- Примеры использования директив

***Директива EQU** позволяет назначать имена переменных и констант. Теперь можно назначить переменной адрес в одном месте и пользоваться идентификатором переменной во всей программе. Правда за использование идентификатора именно в качестве переменной отвечает программист. Тем не менее, если в процессе написания программы потребуется изменить адрес переменной, это можно сделать в одном месте программы, а не просматривать всю программу, разбираясь, является ли в данной конкретной команде число 10 константой, адресом ячейки ли количеством повторов в цикле. Все необходимые изменения сделает сам транслятор. Пример назначения переменных приведен на примере описания интерфейса подключения жидкокристаллического (ЖКИ) к микроконтроллеру. Достоинство такого описания заключается в простом переносе программы на другую, сходную систему, имеющую другое подключение индикатора:*

```
DispDat     EQU     P0      ; Шина данных ЖКИ на порте P0.
                            ;
RDS         EQU     P1.2    ; Сигнал чтение команды ЖКИ подключен к
                            ; линии P1.2.
RW          EQU     P1.1    ; Сигнал выбора записи/чтения ЖКИ подключен к
                            ; линии P1.1
E           EQU     P1.0    ; Сигнал строб синхронизации ЖКИ подключен к
                            ; линии P1.0
BUSY        EQU     P0.7    ; Сигнал занятости ЖКИ подключен к линии P0.7
                            ;
SetDD_Adr   EQU     80h     ; Адрес регистра данных/адреса внутри ЖКИ
FuncSet     EQU     20h     ; Команда ЖКИ установки функций
_8bit       EQU     10h     ; Режим 8 бит интерфейс ЖКИ
_2line      EQU     8       ; Режим 2 строчного ЖКИ
```

Как видно на приведенном примере, использование идентификаторов значительно повышает понятность программы, так как в названии переменной отображается функция, за которую отвечает данная переменная.

Один раз назначенный идентификатор уже не может быть изменен в дальнейшем и при повторной попытке назначения точно такого же имени идентификатора будет выдано сообщение об ошибке.

***Директива SET**. Если требуется в различных местах программы назначать одному и тому же идентификатору различные числа, то нужно пользоваться директивой set. Использование этой директивы полностью идентично использованию директивы equ, поэтому иллюстрироваться примером не будет.*

*Константы, назначаемые директивой equ, могут быть использованы только в одной команде. 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, а хранятся в памяти программ микроконтроллера. Для занесения констант в память программ микроконтроллера используются директивы **DB и DW.***

Рассмотрим применение данных директив на примере. Пусть у нас имеется цифровой семисегментный индикатор с общим катодом, подключенный к микропроцессорной системе согласно рис. 15:

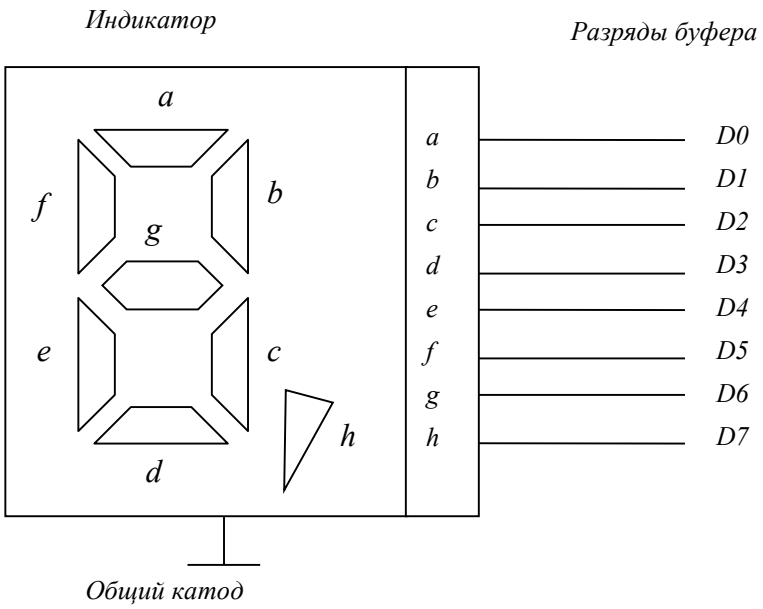

Рисунок 15 – Пример подключения индикатора к МПС

При установке соответствующего бита разряда выходного буфера в логическую «1», зажигается соответствующий сегмент. Например, для отображения числа 1 необходимо установить биты D1 и D2, соответствующие сегментам «b» и «c». Тогда, для вывода чисел на такой индикатор, необходимо выполнить программный дешифратор из числа в код семисегментного индикатора. Можно выполнить табличный перевод, напоминающий поиск значения функции по таблицам Брадиса. Непосредственно данные для свечения на индикаторе задаются директивой ***DB****:*

```
;Подпрограмма перевода числа в код семисегментного индикатора
;Вход: A – ДД число формата 0X
;Выход: A – семисегментный код числа
;-----------------------------------------------------------------------
Decode:
          MOV   DPTR, #Table    ; Загрузка адреса таблицы.
          MOVC  A,@A+DPTR       ; Чтение записи со смещением в A.
          RET
;
Table:    ;     hgfedcba
          DB    00111111b       ; Символ '0'
          DB    00000110b       ; Символ '1'
          DB    01011011b       ; Символ '2'
          DB    01001111b       ; Символ '3'
          DB    01100110b       ; Символ '4'
          DB    01101101b       ; Символ '5'
          DB    01111101b       ; Символ '6'
          DB    00000111b       ; Символ '7'
          DB    01111111b       ; Символ '8'
          DB    01101111b       ; Символ '9'
```

Эта же директива позволяет легко записывать надписи, 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 надписей в память программ микроконтроллера приведен ниже:*

```
Decode:
MOV     R7,#EndNadp-NadpSvjazUst   ;В R7 заносим число символов.
MOV     DPTR,#NadpSvjazUst         ;Подготовить к передаче первый символ
PutNextChar:
CLR     A
```

```
MOVC    A,@A+DPTR                   ;Читаем очередной символ
INC     DPTR
CALL    PutChar                     ;Отправляем символ на передачу
DJNZ    R7,PutNextChar              ;Последний символ?
RET                                 ;Да, возврат из подпрограммы

NadpSvjazUst: DB 'Связь установлена',10,13
EndNadp:
```

Директива DW *позволяет заносить в память программ двухбайтные числа. В этой директиве, как и в директиве* ***DB*** *числа можно заносить через запятую. Пример листинга фрагмента:*

```
0023 0001    294          DW 1,2,0abh,'a','QW'
0025 0002
0027 00AB
0029 0061
002B 5157
```

Иногда требуется расположить команду по определенному адресу. Наиболее часто это требуется при использовании прерываний, когда первая команда программы-обработчика прерываний должна быть расположена точно на векторе прерывания. Это можно сделать, используя команду NOP для заполнения промежутков между векторами прерывания, но лучше воспользоваться директивой ***ORG****.*

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 любому адресу. Пример использования директивы ORG для размещения подпрограмм обработки прерываний на векторах прерываний показан ниже:*

```
Reset:
        LJMP Main           ; Переход на начало основной программы
                            ; Прерывание переполнения таймера 0.
        ORG 0bh             ; Вектор прерывания таймера 0.
        LJMP IntT0          ; Переход на обработчик прерывания T/C0.
                            ; Прерывание последовательного порта.
        ORG 23h             ; Вектор прерывания последовательного порта.
        LJMP IntSerPort
                            ;Для частоты кварцевого резонатора 12МГц.
IntT0:
        Mov TL0, #LOW(-(Fosc/12)*10-2)    ;Настройка таймера
        Mov TH0, #HIGH(-(Fosc/12)*10-2)   ;на 10мкс.
        Reti
                            ;Начало основной программы микроконтроллера.
Main:   Mov SP,#VershSteka    ;Настройка указателя стека.
        Call Init           ;Выполнить п/п инициализации микроконтроллера.
;-------------------------------------------------------------------------
```

Необходимо отметить, что при использовании этой директивы возможна ситуация, когда программист приказывает транслятору разместить новый код программы по уже написанному месту, что приводит к неверной трансляции программы.

Директива 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. Пример использования директивы USING для подпрограммы обслуживания прерываний от таймера 0 приведен ниже:*

```
_code segment code
CSEG AT 0bh                             ; Вектор прерывания от T/C0.
        Jmp IntT0

rseg _code
```

```
        USING 2
IntT0:
        Push PSW                          ; Сохраняем регистры в стек.
        Push ACC
        Mov PSW,#00010000b                ;Включаем банк 2 РОН.
        Mov TL0, #LOW(-(Fosc/12)*10-2)    ;Настройка таймера
        Mov TH0, #HIGH(-(Fosc/12)*10-2)   ;на 10мкс.
        Pop ACC
        Pop PSW
        RETI
```

Директива CALL. *В системе команд микроконтроллера MCS-51 используется три команды безусловного перехода. Выбор конкретной команды зависит от расположения ее в памяти программ, однако программист обычно этого не знает. В результате во избежание ошибок приходится использовать самую длинную команду LJMP. Это приводит к более длинным программам и к дополнительной нагрузке на редактор связей. Транслятор сам может подобрать наилучший вариант команды безусловного перехода. Для этого вместо команды микроконтроллера следует использовать директиву call.*

- Реализация подпрограмм

При написании программ часто при реализации алгоритма работы устройства приходится повторять одни и те же операторы (например операторы, работающие с параллельным или последовательным портом). Было бы неплохо использовать один и тот же участок кода, вместо того, чтобы повторять одни и те же операторы несколько раз.

Участок программы, к которому можно обращаться из различных мест программы для выполнения некоторых действий называется ***подпрограммой****.*

Проблема, с которой приходится сталкиваться при многократном использовании участков кодов – это в какое место памяти программ возвращаться после завершения подпрограммы. Обращение к подпрограмме производится из нескольких мест основной программы. Описанную ситуацию иллюстрирует рис. 16. На этом рисунке изображено адресное пространство микроконтроллера. Младшие адреса адресного пространства на этом рисунке находятся в нижней части.

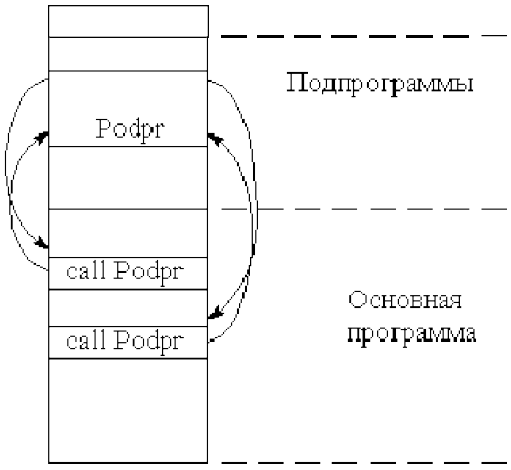

Рисунок 16 – Вызов подпрограммы и возврат к выполнению основной программы

Для обращения к подпрограмме и возврата из нее в систему команд микропроцессоров вводят специальные команды. В микроконтроллерах семейства MCS-51 это команды LCALL, ACALL для вызова подпрограммы и команда RET для возврата из подпрограммы. Эти команды не только осуществляют передачу управления на указанный адрес, но и запоминают адрес команды, следующей за командой вызова подпрограммы в стековой памяти. Команда возврата из подпрограммы RET передает управление команде, адрес которой был запомнен командой вызова подпрограммы. Пример использования подпрограммы на языке программирования ASM-51 приведен ниже:

```
MOV G_Per,#56        ; Передать 56 через последовательный порт.
CALL PeredatByte
...
MOV G_Per,#49        ; Передать 49 через последовательный порт.
Call PeredatByte
...
;---------------------------------------------------------------------*
; Подпрограмма передачи байта по последовательному порту
;---------------------------------------------------------------------*
PeredatByte:
JB TI,$              ; Ожидание конца передачи предыдущего байта.
MOV SBUF,G_Per       ; Передача байта
RET
```

Ни в коем случае нельзя попадать в подпрограмму любым способом кроме команды вызова подпрограммы CALL. В противном случае команда возврата из подпрограммы передаст управление случайному адресу. По этому адресу могут быть расположены данные, которые в этом случае будут интерпретированы как программа, или обратиться к внешней памяти, откуда будут считываться случайные числа.

*Очень часто требуется из одной подпрограммы обращаться к другой подпрограмме. Такое обращение к подпрограмме называется вложенным. Количество вложенных подпрограмм называется уровнем вложенности подпрограмм. Максимально допустимый уровень вложенности подпрограмм определяется количеством ячеек стековой памяти. Логически эти ячейки памяти организованы так, чтобы считывание последнего записанного адреса производилось первым, а первого записанного адреса производилось последним (буфер LIFO). Такая логическая организация формируется специальным счетчиком. Этот счетчик, как было указано выше, называется указателем стека **SP**. Ячейка памяти, в которую в данный момент может быть записан адрес возврата из подпрограммы, называется **вершиной стека**. Количество ячеек памяти, предназначенных для организации стека, называется глубиной стека. Последняя ячейка памяти, в которую можно производить запись называется **дном стека**. Логическая организация стека приведена на рис. 17.*

В микроконтроллерах семейства MCS-51 при занесении информации в стек содержимое указателя стека увеличивается (стек растет вверх), поэтому стек размещается в самой верхней части памяти данных. 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;       Объем внутреннего ОЗУ I8051 – 128 байт.
MOV SP,#DnoSteka-28;    Установим глубину стека 28 байт.
```

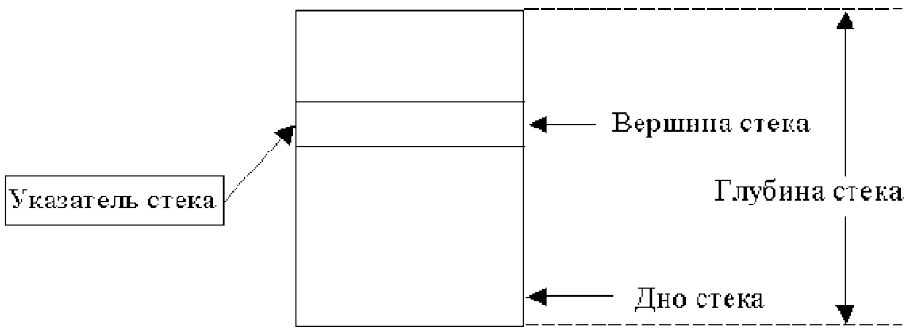

Рисунок 17 – Организация стека в памяти данных микропроцессора

Кроме содержимого программного счетчика часто требуется запоминать содержимое внутренних регистров и флагов процессора, локальных переменных подпрограммы. Стек оказался удобным средством и для этой задачи. Сохранение локальных переменных в стеке позволило осуществлять вызов подпрограммы самой из себя (реализовывать рекурсивные алгоритмы). Это привело к введению в систему команд специальных команд работы со стеком. В микроконтроллерах семейства MCS-51 это команды PUSH и POP. Использование этих команд показывается на следующем примере:

```
Podprogramma:
PUSH PSW          ; Сохраняем используемые в подпрограмме
PUSH ACC          ; регистры в стеке
PUSH R0
...               ; Код самой подпрограммы.
POP R0            ; Извлекаем сохраненные регистры из стека
POP ACC           ; в обратном порядке.
POP PSW
RET               ; Выход из подпрограммы.
```

В приведенном выше примере передачи байта через последовательный порт, сам байт передается в подпрограмму через глобальную переменную G_Per. Однако программа будет эффективнее при использовании подпрограммы с параметрами. Мы знаем, что параметр подпрограммы – это локальная переменная. В этом случае могут значительно снизиться требования к памяти данных. Для размещения локальных переменных лучше всего использовать внутренние регистры процессора. На языке ASM51 для передачи параметра размерностью один байт обычно используется аккумулятор.

Если в подпрограмму нужно передать двухбайтовое значение, то в качестве параметра подпрограммы используется пара регистров (обычно регистры R6-старший байт и R7-младший байт). Пример программы, передающей в подпрограмму двухбайтовое число, написанной на языке программирования ASM-51 приведен ниже:

```
...
;Пример передачи в подпрограмму двухбайтового числа.
Mov R7,#56     ; Передача младшего байта.
Mov R6,#0      ; Передача старшего байта.
Call Podprog   ; Вызвать подпрограмму.
...
```

Если в подпрограмму нужно передать четырехбайтовое значение (это требуется для переменной, соответствующей типу long или float), то используются регистры R4...R7 (регистр R4 – старший байт):

```
...
;Пример передачи в подпрограмму четырехбайтового числа.
Mov R7, #56    ; Передача младшего байта.
Mov R6, #0
Mov R5, #0
Mov R4, #0     ; Передача старшего байта.
Call Podprog   ; Вызвать подпрограмму.
...
```

Регистры R0 и R1 обычно используются в качестве указателей обрабатываемых переменных, таких как строки или массивы. Если требуется, чтобы подпрограмма обработала значительный объем данных, то эти данные можно передать через параметр – указатель. В качестве указателя при обращении к внешней памяти данных или к памяти программ обычно используется регистр-указатель данных DPTR. Пример передачи в качестве параметра строки, написанный на языке программирования ASM-51 приведен ниже:

```
...
;Пример передачи в подпрограмму указателя данных.
Mov DPTR, #Stroka    ; Передача указателя на строку.
Call Podprog         ; Вызвать подпрограмму.
...
Stroka: DB           'Наша строка символов.'
```

При обращении к массивам или структурам, расположенным во внутренней памяти данных в качестве указателя адреса используется регистр R0 или R1:

```
...
;Пример передачи в подпрограмму указателя данных внутренней памяти.
Mov R0, #Array     ; Передача указателя на данные.
```

```
Call Obrabotka     ; Вызвать подпрограмму.
...
```

Часто требуется передавать результат вычислений из подпрограммы в основную программу. Для этого можно воспользоваться подпрограммой-функцией. Подпрограмма-функция возвращает вычисленное значение:

```
Mov A,X            ; Передать в подпрограмму значение X.
Call Sin           ; Вызов функции Y=sin(X)
Mov Y,A            ; Сохранение функции в переменной Y.
```

При этом переменные X и Y должны быть заранее определены директивой EQU.

- Использование сегментов в языке программирования ассемблер

Необходимо отметить, что даже когда мы не задумываемся о сегментах, в программе присутствует два сегмента: сегмент кода программы и сегмент данных. Если внимательно присмотреться к программе, то можно обнаружить, что 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 в довольно сложной программе, 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.

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

- *Программы.*
- *Стека.*
- *Переменных.*
- *Констант.*

Пример размещения сегментов в адресном пространстве памяти программ и внутренней памяти данных приведен на рис. 18:

Рисунок 18 – Разбиение памяти программ и данных на сегменты

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.

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 с другом в памяти микроконтроллера. Использование абсолют-

ных сегментов позволяет более гибко работать с памятью данных, так как 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.

***Директива BSEG** позволяет определить абсолютный сегмент во внутренней памяти данных с битовой адресацией 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. Использование битовых переменных позволяет значительно экономить внутреннюю память программ микроконтроллера. Пример использования директивы BSEG для объявления битовых переменных:*

```
BSEG AT 8               ; Сегмент начинается с 8 бита.
RejInd:     DBIT 1      ; Флаг режима индикации.
RejPriem:   DBIT 1      ; Флаг режима приема.
Flag:       DBIT 1      ; Флаг общего назначения.
```

***Директива 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. Пример использования директивы CSEG для размещения подпрограммы обслуживания прерывания от таймера 0:*

```
CSEG AT 0bh                         ; Вектор прерывания от T/C0.
IntT0:
Mov TL0, #LOW(-(Fosc/12)*10-2)      ;Настройка таймера
Mov TH0, #HIGH(-(Fosc/12)*10-2)     ;на 10мкс.
RETI
```

***Директива DSEG** позволяет определить абсолютный сегмент во внутренней памяти данных по определенному 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. Пример использования директивы DSEG для объявления байтовых переменных:*

```
DSEG AT 20h         ; Разместить сегмент с адреса, где возможна
                    ; битовая адресация (Для возможности одновременной
                    ; битовой и байтовой адресации).
RejInd:  DS 1       ; Переменная, отображающая состояние программ
                    ; обслуживания аппаратуры.
Rejim:   DS 1       ; Переменная режима работы.
Massiv:  DS 10      ; Десятибайтовый массив.
```

В приведенном примере предполагается, что он связан с примером, приведенном выше. Т. е. 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.

***Директива ISEG** позволяет определить абсолютный сегмент во внутренней памяти данных 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. Пример использования директивы ISEG для объявления байтовых переменных:*

```
ISEG AT 40h             ; Расположить сегмент в адресах от 40h.
```

```
Buffer:     DS 10      ; Переменная Buffer объемом 10 байт.
Stack:      DS 245     ; Глубина стека 245 байт.
```

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;

***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, BufferKlav

; Определение переменных
rseg _data
BufferKlav DS 8   ; Объем буфера клавиатуры 8 байт.
VershSteka:       ; Стек начинается здесь.
```

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**:*

```
_bits segment bit
PUBLIC KnIzm, strVv

; Определение битовых переменных
rseg _bits
KnIzm: DBIT 1     ; Флаг нажатия кнопки.
strVv: DBIT 1     ; Флаг введенной строки.
```

*В этом примере директива **SEGMENT** используется для объявления сегмента 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 приходится это делать вручную, а так как в процессе написания программы размер программных модулей постоянно меняется, то приходится вводить защитные области неиспользуемой памяти между программными модулями.

Пример использования перемещаемых сегментов в исходном тексте программы:

```
_code SEGMENT code

CSEG AT 0               ; Начало программы.
Reset:
      Jmp Main

; Начало основной программы микроконтроллера
RSEG _code
Main:
      Movx @DPTR,A
      Mov SP,#VershSteka      ; Настроить указатель стека на вершину
      Call Init               ; Выполнить п/п инициализации процессора.
;-------------------------------------------------------------------------
```

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 поместить 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.