – массив из 286 символов
YEAH PERSONAGRATA;     // PERSONAGRATA – массив из 286 символов
// char PERSONAGRARA [286]; – эквивалентная двух строкам выше запись
```

При объявлении переменных и типов были использованы имена типов (MATH и FIO). Помимо определения переменных, имена типов могут еще использоваться в трех случаях: в списке формальных параметров при определении функций, в операциях приведения типов и в операции sizeof.

21.2 Инициализация данных

При объявлении переменной ей можно присвоить начальное значение при помощи инициатора, который начинается со знака "=":

- "инициатор" – при инициализации переменных основных типов и указателей:
  char screw = 12; char crew = 'V'.
- {"инициализаторы"} – при инициализации составных объектов:
  char [2][5] = {{"fuck"},{"you!"}}; char [][3] = {{1,2,3},{4,5}}.

При инициализации массива можно не указывать его размерность (будет подобрана автоматически) и прописывать не все элементы (тогда они будут приравнены к нулю). Если указать размерность меньше, чем инициализируется – лишнее будет отсечено. Помните, что в строке последний символ – "/0".

При инициализации объединения задается значение первого элемента объединения в соответствии с его типом.

```
union {
    float num;
    char hum [50];
    } bum = 3.14159265358979;
```

При инициализации составных объектов нужно внимательно следить за использованием скобок и списков инициализаторов, дабы не возникло ошибок.

Следующие две записи эквивалентны (см. теорему Вейерштрасса):

char [2][3] = {{2,8,6},{6,6,6}}; char [2][3] = {2,8,6,6,6,6};.

21.3 Отличие константы от константной переменной

Константа задаётся директивой препроцессора #define, определяющей идентификатор и последовательность символов, которой будет за-мещаться данный идентификатор при его обнаружении в тексте программы.

Константная переменная задаётся ключевым словом const, которое указывает на то, что объект или переменная являются неизменяемыми.

22 Программирование на языке С-51: выражения, операнды и операции

Выражение - комбинация знаков операций и операндов, результатом которой является определенное значение.

Знаки операций определяют действия, которые должны быть выполнены над операндами. Каждый операнд в выражении в свою очередь тоже может быть выражением. Значение выражения зависит от расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций.

Пример выражений: A+B

A*(B+C)-(D-E)/F

Пример выражений с преобразованием типов:

```
a=(int)b+(int)c      // переменные a и b могут быть восьмиразрядными
                     // преобразование типов нужно, чтобы избежать переполнения
s=sin((float)a/15)   // если не преобразовать тип переменной a, то деление
                     // будет целочисленным и результат может быть 0, если a<15.
```

(44=200+100, т.к. 300 в двоичной системе имеет 9 разрядов **1**$00101100_2(300_{10})$, а chair - однобайтная переменная- 8 разрядов, наступит переполнение и **1** не сохранится, $101100_2(44_{10})$)

Операнд - это объекты, над которыми или при помощи которых выполняются действия, задаваемые инструкциями или директивами. Это числа(целые и вещественные), литеральная строка, идентификатор, адреса ячеек, функции(математические, статические и др.), выражение в круглых скобках (арифметическое, логическое, строковые). Каждый операнд имеет тип.

Если операнд - константа, то ему соответствует значение и тип представляющей его константы. Целая константа может быть типа int, long, unsigned int, unsigned long, в зависимости от ее значения и от формы записи. Символьная константа имеет тип int. Константа с плавающей точкой всегда имеет тип double.

Например, 1 + 2, где 1 и 2 - это операнды, а знак ‘+’ - это оператор.

Оператор – это символ в языке программирования, а операция - это действие, которое выполняется с помощью этого символа. Например, оператор ‘+’ выполняет операцию сложения. По количеству операндов, участвующих в операции, операции подразделяются на унарные, бинарные и тернарные. Унарные операции:

Знак операции	Наименование операции
-	Унарный минус
~	Поразрядное отрицание(дополнение)
!	Логическое отрицание
*	Разадресация (косвенная адресация)
&	Вычисление адреса
+	Унарный плюс
++	Инкремент (увеличение на 1)
--	Декремент (уменьшение на 1)
sizeof	Размер

Унарные операции выполняются справа налево.

Операции инкремента и декремента увеличивают или уменьшают значение операнда на единицу и могут быть записаны как справа, так и слева от операнда. Если знак операции записан перед операндом, то изменение операнда происходит до его использования в выражении. Если знак операции записан после операнда, то операнд вначале используется в выражении, а затем происходит его изменение.

Бинарные операции выполняются слева направо. Бинарные операции:

Знак операции	Наименование операции	Группа операций
*	Умножение	Мультипликативные
/	Деление	
%	Остаток от деления	
+	Сложение	Аддитивные
-	Вычитание	
<<	Сдвиг влево	Операции сдвига
>>	Сдвиг вправо	
<	Меньше	Операции отношения
<=	Меньше или равно	
>	Больше	
>=	Больше или равно	
==	Равно	
!=	Не равно	
&	Поразрядное И	Поразрядные операции
|	Поразрядное ИЛИ	
^	Поразрядное исключающее ИЛИ	
&&	Логическое И	Логические операции
||	Логическое ИЛИ	
,	Последовательное вычисление	Последовательного вычисления
=	Присваивание	Операции присваивания
*=	Умножение с присваиванием	
/=	Деление с присваиванием	
%=	Остаток от деления с присваиванием	
-=	Вычитание с присваиванием	
+=	Сложение с присваиванием	
<<=	Сдвиг влево с присваиванием	
>>=	Сдвиг вправо с присваиванием	
&=	Поразрядное И с присваиванием	
|=	Поразрядное ИЛИ с присваиванием	
^=	Поразр. исключающее ИЛИ с присв-м	

Левый операнд операции присваивания должен быть выражением, ссылающимся на область памяти (но не идентификатором, объявленным с ключевым словом constant). Левый операнд не может также быть массивом.

(Операции выполняются на этапе трансляции исходного текста программы для определения конкретного числа, которое используется в команде.)

При записи выражений следует помнить, что символы ‘*’, ‘&’, ‘-’, ‘+’ могут обозначать как унарную, так и бинарную операцию.

23 Программирование на языке С-51: преобразования типов при выполнении выражений

Когда переменные и константы различных типов смешиваются в выражениях, то происходит преобразование к одному типу. Компилятор преобразует все операнды к типу большего операнда. Ниже описываются правила преобразования типов.

1. Все переменные типа char и short int преобразуются к типу int. Все переменные типа float преобразуются к типу double.
2. Если один из пары операндов имеет тип long double, другой операнд также преобразуется к long double.
3. Если один из операндов имеет тип double, другой операнд также преобразуется к double.
4. Если один из операндов имеет тип long, другой операнд также преобразуется к long,
5. Если один из операндов имеет тип unsigned, другой операнд также преобразуется к unsigned.

Таким образом, можно отметить, что при вычислении выражений операнды преобразуются к типу того операнда, который имеет наибольший размер. Пример преобразования типов при вычислении математического выражения:

```
float ft,sd;
unsigned  char ch;
unsigned long in;
int i;
....
sd=ft*(i+ch/in);
```

Преобразование значения переменной одного типа в значение другого типа называется приведение типа и бывает явным и неявным:

- При явном приведении перед выражением следует указать в круглых скобках имя типа, к которому необходимо преобразовать исходное значение.
- При неявном приведении преобразование происходит автоматически, по правилам, заложенным в языке Си.

Пример явного приведения типа:

```
int x = 5;
double y = 15.3;
x = (int) y;
y = (double) x;
```

Пример неявного приведения типа:

```
int x = 5;
double y = 15.3;
y = x; //здесь происходит неявное приведение типа к double
x = y; //здесь происходит неявное приведение типа к int
```

Преобразовании знаковых чисел при преобразовании к более короткому/длинному типу:

- Знаковое целое значение преобразуется к короткому знаковому целому значению (short signed int) посредством усечения старших битов.
- Знаковое целое значение преобразуется к длинному знаковому целому значению (long signed int) путем расширения знака влево.
- Преобразование знаковых целых значений к плавающим значениям происходит путем преобразования к типу long, а затем преобразования к плавающему типу.
- При преобразовании знакового целого значения к беззнаковому целому значению (unsigned int) производится лишь преобразование к размеру беззнакового целого типа, и результат интерпретируется как беззнаковое целое значение.

Преобразовании беззнаковых чисел при преобразовании к более короткому/длинному типу:

- Беззнаковое целое значение преобразуется к короткому беззнаковому целому значению или короткому знаковому целому значению путем усечения старших битов.
- Беззнаковое целое значение преобразуется к длинному беззнаковому целому значению или длинному знаковому целому значению путем дополнения нулями слева.
- Беззнаковое целое значение преобразуется к значению с плавающей точкой путем преобразования к типу long, а затем преобразования значения типа long к значению с плавающей точкой.
- Если беззнаковое целое значение преобразуется к знаковому целому значению того же размера, то битовое представление не меняется. Однако, если старший (знаковый) бит был установлен в единицу, представляемое значение изменится.

// Думаю, что это, как дополнительная информация, если спросит

Преобразование типов указателя. Указатель на величину одного типа может быть преобразован к указателю на величину другого типа. Однако результат может быть не определен из-за отличий в требованиях к выравниванию и размерах для различных типов.

Указатель на тип **void** может быть преобразован к указателю на любой тип и указатель на любой тип может быть преобразован к указателю на тип **void** без ограничений. Значение указателя может быть преобразовано к целой величине. Метод преобразования зависит от размера указателя и размера целого типа следующим образом:

— если размер указателя меньше размера целого типа или равен ему, то указатель преобразуется точно так же, как целое без знака;

— если размер указателя больше, чем размер целого типа, то указатель сначала преобразуется к указателю с тем же размером, что и целый тип, и затем преобразуется к целому типу.

Целый тип может быть преобразован к адресному типу по следующим правилам:

— если целый тип того же размера, что и указатель, то целая величина просто рассматривается как указатель (целое без знака);

— если размер целого типа отличен от размера указателя, то целый тип сначала преобразуется к размеру указателя (используются способы преобразования, описанные выше), а затем полученное значение трактуется как указатель.

Преобразования при вызове функции. Преобразования, выполняемые над аргументами при вызове функции, зависят от того, был ли задан прототип функции (объявление «вперед») со списком объявлений типов аргументов.

Если задан прототип функции, и он включает объявление типов аргументов, то над аргументами при вызове функции выполняются только обычные арифметические преобразования.

Если прототип функции отсутствует, то при вызове происходят только обычные арифметические преобразования для аргументов. Эти преобразования выполняются независимо для каждого аргумента. Величины типа **float** преобразуются к **double**, величины типа **char** и **short** преобразуются к **int**, величины типов **unsigned char** и **unsigned short** преобразуются к **unsigned int**. Могут быть также выполнены неявные преобразования переменных типа указатель. Задавая прототипы функций, можно переопределить эти неявные преобразования и позволить компилятору выполнить контроль типов.

24 Программирование на языке С-51: операции вычисления адреса и разадресации; поразрядные логические операции

Эти операции используются для работы с переменными типа указатель.

Операция разадресации ('*') позволяет осуществить доступ к переменной при помощи указателя. Операнд операции разадресации обязательно должен быть указателем. Результатом операции является значение переменной, на которую указывает операнд. Типом результата является тип переменной, на которую ссылается указатель.

В отличие от прямого использования переменных использование указателей может приводить к непредсказуемым результатам. Результат не определен, если указатель содержит недопустимый адрес.

Рассмотрим типичные ситуации, когда указатель содержит недопустимый адрес:

1. указатель является нулевым;
2. указатель определяет адрес такого объекта, который не является активным в момент использования указателя;
3. указатель определяет адрес, который не выровнен до типа объекта, на который он указывает;
4. указатель определяет адрес, не используемый выполняющейся программой.

Операция вычисления адреса переменной (&) возвращает адрес своего операнда. Операндом может быть любой идентификатор. Имя функции или массива также может быть операндом операции «адрес», хотя в этом случае применение знака '&' является лишним, т. к. имена массивов и функций изначально являются адресами.

Операция & не может применяться к элементам структуры, являющимся полями битов, т. к. эти элементы не выровнены по байтам. Кроме того, эта операция не может быть применена к объектам с классом памяти register.

Примеры:

```
int t,          // Объявляется переменная целого типа t
f=0,            // Объявляется переменная f и ей присваивается 0
*adress;        // Объявляется указатель на переменные целого типа
adress = &t     // указателю adress присваивается адрес переменной t
*adress =f;     /* переменной, находящейся по адресу, содержащемуся в
переменной adress, т.е. переменной t, присваивается значение переменной f, т.е. 0, что
эквивалентно оператору t=f; */
```

Операция sizeof. С помощью этой операции можно определить размер области памяти, которая соответствует идентификатору или типу переменной. Она записывается в следующем виде: "sizeof ("Выражение') '

В качестве выражения может быть использован любой идентификатор, либо имя типа, заключенное в скобки. Отметим, что не может быть использовано имя типа void, а идентификатор не может относиться к полю битов структуры или быть именем функции.

Если в качестве выражения указано имя массива или структуры, то результатом является размер всего массива (т. е. произведение числа элементов на длину типа) или структуры.

К поразрядным (побитовым) операциям относятся: операция поразрядного «И» (&), операция поразрядного «ИЛИ» (|) и операция поразрядного «исключающего ИЛИ» (^).

Операнды поразрядных операций могут быть любого целого типа. При необходимости над операндами выполняются преобразования по умолчанию, тип результата — это тип операндов после преобразования.

Операция поразрядного «И» (&) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция поразрядного «ИЛИ» (|) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой (или оба) из сравниваемых битов равен 1, то соответствующий бит результата устанавливается в 1, в противном случае результирующий бит равен 0.

Операция поразрядного «исключающего ИЛИ» (^) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если один из сравниваемых битов равен 0, а второй бит равен 1, то соответствующий бит результата устанавливается в 1, в противном случае, т. е. когда оба бита равны 1 или 0, бит результата устанавливается в 0.

Пример:

```
int     i=0x45FF        /*      i= 0100 0101 1111 1111  */
        j= 0x00FF       /*      j= 0000 0000 1111 1111  */
char r;
        r = i^j;        /*      r=0x4500 = 0100 0101 0000 0000      */
        r = i | j;      /*      r = 0x45FF = 0100 0101 1111 1111    */
        r = i&j         /*      r = 0x00FF = 0000 0000 1111 1111    */
```

25 Программирование на языке С-51: мультипликативные и аддитивные операции, операции сдвига, операция последовательного вычисления

25.1 Мультипликативные операции

К этому классу операций относятся операции: умножения (*), деления (/), получение остатка от деления (%)

Операндами операции (%) должны быть целые числа.

Типы операндов операций умножения и деления могут отличаться, и для них справедливы правила преобразования типов. Типом результата является тип операндов после преобразования.

Операция умножения (*) выполняет умножение операндов.

```
int  i=5;
float f=0.2;
double g,z;
g=f*i;
```

Тип произведения i и f преобразуется к типу **double**, затем результат присваивается переменной g.

Операция деления (/) выполняет деление первого операнда на второй. Если две целые величины не делятся нацело, то результат округляется в сторону нуля.

При попытке деления на ноль выдается сообщение во время выполнения.

```
int i=49, j=10,  n, m;
n = i/j;              // результат   4
m = i/(-j);           // результат  -4
```

Операция остаток от деления (%) дает остаток от деления первого операнда на второй.

Знак результата зависит от конкретной реализации. В данной реализации знак результата совпадает со знаком делимого. Если второй операнд равен нулю, то выдается сообщение.

```
int n = 49, m = 10, i, j, k, l;
i = n % m;            // 9
j = n % (-m);         // 9
k = (-n) % m;         // -9
l = (-n) % (-m);      // -9
```

25.2 Аддитивные операции

К аддитивным операциям относятся сложение (+) и вычитание (-).

Операнды могут быть целого или плавающего типов. В некоторых случаях над операндами аддитивных операций выполняются общие арифметические преобразования. Преобразования, выполняемые при аддитивных операциях, не обеспечивают обработку

ситуаций переполнения и потери значимости. Информация теряется, если результат аддитивной операции не может быть представлен типом операндов после преобразования. При этом сообщение об ошибке не выдается.

```
int   i=30000, j=30000, k;
k=i+j;
```

В результате сложения k получит значение равное -5536.
(Их сумма не укладывается в 15 разрядов, происходит переполнение. Программист должен сам вести контроль, т.е. производить сравнение в программе.)

Результатом выполнения операции сложения является сумма двух операндов. Операнды могут быть целого или плавающего типа или один операнд может быть указателем, а второй — целой величиной.

Когда целая величина складывается с указателем, то целая величина преобразуется путем умножения ее на размер памяти, занимаемой величиной, адресуемой указателем.

Когда преобразованная целая величина складывается с величиной указателя, то результатом является указатель, адресующий ячейку памяти, расположенную на целую величину дальше от исходного адреса. Новое значение указателя адресует тот же самый тип данных, что и исходный указатель.

Пример:

```
char   *a=5; //Объявить указатель и настроить на ячейку памяти с адресом 5
int    *b=5; //Объявить указатель и настроить на ячейку памяти с адресом 5
long   *c=5; //Объявить указатель и настроить на ячейку памяти с адресом 5
a=a+5;       //Увеличить значение указателя на 5 (адрес 10)
b=b+5;       //Увеличить значение указателя на 5 (адрес 15)
c=c+5;       //Увеличить значение указателя на 5 (адрес 25)
```

(char-1 байт, int-2 байта, long-4 байта)

Операция вычитания (-) вычитает второй операнд из первого. Возможна следующая комбинация операндов:

- Оба операнда целого или плавающего типа.
- Оба операнда являются указателями на один и тот же тип.
- Первый операнд является указателем, а второй — целым.

Операции сложения и вычитания над адресами в единицах, отличных от длины типа, могут привести к непредсказуемым результатам.

```
double d[10],* u;
int i;
u = d+2;  /*  u  указывает на третий элемент массива */
i = u-d;  /*  i  принимает значение равное 2        */
```

25.3 Операции сдвига

Операции сдвига осуществляют смещение операнда влево (<<) или вправо (>>) на число битов, задаваемое вторым операндом. Оба операнда должны быть целыми величинами. Выполняются обычные арифметические преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в нуль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа первого операнда. Если тип unsigned, то свободные левые биты устанавливаются в нуль. В противном случае они заполняются копией знакового бита. Результат операции сдвига не определен, если второй операнд отрицательный.

Преобразования, выполненные операциями сдвига, не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат операции сдвига не может быть представлен типом первого операнда, после преобразования.

Сдвиг влево соответствует умножению первого операнда на степень числа 2, равную второму операнду, а сдвиг вправо соответствует делению первого операнда на 2 в степени, равной второму операнду.

```
int  i=0x1234, j,  k ;
k=i<<4 ;      /*  k="0x2340" */
```

Сдвиг влево на 4, а визуально сдвинуто на одну позицию, т.к. каждый разряд шестнадцатеричной соответствует четырём разрядам двоичной:

$1234_{16} = 0010\ 010\ 0011\ 0100_2$

(т.е. при сдвиге четырёх двоичных разрядов сдвигается один шестнадцатеричный)

```
j=i<<8 ;      /*  j="0x3400" */
i=j>>8 ;      /*  i= 0x0034   */
```

25.4 Операция последовательного вычисления

Операция последовательного вычисления обозначается запятой (,) и используется для вычисления двух и более выражений там, где по синтаксису допустимо только одно выражение. Эта операция вычисляет два операнда слева направо. При выполнении операции последовательного вычисления, преобразование типов не производится. Операнды могут быть любых типов. Результат операции имеет значения и тип второго операнда. Запятая может использоваться также как символ разделитель, поэтому необходимо по контексту различать, запятую, используемую в качестве разделителя или знака операции.

```
/* пример 1 */
for(i=j=1;   i+j<20;   i+=i,  j--)…
```

Каждый операнд третьего выражения оператора цикла for вычисляется независимо. Сначала вычисляется i+=i, затем j--.

```
/* пример 2 */
func_one( x, y + 2, z);
```

```
func_two((x--, y + 2), z);
```

Во втором примере символ "запятая" используется как разделитель в двух различных контекстах. В первом вызове функции func_onc передаются три аргумента, разделенных запятыми: x, y+2, z. Здесь символ "запятая" используется просто как разделитель.

В вызове функции func_two внутренние скобки вынуждают компилятор интерпретировать первую запятую как операцию последовательного вычисления. Этот вызов передает функции func_two два аргумента. Первый аргумент — это результат последовательного вычисления (x--,y+2), имеющий значение и тип выражения y+2. Вторым аргументом является z.

26 Программирование на языке С-51: условная операция, инкрементирование и декрементирование, простое и составное присваивания; приоритеты операций

26.1 Условная операция

В языке С-51 имеется одна тернарная операция — условная операция:

Операнд1 '?' Операнд2 ':' Операнд3

В условной операции *Операнд1* должен иметь целый или плавающий тип или быть указателем. Он оценивается с точки зрения эквивалентности 0.

Если *Операнд1* не равен 0, то вычисляется *Операнд2*, и его значение является результатом операции. Если *Операнд1* равен 0, то вычисляется *Операнд3* и его значение является результатом операции. Следует отметить, что при выполнении этой операции вычисляется либо *Операнд2*, либо *Операнд3*, но ни в коем случае не оба сразу. Тип результата зависит от типов *Операнд2* и *Операнд3* следующим образом:

1. Если операнды имеют целый или плавающий тип (отметим, что их типы могут отличаться), то выполняются обычные арифметические преобразования. Типом результата является тип операнда после преобразования.

2. Если оба операнда имеют один и тот же тип структуры, объединения или указателя, то тип результата будет тем же самым типом структуры, объединения или указателя.

3. Если оба операнда имеют тип **void**, то результат имеет тип **void**.

4. Если один операнд является указателем на объект любого типа, а другой операнд является указателем на **void**, то указатель на объект преобразуется к указателю на void, который и будет типом результата.

5. Если один из операндов является указателем, а другой константным выражением со значением 0, то типом результата будет тип указателя.

Пример:

```
max = (d<=b)? b: d;
```

Переменной max присваивается максимальное значение переменных d и b.

26.2 Операции инкремента и декремента

Операции инкремента (++) и декремента (--) являются унарными операциями присваивания. Они соответственно увеличивают или уменьшают значения операнда на единицу. Эти операции обычно используются для организации циклов и перехода от адреса одной переменной к адресу другой переменной. Операнд может быть целого или плавающего типа или быть указателем и при этом обязательно должен быть модифицируемым. Тип результата соответствует типу операнда. Операнд адресного типа увеличивается или уменьшается на размер объекта, который он адресует.

В языке программирования С-51 допускается префиксная или постфиксная формы операций инкремента (декремента). Если знак операции стоит перед операндом (префиксная форма записи), то изменение операнда происходит до его использования в выражении и результатом операции является увеличенное или уменьшенное значение операнда.

В том случае, если знак операции стоит после операнда (постфиксная форма записи), то операнд вначале используется для вычисления выражения, и только затем происходит изменение операнда.

Примеры использования операции инкремента:

```
int t=1, s=2, z, f;
    z=(t++)*5;
```

Вначале происходит умножение t*5, а затем увеличение t. В результате получится z=5, t=2.

```
f=(++s)/3;
```

Вначале значение s увеличивается, а затем используется в операции деления. В результате получим s=3, f=1.

В случае, если операции увеличения и уменьшения используются как самостоятельные операторы, префиксная и постфиксная формы записи становятся эквивалентными.

```
z++; /* эквивалентно */ ++z;
```

26.3 Простое присваивание

Операция простого присваивания используется для замены значения левого операнда, значением правого операнда. При присваивании производится преобразование типа правого операнда к типу левого операнда по правилам, упомянутым раньше. Левый операнд должен быть модифицируемым.

Пример:

```
int t;
char f;
long z;
t=f+z;
```

Значение переменной f преобразуется к типу **long**, вычисляется f+z, результат преобразуется к типу **int** и затем присваивается переменной t.

26.4 Составное присваивание

Кроме простого присваивания, имеется группа операций присваивания, которые объединяют простое присваивание с одной из бинарных операций. Такие операции называются составными операциями присваивания и имеют следующий вид:

Операнд1 Бинарная операция '=' Операнд2.

Составное присваивание эквивалентно следующему простому присваиванию:

Операнд1 '=' Операнд1 Бинарная операция Операнд2.

Каждая операция составного присваивания выполняет преобразования, которые осуществляются соответствующей бинарной операцией. Левым операндом операций «+=» и «-=» может быть указатель, в то время как правый операнд должен быть целым числом.

Примеры:

```
double arr[4]={ 2.0, 3.3, 5.2, 7.5 };
double b=3.0;
b+=arr[2];          /* эквивалентно оператору b=b+arr(2)          */
arr[3]/=b+1;  /* эквивалентно оператору  arr[3]=arr[3]/(b+1) */
```

Заметим, что при втором присваивании использование составного присваивания дает заметный выигрыш времени выполнения программы, т. к. левый операнд является индексным выражением, а значит для его вычисления потребуется несколько команд микроконтроллера.

26.5 Приоритеты операций и порядок вычислений

В языке С-51 операции с высшими приоритетами вычисляются первыми. Наивысшим является приоритет, равный 1. Приоритеты и порядок операций приведены в таблице ниже.

Приоритет	Знак операции	Типы операции	Порядок выполнения
1	+ - ~ ! *& ++ -- sizeof приведение типов	Унарные	Справа налево
2	( ) [ ] . ->	Выражение	Слева направо
3	* / %	Мультипликативные	Слева направо
4	+ -	Аддитивные	Слева направо
5	<< >>	Сдвиг	Слева направо
6	< > <= >=	Отношение	Слева направо
7	== !=	Отношение(равенство)	Слева направо
8	&	Поразрядное “И”	Слева направо
9	^	Поразрядное “исключающее ИЛИ”	Слева направо
10	|	Поразрядное “ИЛИ”	Слева направо
11	&&	Логическое “И”	Слева направо
12	||	Логическое “ИЛИ”	Слева направо
13	? :	Условная	Справа налево
14	= *= /= %= += -= &= |= >>= <<= ^=	Простое и составное присваивание	Справа налево

15	,	Последовательное вычисление	Слева направо

Побочные эффекты

Операции присваивания в сложных выражениях могут вызывать побочные эффекты, т. к. они изменяют значение переменной. Побочный эффект может возникать и при вызове функции, если он содержит прямое или косвенное присваивание (через указатель). Это связано с тем, что аргументы функции могут вычисляться в любом порядке. Например, побочный эффект имеет место в следующем вызове функции:

prog (a, a=k*2);

В зависимости от того, какой аргумент вычисляется первым, в функцию могут быть переданы различные значения.

Порядок вычисления операндов некоторых операций зависит от реализации компилятора, и поэтому могут возникать разные побочные эффекты, если в одном из операндов используются операции увеличения или уменьшения, а также другие операции присваивания.

Например, выражение i*j + (j++) + (-i) может принимать различные значения при обработке разными компиляторами. Чтобы избежать недоразумений при выполнении программ из-за побочных эффектов, необходимо придерживаться следующих правил:

1. Не использовать операции присваивания переменной значения в вызове функции, если эта переменная участвует в формировании других аргументов функции.

2. Не использовать операции присваивания переменной значения в выражении, где она используется более одного раза.

27 Программирование на языке С-51: операторы (пустой, «выражение», составной)

27.1 Пустой оператор

Пустой оператор состоит только из точки с запятой. При выполнении этого оператора ничего не происходит. Он обычно используется в следующих случаях:

- в операторах do, for, while, if в строках, когда не требуется выполнение каких-либо действий, но по синтаксису требуется хотя бы один оператор;
- при необходимости пометить фигурную скобку. Синтаксис языка программирования С-51 требует, чтобы после метки обязательно следовал оператор. Фигурная же скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор.

Пример:

```
int main( )
{...
        { if (...) goto a;      // переход на скобку
                { …
                }
a:;     }
        return 0;
        }
```

27.2 Оператор-выражение

Любое выражение, которое заканчивается точкой с запятой, является оператором.

Выполнение оператора-выражения заключается в вычислении выражения. Полученное значение никак не используется, поэтому, как правило, такие выражения вызывают побочные эффекты. Заметим, что вызвать функцию, не возвращающую значение, можно только при помощи оператора-выражения. Правила вычисления выражений были сформулированы выше.

Примеры:

++ i; – этот оператор представляет выражение, которое увеличивает значение переменной i на единицу.

a=cos(b*5); – этот оператор представляет выражение, включающее в себя операции присваивания и вызова функции.

a(x, y); – этот оператор представляет выражение, состоящее из вызова функции.

27.3 Составной оператор

Составной оператор, часто называемый блоком, представляет собой несколько операторов и объявлений, заключенных в фигурные скобки:

'{' [Список объявлений)
[Список операторов)
'}'

В конце составного оператора точка с запятой не ставится.

Выполнение составного оператора заключается в последовательном выполнении составляющих его операторов. Он используется в условных операторах и операторах цикла для того, чтобы выполнить несколько операторов.

Пример:

```
int main( )
{       int q,b;
        double t,d;
        ...
        if (...)
           { int e,g;
             float f,q;
             …
            }
            ...
            return 0;
}
```

Переменные *e, g, f, q* будут уничтожены после выполнения составного оператора. Переменная *q* является локальной в составном операторе, т. е. она никоим образом не связана с переменной *q* объявленной в начале функции main с типом int.

28 Программирование на языке С-51: условные операторы (if, switch)

28.1 Оператор if

Формат оператора:

"if (" Выражение')' Оператор 1';' ["else" Оператор2';']

Выполнение оператора if начинается с вычисления выражения.

Далее выполнение происходит по следующей схеме:

1) если Выражение истинно (т. е. отлично от 0), то выполняется Оператор1;
2) если Выражение ложно (т. е. равно 0), то выполняется Оператор2;
3) если Выражение ложно и отсутствует Оператор2, то выполняется следующий за if оператором.

После выполнения оператора if значение передается на следующий оператор программы, если последовательность выполнения операторов программы не будет принудительно нарушена использованием операторов перехода.

Пример:

```
if (i < j)
   i++;
  else
{j=i-3;
 i++;
}
```

Этот пример иллюстрирует также и тот факт, что на месте Оператора1, так же как и на месте Оператора2, могут находиться сложные конструкции.

Допускается использование вложенных операторов if. Оператор if может быть включен в конструкцию if или в конструкцию else другого оператора if. Чтобы сделать программу более читабельной, рекомендуется группировать операторы и конструкции во вложенных операторах if, используя фигурные скобки (образуя составной оператор). Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово else с наиболее близким ключевым словом if, для которого нет else.

Примеры:

```
int main ( )
  {int t=2, b=7, r=3;
         if(t>b)
        {if(b<r)
        r=b;
        }
        else
        r=t;
   return (0);
  }
```

В результате выполнения этой программы переменная r станет равной 2. Если же в программе опустить фигурные скобки, стоящие после оператора if, то программа будет иметь следующий вид:

```
int main( )
 {int t=2, b=7, r=3;
       if(t>b)
       if (b<r)
       t=b;
       else
       r=t ;
  return (0);
}
```

В этом случае r получит значение, равное 3, т. к. ключевое слово else относится ко второму оператору if, который не выполняется, поскольку не выполняется условие, проверяемое в первом операторе if.

Следующий фрагмент иллюстрирует вложенные операторы if:

```
char ZNAC;
int x,y,z;
   :
if (ZNAC == '-') x = y - z;
else if (ZNAC == '+') x = y + z;
else if (ZNAC == '*') x = y * z;
else if (ZNAC == '/') x = y / z;
else ...
```

Из рассмотрения этого примера можно сделать вывод, что конструкции, использующие вложенные операторы if, выглядят довольно громоздко.

Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора switch.

Однако надо сказать, что использование этого оператора приводит к менее быстродействующим программам и объем программы возрастает по сравнению с предыдущим случаем использования условных операторов if.

28.2 Оператор switch

Оператор switch предназначен для организации выбора из множества различных вариантов. Формат оператора следующий:

```
“switch (“Выражение‘)’
{      [Объявление]
         ….
         [“case” Константное выражение1] ‘:’ [Список операторов1]
         [“case” Константное выражение2] ‘:’ [Список операторов2]
```

…
[“default:”[Список операторов]]
}

Выражение, следующее за ключевым словом switch в круглых скобках, может быть любым выражением, допустимыми в языке С-51, значение которого должно быть целым. Отметим, что можно использовать явное приведение к целому типу, однако необходимо помнить о тех ограничениях и рекомендациях, о которых говорилось выше.

Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора switch состоит из нескольких операторов, начинающихся с ключевого слова case с последующим константным выражением. Обычно в качестве константного выражения используются целые или символьные константы.

Все константные выражения в операторе switch должны быть различными. Кроме операторов, начинающихся с ключевого слова case, в составе оператора switch может быть один фрагмент, помеченный ключевым словом default. Он будет выполняться, если не выполнится ни одно из условий.

Список операторов может быть пустым либо содержать один или более операторов. Причем в операторе switch не требуется заключать последовательность операторов в фигурные скобки.

Отметим также, что в операторе switch можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом case, однако в объявлениях не должна использоваться инициализация.

Схема выполнения оператора switch следующая:

1) вычисляется выражение в круглых скобках;
2) вычисленное значение последовательно сравнивается с константными выражениями, следующими за ключевыми словами case;
3) если одно из константных выражений совпадает со значением выражения, то управление передается на оператор, помеченный соответствующим ключевым словом case;
4) если ни одно из константных выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом default. В случае отсутствия ключевого слова default управление передается на следующий оператор.

Отметим интересную особенность использования оператора switch: конструкция со словом default может быть не последней в теле оператора switch.

Все операторы между первым выполнившимся условием и концом оператора switch выполняются последовательно вне зависимости от выполнения последующих условий, если только в каком-либо из условий case выполнение оператора switch не будет прервано при помощи ключевого слова break. Поэтому программист должен сам позаботиться о выходе из оператора case, если необходимо, чтобы выполнялось только одно из условий оператора switch.

Например:

```
int i=2;
switch (i)
  {case 1: i += 2;
        case 2: i *= 3;
        case 0: i /= 2;
        case 4: i -= 5;
  default:       ;
  }
```

Выполнение оператора switch начинается со строки, помеченной case 2. Таким образом, переменная i получает значение, равное 6. Далее выполняется оператор, помеченный ключевым словом case 0, а затем — case 4, переменная i примет значение 3, а затем значение - -2. Пустой оператор, помеченный ключевым словом default, не изменяет значения переменной.

Рассмотрим, как выглядит ранее приведенный пример, в котором иллюстрировалось использование вложенных операторов if, если его переписать с использованием оператора switch.

```
char ZNAC;
int x,y,z;
switch (ZNAC)
{case '+' : x = y + z;  break;
 case '-' : x = y - z;   break;
 case '*' : x = y * z;   break;
 case '/' : x= y / z;  break;
default  : ;
}
```

29 Программирование на языке С-51: операторы цикла (for, while, do - while)

29.1 Оператор цикла for

Оператор for – это наиболее общий способ организации цикла. Он имеет следующий формат:

"for (" Выражение1';' Выражение2';' Выражение3 ')' Тело цикла';'

Выражение1 используется для задания начального значения переменных, управляющих циклом. Выражение2 определяет условие, при котором тело цикла будет выполняться. Выражение3 выполняется после каждого прохода тела цикла (после каждой итерации). Обычно в выражении 3 изменяются переменные, управляющие циклом.

Последовательность выполнения оператора цикла for:

1) Вычисляется Выражение 1;

2) Вычисляется Выражение2;

3) если значение Выражение2 отлично от нуля, то выполняется тело цикла, вычисляется Выражение3 и осуществляется переход к пункту 2, если Выражение2 равно нулю, то управление передается на оператор, следующий за оператором for.

Тут проверка условия всегда выполняется в начале цикла. Это означает, что тело цикла может ни разу не выполниться, если условие выполнения оператора цикла сразу будет ложным.

Пример использования оператора цикла for:

```
int main ()
{int i,b;
 for (i=1; i<10; i++)
   b=i*i;
  return 0;
}
```

В этом примере вычисляются квадраты чисел от 1 до 9.

Некоторые варианты применения оператора for повышают его возможности за счет использования сразу нескольких переменных, управляющих циклом.

Пример использования нескольких переменных в операторе цикла for:

```
int main()
{int top,bot;
 char string (100), temp;
 for(top=0, bot=100; top<bot; top++,bot--)
    {temp=string[top];
     string[bot]=temp;
    }
}
```

В этом примере, реализующем запись строки символов в обратном порядке, для управления циклом используются две переменные top и bot.

Тут на месте Выражения1 и Выражения3 используются несколько выражений, записанных через запятую и выполняемых последовательно.

В СКОБКАХ НА ЗАМЕТКУ ПРОСТО-(В этом же примере можно наглядно проследить за тем, как влияет выбор типа переменных на размер загрузочного файла. В приведенном примере для организации цикла использованы переменные типа int. В результате получился машинный код размером 59 байт. При замене типа этих же переменных на unsigned char размер кода сокращается до 41 байта. Эти же действия увеличивают быстродействие программы в полтора раза!)

В микроконтроллерах оператор цикла for используется для реализации бесконечного цикла, который необходим для непрерывной работы устройства. Организация такого цикла возможна при использовании пустого Выражения2. Иногда для реализации алгоритма работы устройства требуется при выполнении какого-либо условия выйти из бесконечного цикла. Для этого можно воспользоваться оператором break.

Пример реализации бесконечного цикла с возможностью выхода из него при помощи оператора for:

```
for (;;)
{ ...
  ... break;
  ...
}
```

Так как в языке программирования С присутствует пустой оператор, то и в качестве тела цикла оператора for также можно использовать пустой оператор. Такая форма оператора может быть использована для организации временных задержек или поиска.

Пример использования пустого оператора для поиска:

```
for(i=0;t[i]<10;i++);
```

В данном примере при завершении цикла переменная цикла i принимает значение номера первого элемента массива t, значение которого больше 10.

29.2 Оператор цикла while

Оператор while называется циклом с предусловием и имеет следующий формат:

"while ("Выражение')' Тело ';'

Этот оператор обычно приводит к более коротким и эффективным программам по сравнению с предыдущим оператором цикла, т. к. элементарно накладывается на машинные инструкции микроконтроллера.

В качестве выражения допускается использовать любое выражение языка С-51, а в качестве тела — любой оператор, в том числе пустой или составной операторы. Последовательность выполнения оператора while:

1) вычисляется Выражение;

2) если Выражение ложно, то выполнение оператора while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора while;

3) переход к пункту 1.

Оператор цикла вида

"for ("Выражение 1';' Выражение2';' Выражение3')' Тело ';'

может быть заменен оператором while следующим образом:

Выражение1;

"while ("Выражение2')'

'{' Тело

Выражение3;

'}'

Так же, как и при выполнении оператора цикла for, в операторе while вначале происходит проверка Выражения2. Поэтому оператор while удобно использовать в ситуациях, когда тело не всегда нужно выполнять.

Внутри операторов for и while можно использовать локальные переменные, которые должны быть объявлены с определением соответствующих типов.

29.3 Оператор цикла do-while

Оператор цикла do-while называется оператором цикла с проверкой условия после тела цикла и используется в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз. Формат оператора do-while имеет следующий вид:

"do» Тело «while» '('Выражение");"

Схема выполнения оператора do-while:

1) выполняется тело цикла (которое может быть составным оператором);

2) вычисляется Выражение;

3) если Выражение ложно, то выполнение оператора do-while заканчивается и выполняется следующий по порядку оператор. Если Выражение истинно, то выполняется переход к пункту 1.

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор break.

Операторы while и do-while могут быть вложенными.

Пример использования вложенных циклов:

```
int i,j,k;
...
i=0; j=0; k=0;
do {i++;
    j--;
    while(a[k]<i)k++;
}while(i<30&&j<-30);
```

Этот пример использования оператора do-while не является образцом для подражания, т. к. использование операции && заставляет компилятор создавать достаточно сложную программу выполнения логического выражения. Использование для той же цели оператора break приводит к более длинному исходному тексту программы.

При этом код программы получается более коротким и быстродействующим, т. к. в этом случае один оператор языка программирования С соответствует одной машинной команде микроконтроллера:

```
char i=0, j=0, k=0;
...
do{i++; j--;
    while(a[k]<i)k++;
    if(j>=-30)break;
}while(i<30);
```

30 Программирование на языке С-51: операторы перехода (break, continue, return, goto)

30.1 Оператор break

Оператор break обеспечивает прекращение выполнения самого внутреннего из объемлющих его операторов: switch, do, for или while. После выполнения оператора break управление передается оператору, следующему за прерванным оператором.

Пример реализации бесконечного цикла с возможностью выхода из него при помощи оператора for:

```
for (;;)
{ ...
  ... break;
  ...
}
```

Оператор break позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора switch путем передачи управления оператору, следующему за switch.

```
switch (a)
{
        case 0: P0=48; break;      // "0"
        case 1: P0=49;  break;     // "1"
        case 2: P0=50; break;      // "2"
        case 3: P0=51;  break;     // "3"
        default: break;
}
```

Фрагмент программы, в котором выставляется на порту P0 код цифры в переменной a для отображения на 7-сегментном LED-индикаторе. Использование оператора break приводит к тому, что в операторе switch выполняется всего одна строка. Если бы его не было, то выполнялась бы строка, соответствующего кейса a и следующие за ней до конца функции switch.

30.2 Оператор continue

Оператор continue используется только внутри операторов цикла, но в отличие от break, осуществляется не выход из цикла, а переход к следующему циклу. Формат записи оператора continue:

"continue;"

Пример использования оператора continue:

```
int main()
{int a,b;
 for (a=1,b=0; a<100; b+=a,a++)
   {if (b%2) continue;
```

```
        ... /* обработка четных сумм */
    }
 return 0;
}
```

Когда сумма чисел от 1 до a становится нечетной, оператор continue передает управление на очередную итерацию цикла for, не выполняя операторы обработки четных сумм.

Оператор continue, как и оператор break, прерывает самый внутренний из объемлющих его циклов.

30.3 Оператор возврата из функции return

Оператор return завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом данной функции. Формат оператора возврата из функции:

"return" [Выражение]';'

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызванной функции. Если выражение опущено, то возвращаемое значение не определено. Это используется в функция типа void. Выражение может быть заключено в круглые скобки, хотя их наличие не обязательно.

Если в какой-либо функции отсутствует оператор return, то передача управления в точку вызова происходит после выполнения последнего оператора вызываемой функции. При этом возвращаемое значение не определено. Если функция не должна возвращать значения (подпрограмма-процедура), то ее нужно объявлять с типом void.

Таким образом, оператор return используется либо для немедленного выхода из функции, либо для передачи в основную программу возвращаемого из функции значения.

Пример использования оператора return для возвращения результата работы функции суммирования двух переменных:

```
int sum (int a, int b)
{return (a+b);}
```

Функция sum объявлена с двумя формальными параметрами, a и b, int-типа и возвращает значение такого же типа, о чем говорит описатель, стоящий перед именем функции. Возвращаемое оператором return значение равно сумме фактических параметров.

Пример использования оператора return для выхода из подпрограммы-процедуры:

```
void prov (int a, double b)
{ double c;
if (a<3) return;
else if (b>10) return;
else { c=a+b;
      if ((2*c-b)==11) return;
```

```
}
}
```

В этом примере оператор return используется для выхода из функции в случае выполнения одного из проверяемых условий.

30.4 Оператор безусловного перехода goto

Использование оператора безусловного перехода goto в практике программирования на языке С-51 настоятельно не рекомендуется, т. к. затрудняет понимание программ и возможность их модификаций. В то же самое время алгоритм любой степени сложности может быть построен при использовании оператора условного перехода if и операторов циклов while и do-while. Оператор безусловного перехода goto может быть использован только в случае крайней необходимости.

Формат этого оператора записывается в следующем виде:

"goto" Имя метки';'

...

Имя метки':' Оператор';'

Оператор goto передает управление на оператор, помеченный меткой имя метки. Помеченный оператор должен находиться в той же функции, что и goto, а используемая метка должна быть уникальной, т. е. одно имя метки не может быть использовано для разных операторов программы, имя метки — это идентификатор.

Любой оператор в составном операторе может иметь свою метку. Используя goto, можно передавать управление внутрь составного оператора. Но нужно быть осторожным при входе в составной оператор, содержащий объявления переменных с инициализацией, т.к. объявления располагаются перед выполняемыми операторами и значения объявленных переменных при таком переходе будут не определены.

31 Программирование на языке С-51: определения и вызов функций

Функция — это совокупность операторов и объявлений локальных переменных, обычно предназначенная для решения определенной задачи.

Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе, написанной на языке программирования С-51, должна быть функция с именем main (главная функция), именно с нее, в каком бы месте программы она не находилась, начинается выполнение программы.

При вызове функции ей при помощи аргументов (формальных параметров) могут быть переданы некоторые значения (фактические параметры), используемые во время ее выполнения. Функция может возвращать значение (одно!). Это возвращаемое значение и является результатом выполнения функции, который после выполнение программы заносится в переменную, стоящую в левой части выражения приравнивая, в правой части которого происходит вызов функции. Допускается также использовать функции, не имеющие аргументов, и функции, не возвращающие никаких значений (подпрограммы-процедуры).

С использованием функций в языке программирования С-51 связаны три понятия: определение функции (описание действий, выполняемых функцией в виде операторов), объявление функции (задание формы обращения к функции) и вызов функции.

Определение функции задает тип возвращаемого значения, имя функции, типы и число формальных параметров, а также объявления локальных переменных и операторы, называемые телом функции, и определяющие выполняемые ею действия. В определении функции также может быть задан класс памяти.

Пример определения подпрограммы-функции:

```
int rus (unsigned char r)
{if (r>='A' && c<=' ')
      return 1;
      else
      return 0;
}
```

В данном примере определена функция с именем *rus*, имеющая один параметр с именем *r* и типом unsigned char. Функция возвращает целое значение, равное 1, если параметр функции является буквой русского алфавита, или 0 в противном случае.

В языке программирования С-51 требуется, чтобы определение функции обязательно предшествовало ее вызову. Определения используемых функций могут находиться перед определением функции main в одном с нею файле или в другом файле.

Если же по каким-либо причинам требуется вызвать функцию раньше ее фактического определения или функцию, определение которой находится в другом файле, то до вызова нужно поместить объявление (прототип) вызываемой функции. Это позволит

компилятору проверить соответствия типов передаваемых фактических параметров типам формальных параметров функции.

Объявление функции имеет такой же вид, что и определение, с той лишь разницей, что тело функции (исполняемые операторы) отсутствует, а имена формальных параметров могут быть опущены. Для функции, определенной в последнем примере, прототип может быть представлен в виде:

```
int rus (unsigned char r);
```

или

```
int rus (unsigned char);
```

В программах, написанных на языке программирования С-51, широко используются так называемые библиотечные функции, т. е. функции, предварительно разработанные и записанные в библиотеки. Прототипы библиотечных функций находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования, и включаются в программу с помощью директивы #include.

Если объявление функции не задано, то по умолчанию строится прототип функции на основе анализа первой ссылки на функцию, будь то вызов или определение функции. Однако такой прототип не всегда согласуется с последующим определением или вызовом функции. Рекомендуется всегда задавать прототип функции. Это позволит компилятору либо выдавать диагностические сообщения при неправильном использовании функции, либо корректным образом регулировать несоответствие аргументов, устанавливаемое при выполнении программы.

В соответствии с синтаксисом языка С-51, определение функции имеет следующую форму:

```
[Спецификатор класса памяти] [Спецификатор типа] Имя функции
'(' [Список формальных параметров]')'
'{ 'Тело функции' }'
```

Необязательный Спецификатор класса памяти задает класс памяти функции, который может быть static или extern. Подробно классы памяти будут рассмотрены в следующем разделе.

Спецификатор типа функции задает тип возвращаемого значения, который может быть любым. Если спецификатор типа не задан, то предполагается, что функция возвращает значение типа int.

Функция не может возвращать массив или функцию, но может возвращать указатель на любой тип, в том числе и на массив, и на функцию.

Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу в объявлении этой функции.

Функция возвращает значение, если ее выполнение заканчивается оператором return, содержащим некоторое выражение. Указанное выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения и возвращается в точку вызова функции в качестве результата.

Если оператор return не содержит выражения или выполнение функции завершается после выполнения последнего ее оператора (без выполнения оператора return), то возвращаемое значение не определено. Для функций, не использующих возвращаемое значение, в описателе типа должен быть использован тип void, указывающий на отсутствие возвращаемого значения. Если функция определена как возвращающая некоторое значение, а в операторе return при выходе из нее отсутствует выражение, то поведение вызывающей функции после передачи ей управления может быть непредсказуемым, поэтому транслятор с языка программирования проверяет такую ситуацию и выдает сообщение об ошибке.

Список формальных параметров — это последовательность объявлений формальных параметров, разделенная запятыми. Формальные параметры — это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений соответствующих фактических Параметров. Список формальных параметров может заканчиваться запятой (,) или запятой с многоточием (….). Это означает, что число аргументов функции переменно. Однако предполагается, что функция имеет, по крайней мере, столько обязательных аргументов, сколько формальных параметров задано перед последней запятой в списке параметров. Такой функции может быть передано большее число аргументов, но для дополнительных аргументов не проводится контроль типов.

Если функция не использует параметров, то наличие круглых скобок обязательно, а вместо списка параметров рекомендуется указать слово void.

Порядок и типы формальных параметров должны быть одинаковыми в определении функции и во всех ее объявлениях. Типы фактических параметров при вызове функции должны быть совместимы с типами соответствующих формальных параметров. Формальный параметр может иметь любой основной тип, а также быть структурой, объединением, перечислением, указателем или массивом. Параметр, тип которого не указан, считается имеющим тип int.

Для формального параметра можно задавать класс памяти register, при этом для величин целого типа спецификатор типа можно опустить.

Идентификаторы формальных параметров используются в теле функции в качестве ссылок на переданные при вызове значения. Они не могут быть переопределены в блоке, образующем тело функции, но могут быть переопределены во внутреннем блоке внутри тела функции. Несоответствие типов фактических аргументов и формальных параметров может быть причиной неверной интерпретации.

Тело функции — это составной оператор, содержащий операторы, определяющие действие функции.

Все переменные, объявленные в теле функции без указания класса памяти, являются локальными. По умолчанию они считаются автоматическими, но могут быть и статическими, если использован модификатор static. При этом значение, записанное в эту переменную, сохраняется даже при выходе из функции и последующем входе в нее. При вызове функции в стандартном языке программирования С автоматическим локальным

переменным отводится память в стеке и, если указано, производится их инициализация. В языке программирования С-51 для локальных переменных выделяются ячейки внутренней памяти данных. При этом для различных функций используются одни и те же ячейки памяти. Это сделано из соображений экономии внутренней памяти.

Иногда при написании программы требуется вызов функции самой из себя или функция может вызываться из основной программы и подпрограммы обслуживания прерывания. В стандартном языке программирования С это не создает проблем, ведь там локальные переменные хранятся в стеке. В языке программирования С-51 для таких функций следует применять атрибут reentrant. При его использовании локальные переменные будут располагаться в стеке. При этом стек будет размещаться в зависимости от вида принятой для компиляции модели памяти (small, compact, large) в области памяти data, pdata, xdata соответственно. Пример использования атрибута reentrant:

```
int calc (char i, int reentrant {
int x;
x = table [i] ;
return (x * b);
}
```

При вызове функции производится инициализация локальных переменных. Затем управление передается первому оператору тела функции и начинается ее выполнение, продолжающееся до тех пор, пока не встретится оператор return или последний оператор тела функции. Управление при этом возвращается оператору, следующему за точкой вызова, а локальные переменные становятся недоступными. При новом вызове функции для автоматических локальных переменных память распределяется вновь, и поэтому старые значения таких переменных теряются.

Параметры функции могут рассматриваться как локальные переменные, для которых при вызове функции выделяется память и производится инициализация значениями фактических параметров, поэтому в теле функции нельзя изменить значения переменных вызывающей программы путем изменения значений параметров функции. При выходе из функции значения этих переменных теряются. Однако если в качестве параметра передать указатель на некоторую переменную, то в функции можно будет изменить значение этой переменной.

Пример попытки неправильного использования параметров функции:

```
/* Неправильное использование параметров */
void change (int x, int y)
{ int k=x;
      x=y;
      y=k;
}
```

В данной функции значения переменных x и y, являющихся формальными параметрами, меняются местами, но поскольку эти переменные существуют только внутри функции change, значения фактических параметров, переданные при вызове

функции, останутся неизменными. Для того чтобы менялись местами значения фактических аргументов, нужно использовать функцию, подобную приведенной в следующем примере:

```
/* Правильное использование параметров */
  void change (int *x, int *y)
{ int k=*x;
      *x=*y;
      *y=k;
}
```

При вызове такой функции в качестве фактических параметров необходимо использовать не значения переменных, а их адреса, как показано в следующем примере:

```
change (&a,&b);
```

Если требуется вызвать функцию до ее определения в рассматриваемом файле, или с определением, находящимся в другом исходном файле, то необходимо предварительно объявить эту функцию.

Прототип — это явное объявление функции, которое предшествует ее определению. Тип возвращаемого значения при объявлении функции должен соответствовать типу возвращаемого значения в ее определении.

В отличие от определения функции, в прототипе за заголовком сразу же следует точка с запятой, а тело функции отсутствует.

Прототип функции необходимо задавать в следующих случаях:

— Функция возвращает значение типа, отличного от int.

— Требуется проинициализировать некоторый указатель на функцию до того, как эта функция будет определена.

Объявление (запись прототипа) функции производится в следующем формате:

[Спецификатор класса памяти] [Спецификатор типа] Имя функции '(' [Список формальных параметров]')' [',' Список имен функций]';'

Если прототип не задан, а встретился вызов функции, то строится неявный прототип из анализа формы вызова функции. Тип возвращаемого значения создаваемого прототипа — int, а список типов и числа параметров функции формируется на основании типов и числа фактических параметров, используемых при данном вызове.

Учитывая, что построенный прототип функции может не совпасть с определением, лучше не надеяться на автоматическое построение прототипа, а объявлять его явным образом.

Наличие в прототипе полного списка типов параметров позволяет выполнить проверку соответствия типов фактических параметров при вызове функции типам формальных параметров, и, если необходимо, выполнить соответствующие преобразования.

Вызов функции имеет следующий формат:

Адресное выражение '(' [Список выражений]')'

Поскольку синтаксически имя функции является адресом начала тела функции, в качестве обращения к функции может быть использовано Адресное выражение (в том числе и имя функции или разадресация указателя на функцию), имеющее значение адреса функции.

Список выражений представляет собой список фактических параметров, передаваемых в функцию. Этот список может быть и пустым, но наличие круглых скобок обязательно.

Фактический параметр может быть величиной любого основного типа, структурой, объединением, перечислением или указателем на объект любого типа. Массив и функция не могут быть использованы в качестве фактических параметров, но можно использовать указатели на эти объекты.

Выполнение вызова функции происходит следующим образом:

1. Вычисляются выражения в Списке выражений и подвергаются обычным арифметическим преобразованиям. Затем, если известен прототип функции, тип полученного значения сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо формируется сообщение об ошибке. Число выражений в списке должно совпадать с числом формальных параметров, если только функция не имеет переменного числа параметров. В последнем случае проверке подлежат только обязательные параметры. Если в прототипе функции указано, что ей не требуются параметры, а при вызове они указаны, формируется сообщение об ошибке.

2. Происходит присваивание значений фактических параметров соответствующим формальным параметрам.

3. Управление передается на первый оператор функции.

4. Выполнение оператора return в теле функции возвращает управление и, возможно, значение в вызывающую функцию. При отсутствии оператора return управление возвращается после выполнения последнего оператора тела функции, а возвращаемое значение не определено.

Адресное выражение, стоящее перед скобками, определяет адрес вызываемой функции. Это значит, что функция может быть вызвана через указатель на функцию.

Пример объявления переменной указателя на функцию:

```
int (*fun)(int x, int *y);
```

Здесь объявлена переменная fun как указатель на функцию с двумя параметрами типа int и указателем на int. Сама функция должна возвращать значение типа int. Круглые скобки, содержащие имя указателя fun и признак указателя *, обязательны, иначе запись

```
int *fun (int x, int *y);
```

будет интерпретироваться как объявление функции fun, возвращающей указатель на int.

Вызов функции при помощи указателя fun возможен только после инициализации этого указателя. Вызов самой функции при этом будет выглядеть следующим образом:

```
(*fun)(i,&j);
```

В этом выражении для получения адреса функции, на которую ссылается указатель fun, используется операция *.

Указатель на функцию может быть передан в качестве параметра функции. При этом разадресация происходит во время вызова функции, на которую ссылается указатель на функцию. Присвоить значение указателю на функцию можно в операторе присваивания, употребив имя функции без списка параметров.

Пример:

```
double (*fun1)(int x, int y);
double fun2(int k, int l);
    fun1=fun2;          /* инициализация указателя на функцию */
    (*fun1) (2,7);      /* обращение к функции                */
```

В рассмотренном примере указатель на функцию fun1 описан как указатель на функцию с двумя параметрами, возвращающую значение типа double, и также описана функция fun2. В противном случае т. е. когда указателю на функцию присваивается адрес функции, описание которой отличается от описания указателя, произойдет ошибка.

Рассмотрим пример использования указателя на функцию в качестве параметра функции, вычисляющей производную от cos(x).

```
double proiz (double x, double dx, double (*f) (double x) );
double fun(double z);
int main()
{
double  x;  /* точка вычисления производной*/
double dx;  /*      приращение */
double z;           /*      значение производной */
scanf("%f,%f",&x,&dx);  /* ввод  значений  x  и  dx */
z = proiz (x, dx, fun);     /*      вызов  функции */
printf("%f",z);     /* печать значения производной*/
return 0;
}
double proiz (double x, double dx, double (*f) (double z))
/* функция вычисляющая производную * /
double xk, xk1, pr;
xk=fun(x);
xkl=fun(x+dx) ;
pr=(xkl/xk-le0)*xk/dx;
return pr;
}
double fun (double z)
/* функция, от которой вычисляется производная:     */
return (cos(z));
```