



Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования  
«Московский государственный технический университет имени Н.Э. Баумана»  
(национальный исследовательский университет)  
(МГТУ им. Н.Э. Баумана)

**Методическое пособие для выполнения лабораторных работ и КР по  
дисциплине СП в 4-м семестре кафедры ИУ5  
по курсу  
"СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ"**

## Методическое пособие для выполнения лабораторных работ и КР по дисциплине СП в 4-м семестре кафедры ИУ5

### Предисловие

Данное методическое пособие предназначено для студентов кафедры ИУ5 обучающихся на 4-м семестре и изучающих дисциплину “Системное программирование”. Для выполнения лабораторных работ по курсу необходим определенный набор базовых знаний и умений.

В пособии предложены базовые понятия и технологии, необходимые для разработки, компиляции и сборки программ на языке Ассемблер. Даны основные сведения для работы в режиме командной строки, запуска командных файлов, некоторые необходимые сведения по системам программирования на языке Ассемблер применительно к объему курса, изучаемого по нашей специальности.

Для текста программ и документальных справок о них, полученных автоматически, с целью более легкого ориентирования в материале следующим образом будут цветом фона текста выделять его фрагменты:

Работа в режиме командной строки (серый):

```
> DIR
```

Вставка текста программ (светло - голубой):

```
0000          mycode SEGMENT 'code'
...
```

Вставка справок, полученных автоматически (желтый):

```
Turbo Assembler Version 3.1 Copyright (c) 1988, 1992 Borland International
Syntax:  TASM [options] source [,object] [,listing] [,xref]
```

## Содержание

Методическое пособие для выполнения лабораторных работ и КР по дисциплине СП в 4-м семестре кафедры ИУ5 .....	2
Предисловие .....	2
Содержание .....	3
Детальное содержание пособия .....	4
1. Работа в режиме командной строки.....	7
2. Пример простейшей программы .....	10
3. Ввод текста, компиляция, редактирование и отладка.....	15
4. Формальное описание синтаксиса в БНФ .....	24
5. Командные файлы и их применение.....	27
6. Утилиты VE и CHOICE .....	37
7. Работа с интегрированными файловыми менеджерами .....	42
10. Процедуры и их применение .....	45
16. Особенности работы некоторых команд для ЛР .....	51
18. Построение резидентных программ .....	62
21. Разработка блок-схем программ .....	92
22. Макросы и их применение.....	103
23. Коды их назначение и виды.....	122
24. Приложение 1 .....	<b>Ошибка! Закладка не определена.</b>

## Детальное содержание пособия

Методическое пособие для выполнения лабораторных работ и КР по дисциплине СП в 4-м семестре кафедры ИУ5 .....	2
Предисловие .....	2
Содержание .....	3
Детальное содержание пособия .....	4
1. Работа в режиме командной строки.....	7
1.1. Режим командной строки и его назначение .....	7
1.2. Разновидности командных интерпретаторов.....	7
1.3. Запуск и завершение работы режима командной строки .....	8
1.4. Запуск команд и программ в режиме командной строки .....	8
1.5. Получение справок о командах в режиме командной строки.....	9
2. Пример простейшей программы .....	10
2.1. Исходный текст программы на Ассемблере .....	10
2.2. Компиляция и листинг программ Ассемблера .....	11
2.3. Редактирование связей и его результаты .....	13
2.4. Исполнение программы Ассемблера и его результаты .....	14
2.5. Запоминание результатов в отдельном файле .....	14
2.6. Отладка программ, начальные сведения .....	14
3. Ввод текста, компиляция, редактирование и отладка.....	15
3.1. Ввод и редактирование текста программы Ассемблера .....	17
3.2. Запуск на компиляцию программы Ассемблера .....	18
3.3. Использование оболочки QC25 для работы на Ассемблере .....	<b>Ошибка!</b>
<b>Закладка не определена.</b>	
3.4. Запуск на редактирование связей программы Ассемблера.....	<b>Ошибка! Закладка не определена.</b>
3.5. Запуск на выполнение и отладку программы Ассемблера.....	20
3.6. Запуск на выполнение программы Ассемблера .....	22
3.7. Применение командных файлов для работы с Ассемблером .....	23
4. Формальное описание синтаксиса в БНФ .....	24
4.1. Назначение и состав языка БНФ .....	24
4.2. Правила, нетерминальные переменные и метасимволы.....	25
4.3. Примеры описания на БНФ .....	26
5. Командные файлы и их применение.....	27
5.1. Язык командных файлов .....	28
5.2. Разработка, ввод, отладка и исполнение командных файлов.....	29
5.3. Переменные командных файлов .....	30
5.4. Параметры командных файлов.....	31
5.5. Создание простого командного файла .....	33
5.6. Вложенные командные файлы .....	34
5.7. Примеры командных файлов.....	35
6. Утилиты VE и CHOICE .....	37
6.1. Утилита VE , назначение и использование .....	37

6.2. Утилита CHOICE, назначение и использование .....	39
6.3. Утилита GREP, назначение и использование .....	39
7. Работа с интегрированными файловыми менеджерами .....	42
7.1. Dos Navigator .....	42
7.2. Far manager .....	42
7.3. Windows/Total Commander .....	43
10. Процедуры и их применение .....	45
10.1. Процедуры, их назначение и применение .....	44
10.2. Описание процедур .....	44
10.3. Параметры процедур и вызов процедур .....	44
10.4. Вложенные вызовы процедур .....	46
10.5. Пример программы с процедурами .....	46
10.6. Листинг программы с процедурами .....	47
16. Особенности работы некоторых команд для ЛР .....	51
16.1. Команда XLAT .....	50
16.2. Команда MOVS, префикс REP и команды цепочек .....	51
16.3. MUL/IMUL и DIV/IDIV .....	55
16.4. Команды: TEST, CMP и AND .....	57
16.5. Команды: AAA , CBW и CWD .....	58
16.6. Команды: ADC и SBB .....	58
16.7. Команды: LOOP/LOOPE/LOOPNE .....	59
18. Построение резидентных программ .....	62
18.1. Курсовая работа по СП .....	61
18.2. Резидентные программы .....	61
18.3. Вектор прерываний .....	62
18.4. Структура резидентной программы (2-е части) .....	62
18.5. Понятие прерывания и назначение прерываний .....	63
18.6. Обработка прерываний в процессоре (программных и аппаратных) .....	64
18.7. Установка резидента .....	66
18.8. Расчет размера резидента .....	66
18.9. Запуск части инициализации .....	66
18.10. Определение и запоминание старого обработчика .....	67
18.11. Задание нового обработчика прерывания .....	67
18.12. Вызов старого обработчика прерывания .....	68
18.13. Пример простейшего резидента .....	68
18.14. Работа с вектором прерываний напрямую .....	71
18.15. Обработка в одном резиденте нескольких прерываний .....	71
18.16. Связь с резидентом из других программ .....	72
18.17. Выгрузка резидента .....	73
18.18. Разбор параметров командной строки .....	74
18.19. Контроль наличия резидента (другой способ) .....	74
18.20. Связь с резидентом с помощью клавиатуры .....	75
18.21. Освобождение памяти внешнее из отдельной программы .....	78
18.22. Завершение основной программы при проверке повторной загрузки .....	79
18.23. Проверка загрузки и выгрузки с помощью утилиты mem.exe .....	79
18.24. Описание данных и процедур резидента .....	80
18.25. Русификация сообщений резидента .....	80
18.26. ДЗ и ТЗ на КР .....	81

## Дисциплина “Системное программирование”

18.27. КР .....	81
18.28. 18.28 Автономная программа для выгрузки TSR.....	81
18.3 Пример резидентной программы .....	82
21. Разработка блок-схем программ .....	92
21.1. Назначение блок-схем программ .....	91
21.2. Элементы блок-схем программ .....	92
21.3. Примеры блок-схем программ .....	96
21.4. Оформление блок-схемы программы .....	101
21.5. Блок-схемы и описания данных .....	102
22. Макросы и их применение.....	103
22.1. Назначение макросов .....	103
22.2. Описание макросов.....	103
22.3. Параметры макросов и макровыводы .....	104
22.4. Параметры по умолчанию .....	104
22.5. Макрорасширения .....	105
22.6. Директивы написания макрокоманд.....	106
22.7. Циклическая компиляция .....	106
22.8. Локальные метки и переменные макрокоманд.....	108
22.9. Служебные символы макрокоманд.....	109
22.10. Вложенные вызовы макрокоманд .....	111
22.11. Условная компиляция .....	111
22.12. Директива EXITM.....	113
22.13. Отладка макрокоманд.....	113
22.14. Сравнение макросов и процедур .....	114
22.15. Макрокоманды – процедуры этапа компиляции .....	114
22.16. Примеры программы с макросами.....	114
23. Коды их назначение и виды.....	122
23.1. ASCII.....	123
23.2. Кодировка ANSI.....	124
23.3. Русификаторы .....	125
23.4. Перекодировка символов .....	125
23.5. SCAN – коды .....	126
23.6. Кодировка UNICODE .....	128
23.7. Программы для получения списка кодов.....	<b>Ошибка! Закладка не определена.</b>
25. Приложение 1 .....	<b>Ошибка! Закладка не определена.</b>

## Работа в режиме командной строки

Режим командной строки используется системными программистами и пользователями для выполнения команд операционной системы (ОС) и командных файлов (\*.bat). Кроме того, в режиме командной строки может быть запущена любая программа для операционных систем ОС (или в режиме эмулятора ДОС) и WINDOWS.

### 1.1. Режим командной строки и его назначение

Режим командной строки доступен в ОС для выполнения различного рода системных и пользовательских работ. Такой режим называют еще пакетным режимом работы программ. Он используется для следующих случаев:

- Выполнения системных работ, например, инсталляции ОС.
- При выполнении ремонтных работ в ОС, когда по каким-либо причинам работа ОС WINDOWS невозможна (например, при заражении вирусами).
- Запуска программ и команд ОС, результаты которых должны поступать на дисплей непосредственно.
- Запуска командных файлов (\*.bat).
- Запуска системных программ, работающих в этом режиме, например, утилиты MEM.EXE для просмотра состояния оперативной памяти и работающих программах.
- Выполнения многих других работ.

При запуске режима командной строки на экране появляется специальная подсказка (“>”), после которой можно вводить текст команд, например, введем команду **DATE** для получения текущей даты (↵ - означает нажатие клавиши **Enter**):

```
>DATE↵
```

Формат подсказки ввода можно изменить специальной командой **PROMPT**. После ввода команды, она будет выполнена, а строки будут сдвигаться вверх (перечень команд можно найти в литературе, в электронном справочнике – ЛР № 1, или получить автоматически в виде справки). Например, после ввода команды **DATE** (получения и изменения системной даты) мы получим:

```
>DATE
Текущая дата: Сб. 21.02.2009
Введите новую дату (дд-мм-гг) :
>22-02-2009↵
```



Символом “↵” здесь я обозначил клавишу “**Enter**”(и далее буду его использовать для этой цели), которую необходимо нажать для ввода команды. Если Вы введете в режиме командной строки полное название программы, работающей под WINDOWS, то она тоже будет выполняться, но в отдельном окне.

При выполнении командных файлов, команд ОС их результаты будут размещаться в окне командной строки. Нужно иметь в виду, что при сдвиге строк их число в окне ограничено, поэтому в верхней части экрана они будут пропадать. Число строк, которое можно просмотреть в оконном режиме командной строки (см. ниже) может превышать стандартный размер экрана (25 строк) и может настраиваться.

### 1.2. Разновидности командных интерпретаторов

В современных операционных системах (WIN 32) сохранилась возможность запуска нескольких модификаций командных интерпретаторов (или процессоров). Ранее, при запуске операционной системы ДОС командный интерпретатор запускался автоматически при старте ОС и дальнейшая работа проходила только в этом режиме. Различают следующие разновидности командных интерпретаторов:

- COMMAND.COM – 16-ти разрядный
- CMD.EXE – обновленный командный процессор с расширенными возможностями, эмулирующий работу в MS DOS.

- DOSBox – отдельный программный продукт для эмуляции ДОС.

Различают также режимы работы командных интерпретаторов в оконном режиме и в полноэкранном режиме. При переключении в полноэкранный режим (клавиши Ctrl+Enter) командный интерпретатор выполняет свои функции в более полном объеме. Возврат в оконный режим выполняется также с помощью клавиш - Ctrl+Enter.

### 1.3. Запуск и завершение работы режима командной строки

Режим командной строки может быть запущен явно и неявно. Неявный запуск выполняется автоматически при запуске программ и утилит, работающих в режиме ДОС. Явный запуск командного интерпретатора может быть выполнен так:

ПУСК(Start) => Выполнить...(Run...) => COMMAND.COM => OK

Или

ПУСК(Start) => Программы...(Programm...) => Стандартные => Командная строка

Можно создать ярлык для запуска командного процессора, при этом становятся доступными настройки для его выполнения (посмотрите самостоятельно). Завершение работы режима командной строки выполняется: либо командой EXIT, выполняемой в этом режиме, либо, если включен оконный режим, обычным закрытием окна при нажатии кнопки в правом верхнем углу окна (“х”). Первый способ является более корректным.

При выполнении командного файла возможен вложенный вызов командного интерпретатора, например, если вызывается вложенный командный файл. В этом случае параметры его запуска нужно задать специальной командами (SHELL в файле CONFIG и COMMAND и COMSPEC – см. литературу по MS DOS и лекции по курсу ОС).

Примечание: Здесь и в дальнейшем я ориентируюсь на работу в ОС Windows XP, для других операционных систем возможны некоторые отличия, в частности в названиях системных пунктов меню.

### 1.4. Запуск команд и программ в режиме командной строки

Запуск команд и программ в режиме командной строки выполняется ручным набором имени файла программы или команды после подсказки.

```
>charmap↵
```

Выше приведен пример запуска программы WINDOWS, которая запустится в отдельном окне.

```
>c:\dn\dn.com↵
```

При запуске программ нужно удостовериться, что ОС известен путь (PATH) для запуска программы. Если программа не запускается, то необходимо указать явный путь (как в примере выше) или сделать директорию (каталог) для запуска программы текущей:

```
>c:
>cd c:\dn\
>dn.com↵
```

При запуске программ нужно быть внимательным и набирать имя программы или команды точно. Чаще всего, чтобы избежать ошибок, необходимо набирать и расширение для файла. Если команда или имя программы введены неверно, то Вы получите сообщение операционной системы вида:

```
>c:\dn\dn.exe↵
"DN.exe" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.
```

В этом случае нужно проверить: путь, текущий каталог, имя программы или команды и выполнить ввод заново.



## 1.5. Получение справок о командах в режиме командной строки

Информацию о командах режима командной строки можно получить в литературе или в электронных справочниках (см. ЛР № 1). Кроме того, оперативно можно справку о конкретной команде с помощью директивы HELP, например:

<b>&gt; help EXIT.</b>	
Завершает программу CMD.EXE (интерпретатор команд) или текущий пакетный файл-сценарий.	
EXIT [/B] [exitCode]	
/B	Предписывает завершить текущий пакетный файл-сценарий вместо завершения CMD.EXE. Если выполняется вне пакетного файла-сценария, то будет завершена программа CMD.EXE
exitCode	Указывает цифровое значение. Если указан ключ /B, определяет номер для ERRORLEVEL. В случае завершения работы CMD.EXE, устанавливает код завершения процесс с данным номером.

Полный перечень команд можно получить, выполняя команду HELP без параметров:

<b>&gt;help.</b>	
Для получения сведений об определенной команде наберите HELP <имя команды>	
ASSOC	Вывод либо изменение сопоставлений по расширениям имен файлов.
AT	Выполнение команд и запуск программ по расписанию.
ATTRIB	Отображение и изменение атрибутов файлов.
BREAK	Включение/выключение режима обработки комбинации клавиш CTRL+C.
CACLS	Отображение/редактирование списков управления доступом (ACL) к файлам.
CALL	Вызов одного пакетного файла из другого.
CD	Вывод имени либо смена текущей папки.
CHCP	Вывод либо установка активной кодовой страницы.
CHDIR	Вывод имени либо смена текущей папки.
CHKDSK	Проверка диска и вывод статистики.
CHKNTFS	Отображение или изменение выполнения проверки диска во время загрузки.
CLS	Очистка экрана.
CMD	Запуск еще одного интерпретатора командных строк Windows.
COLOR	Установка цвета текста и фона, используемых по умолчанию.
COMP	Сравнение содержимого двух файлов или двух наборов файлов.
COMPACT	Отображение/изменение сжатия файлов в разделах NTFS.
CONVERT	Преобразование дисковых томов FAT в NTFS. Нельзя выполнить преобразование текущего активного диска.
COPY	Копирование одного или нескольких файлов в другое место.
DATE	Вывод либо установка текущей даты.
DEL	Удаление одного или нескольких файлов.
DIR	Вывод списка файлов и подпапок из указанной папки.
...	

Командный интерпретатор CMD.EXE может работать и в расширенном режиме. В этом режиме доступны дополнительные возможности. Для описания этих возможностей необходимо в режиме командной строки вызвать командный интерпретатор с параметром справки:

<b>&gt;CMD.EXE /?.</b>	
Запуск новой копии интерпретатора команд Windows XP.	
CMD [/A   /U] [/Q] [/D] [/E:ON   /E:OFF] [/F:ON   /F:OFF] [/V:ON   /V:OFF] [[/S] [/C   /K] строка]	
/C	Выполнение указанной команды (строки) с последующим завершением.

/K	Выполнение указанной команды (строки) без последующего завершения.
/S	Изменение поведения после /C или /K (см. ниже)
/Q	Отключение режима вывода команд на экран (ECHO).
/D	Отключение выполнения команд AutoRun из реестра (см. ниже)
/A	Вывод результатов выполнения команд в формате ANSI.
/U	Вывод результатов выполнения команд в формате UNICODE.
/T:цв	Выбор цвета текста/фона (более подробно см. COLOR /?)
/E:ON	Разрешение расширений команд (см. ниже)
/E:OFF	Запрет расширений команд (см. ниже)
...	

Ниже в этой справке дано подробное описание возможностей расширенного режима. Запуск интерпретатора в расширенном режиме выполняется так:

```
>CMD.EXE /E:ON,|
```

Выключение расширенного режима выполняется так:

```
>CMD.EXE /E:OFF,|
```

Справку о работе команд в расширенном режиме можно получить, запустив команду в режиме справки, предварительно переключившись в расширенный режим, или с помощью команды HELP <команда> в обычном режиме.

## Пример простейшей программы

В этом разделе мы предварительно рассмотрим простейшую (возможно, первую для Вас) программу на языке Ассемблер и все необходимые действия в режиме командной строки, которые нужно выполнить для получения исполнимой программы.

Программа выводит на экран один символ (буква “А”) и корректно завершает работу. Программу будем готовить в режиме командной строки с использованием системы программирования TASM (Turbo Assembles – СП можно найти на сайте). Для получения готовой программы нужно выполнить следующие действия:

- Подготовить исходный текст программы, для этого необходимо использовать любой текстовый редактор, но ввод символов необходимо выполнять в формате ANSI или ASCII. Рекомендую использовать программу ASM\_ED.EXE – есть на сайте.
- Выполнить компиляцию программы с помощью компилятора TASM и получить листинг программы.
- Выполнить редактирование связей с помощью редактора связей TLINK.
- Выполнить программу в режиме командной строки.
- Проверить выполнение программы с помощью отладчика TD.

### 2.1. Исходный текст программы на Ассемблере

Введем текст программы, расположенный ниже, и сохраним файл с названием “FIRST.ASM” в каталоге, где расположена система программирования TASM. Данный файл называется также исходным текстом программы, исходным модулем или ее “распечаткой”. Ввод текста можно вводить в любом текстовом редакторе, даже в notepad, но имейте в виду, что ввод для Ассемблера должен осуществляться однобайтовыми символами и должен включать служебные символы. Поэтому ввод текста в MS WORD недопустим. Обратите внимание, что в исходном тексте программы форматирование текста (пробелы перед строками) выполняет сам программист.

```
MYCODE SEGMENT 'CODE'
    ASSUME CS:MYCODE, DS:MYCODE
    LET DB 'A'
    START:
    ; Загрузка сегментного регистра данных DS
    PUSH CS
```

```

        POP    DS
; Вывод одного символа на экран
        MOV    AH, 02
        MOV    DL, LET
        INT    21H
; Ожидание завершения программы
        MOV    AH, 01H
        INT    021H
; Выход из программы
        MOV    AL, 0
        MOV    AH, 4CH
        INT    21H
MYCODE ENDS
END START

```

В нашем примере СП TASM располагается по адресу (в каталоге): C:\BORLANDC\TASM.

## 2.2. Компиляция и листинг программ Ассемблера

Далее нужно перейти в режим командной строки и запустить программу на компиляцию:

```

C:\BORLANDC\TASM>tasm.exe /l /zi /c first.asm
Turbo Assembler Version 3.1 Copyright (c) 1988, 1992 Borland International

Assembling file:   first.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  414k

```

Примечание. Символ “`␣`” не будет появляться на экране. Более подробно параметры запуска компилятора TASM.EXE мы рассмотрим ниже.

Если при компиляции обнаружены ошибки, например, Вы ошиблись в названии команды Ассемблера (я сознательно заменил в программе команду PUSH на команду PASH), то в поток командной строки будет выведено сообщение с номером строки текста, в которой была обнаружена ошибка и ее причина на английском языке (см. ниже – выделено жирным):

```

C:\BORLANDC\TASM>tasm /l /zi first.asm
Turbo Assembler Version 3.1 Copyright (c) 1988, 1992 Borland International

Assembling file:   first.asm
**Error** first.asm(6) Illegal instruction
Error messages:    1
Warning messages:  None
Passes:            1
Remaining memory:  414k

```

Текст сообщения об ошибке - “**Illegal instruction**” означает “**недопустимая команда**”. Если при запуске компилятора задан параметр “/z”, то будет распечатана та строка, в которой была обнаружена ошибка (см. ниже).

```

C:\BORLANDC\TASM>tasm /z /zi /l first.asm
Turbo Assembler Version 3.1 Copyright (c) 1988, 1992 Borland International

Assembling file:   first.asm
PASH CS
**Error** first.asm(6) Illegal instruction

```

```
Error messages: 1
Warning messages: None
Passes: 1
Remaining memory: 414k
```

Параметр запуска компилятора “/l”, заставляет ассемблер сформировать специальный документ – листинг программы. Если не указано другого имени, листинг формируется с названием файла исходного текста программы и расширением - \*.lst. Листинг формируется компилятором Ассемблера и содержит много полезной информации для программиста (описание листинга смотри ниже). Листинг, полученный в нашем случае будет иметь вид:

```
Turbo Assembler Version 3.1 24/02/09 12:46:17 Page 1
first.asm
```

```
1 0000 MYCODE SEGMENT 'CODE'
2 ASSUME CS:MYCODE, DS:MYCODE
3 0000 41 LET DB 'A'
4 0001 START:
5 ; Загрузка сегментного регистра данных DS
6 0001 0E PUSH CS
7 0002 1F POP DS
8 ; Вывод одного символа на экран
9 0003 B4 02 MOV AH, 02
10 0005 2E: 8A 16 0000r MOV DL, LET
11 000A CD 21 INT 21H
12 ; Выход из программы
13 000C B0 00 MOV AL, 0
14 000E B4 4C MOV AH, 4CH
15 0010 CD 21 INT 21H
16 0012 MYCODE ENDS
17 END START
```

```
Turbo Assembler Version 3.1 24/02/09 12:46:17 Page 2
Symbol Table
```

Symbol Name	Type	Value
??DATE	Text	"24/02/09"
??FILENAME	Text	"first"
??TIME	Text	"12:46:17"
??VERSION	Number	030A
@CPU	Text	0101H
@CURSEG	Text	MYCODE
@FILENAME	Text	FIRST
@WORDSIZE	Text	2
LET	Byte	MYCODE:0000 #3 10
START	Near	MYCODE:0001 #4 17
Groups & Segments	Bit Size Align Combine Class	
MYCODE	16 0012 Para	none CODE

В листинге также фиксируются ошибки, причем в том месте, где их определил компилятор. Фрагмент листинга с ошибкой:

```
...
5 ; Загрузка сегментного регистра
данных DS
6 PASH CS
```

```

**Error** first.asm(6) Illegal instruction
      7  0001  1F                      POP  DS
      8                               ; Вывод одного символа на экран
....

```

В конце листинга также дается перечень ошибок, которые были обнаружены

```

...
Error Summary

      PASH CS
**Error** first.asm(6) Illegal instruction

```

Результатом правильной работы компилятора будет формирование в текущем каталоге (у нас - C:\BORLANDC\TASM) объектного модуля с именем файла “FIRST.OBJ”. Проверить его наличие можно с помощью команды DIR, не выходя из режима командной строки. Например:

```

C:\BORLANDC\TASM>dir first.obj
Том в устройстве C имеет метку SYSTEM
Серийный номер тома: D08B-21A4

Содержимое папки C:\BORLANDC\TASM

24.02.2009  13:23                354 FIRST.OBJ

```

Если компиляция завершена с ошибками, то объектный модуль не формируется и нужно вернуться к этапу синтаксической отладки исходного модуля. После успешной компиляции можно приступить к следующему этапу редактированию связей и созданию исполнимого модуля.

### 2.3. Редактирование связей и его результаты

На самом деле операция редактирования связей важна для многомодульных программ, то есть, программ, состоящих из нескольких объектных модулей. Но правила таковы, что даже для программ состоящих из одного модуля эту стадию опустить нельзя. Нужно также учитывать, что на этой стадии формируется исполнимый модуль программы в виде \*.EXE или \*.COM файлов. Для запуска редактирования связей нужно запустить утилиту TLINK следующим образом:

```

C:\BORLANDC\TASM>tlink.exe /v /l /m first.obj
Turbo Link Version 5.1 Copyright (c) 1992 Borland International
Warning: No stack

```

**Примечание.** Символ “␣” не будет появляться на экране. Более подробно параметры запуска редактора связей TLINK.EXE мы рассмотрим ниже. Предупреждение “**Warning: No stack**” выдается по причине отсутствия явного описания сегмента стека в программе на Ассемблере.

При таком завершении редактора связей исполнимый модуль типа \*.EXE будет построен, а его формирование можно проверить командой DIR. Например:

```

C:\BORLANDC\TASM>dir first.exe
Том в устройстве C имеет метку SYSTEM
Серийный номер тома: D08B-21A4

Содержимое папки C:\BORLANDC\TASM

24.02.2009  14:20                990 FIRST.EXE

```

Если при редактировании возникают ошибки, то их содержание и место появляется на дисплее.

```

C:\BORLANDC\TASM>tlink /v /l /m first
Turbo Link Version 5.1 Copyright (c) 1992 Borland International

```

```
Error: Undefined symbol VAR in module FIRST.ASM
Warning: No stack
```

В этом случае команда обращается к внешней переменной (из другого модуля), но при редактировании связей ссылка на нее не была найдена. При редактировании связей мы можем получить специальный файл (\*.map) содержащий информацию о редактировании и возможные ошибки. Такой файл создается при задании параметра “/m”. такой файл называется картой редактирования (map) и играет большую роль при создании многомодульных программ и их отладки. Ниже приведен пример такого файла с включенной ошибкой.

```
Start Stop Length Name Class
00000H 00016H 00017H MYCODE CODE
Address Publics by Name
Address Publics by Value

Line numbers for first.obj(FIRST.ASM) segment MYCODE
7 0000:0001 8 0000:0002 10 0000:0003 11 0000:0005
12 0000:000A 14 0000:000C 15 0000:0011 16 0000:0013
17 0000:0015
Error: Undefined symbol VAR in module FIRST.ASM
Program entry point at 0000:0001
Warning: No stack
```

При правильном завершении редактирования связей ошибки должны отсутствовать:

```
Start Stop Length Name Class
00000H 00016H 00017H MYCODE CODE
Address Publics by Name
Address Publics by Value

Line numbers for first.obj(FIRST.ASM) segment MYCODE
7 0000:0001 8 0000:0002 10 0000:0003 11 0000:0005
12 0000:000A 14 0000:000C 15 0000:0011 16 0000:0013
17 0000:0015
Program entry point at 0000:0001
Warning: No stack
```

## 2.4. Исполнение программы Ассемблера и его результаты

После выполнения всех шагов полученная программа может быть выполнена в режиме командной строки. Для этого нужно ее запустить:

```
C:\BORLANDC\TASM>first.exe.␣
A
```

## 2.5. Запоминание результатов в отдельном файле

Если программа выводит информацию на экран и работает в пакетном режиме (т.е. выполняется и сама завершается, как в нашем примере), то можно перенаправить вывод на дисплей в текстовый файл и затем его просмотреть или распечатать. В нашем примере это может быть сделано так:

```
C:\BORLANDC\TASM>first >> first.txt.␣
```

Сформированный файл “first.txt” содержит одну строку, в начальной позиции которой будет напечатана буква “A”.

## 2.6. Отладка программ, начальные сведения

Для отладки программ в системе программирования TASM используется утилита TD.EXE. Для ее полнофункциональной работы на этапах компиляции и редактирования связей мы установили специальные режимы (см. выше):

- TASM.EXE – режим “/zi” – включение отладочной информации в объектный модуль.
- TLINK.EXE – режим “/v” - включение отладочной информации в исполнимый модуль.

В этом случае мы можем запустить TD и иметь в отдельном окне исходный текст нашей программы. Запуск TD выполняется так:

```
C:\BORLANDC\TASM>TD.EXE first.exe.┘
```

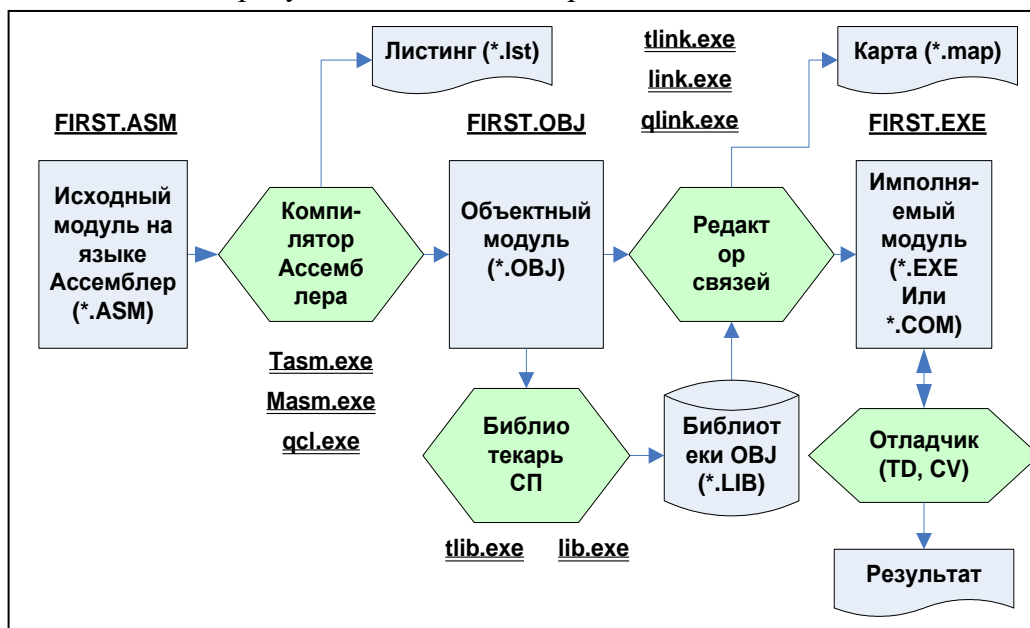
После запуска в окне командной строки появится окно отладчика. Нажимая клавишу F7, мы можем по отдельным командам (по шагам) выполнить нашу программу. Результат работы программы мы можем увидеть, нажав комбинацию Alt+F5. Более подробно об отладке и отладчике мы будем говорить ниже. Для завершения отладчика нужно нажать “Alt + x”.

## Ввод текста, компиляция, редактирование и отладка

В предыдущем разделе мы кратко на простейшем рассмотрели технологические вопросы, связанные с подготовкой и созданием Программ на Ассемблере. В этом разделе мы более детально обратимся к основным понятиям и особенностям такой работы.

Процессы преобразования программ можно упрощенно представить следующим рисунком. На этом рисунке представлены основные компоненты систем программирования, которые участвуют в обработке программ. Кроме того, на рисунке выделены основные виды модулей и файлов, формируемых при такой обработке.

Данная технология формирования программ характерна практически для всех систем программирования, используемых в настоящее время. В тех случаях, когда мы работаем в интегрированной оболочке СП (QC25), мы можем прямо не увидеть промежуточных компонент, участвующих в такой обработке, однако они всегда присутствуют. Эти компоненты отмечены на рисунке двойным подчеркиванием.



### Системы программирования

Для наших лабораторных работ желательно использовать одну из трех систем программирования (они все есть на сайте):

- **Turbo Assembler (TASM)** разных версий. Он включает: **tasm.exe**, **tlink.exe**, **tlib.exe**, **td.exe** и много других вспомогательных программ.
- **Microsoft Assembler (MASM)** разных версий. Он включает: **masm.exe**, **link.exe**, **lib.exe**, **cv.exe** и много других вспомогательных программ.
- **Quick C and Quick Assembler (QC25)** - Он включает: **qcl.exe**, **qlink.exe**, **lib.exe** и много других вспомогательных программ. QC является интегрированной оболочкой, позволяющей выполнять все операции создания программ без переключения в командную строку. В QC встроен также текстовый редактор для подготовки исходных текстов программ.

Другие разнообразные системы программирования на языке Ассемблера (FASM, NASM, RADASM, SASM, EMU8086 и др.) имеют отдельные недостатки для методики преподавания и в данном курсе их использование не рекомендуется. Система программирования **TASM** (Turbo Assembler) – разных версий (3-5) в нашем курсе, в настоящий момент, является предпочтительной.

### Модули представления программ

Существующая технология создания исполнимых программ включает в себя следующие разновидности модулей:

- Исходные модули (ИМ), написанные на языке программирования (например, языке Ассемблер). Исходные модули, обычно, представляют собой длинную строку символов (или совокупность строк) и имеют следующее расширение файлов: **\*.ASM**, **\*.INC**. Исходные модули формируются программистом или подключаются из библиотек (заголовочные файлы – **\*.INC**). Исходные модули кодируются в форматах: ASCII (DOC) или ANSI (Windows)/
- Объектные модули (ОМ) формируются компиляторами после успешной компиляции (без ошибок) исходных модулей. Объектные модули имеют расширение **\*.OBJ** и имеют стандартизованную структуру. Изменять содержание объектных модулей вручную не рекомендуется.
- Исполнимые модули (ИСМ) формируются редактором связей (компоновщиком), который объединяет множество объектных модулей в единую программу. Исполнимые модули могут непосредственно выполняться на компьютере. Они имеют формат: **\*.EXE** или **\*.COM**.
- Библиотеки объектных модулей объединяют множества объектных моделей, связанных по смыслу и функциям. Библиотеки бывают стандартными и пользовательскими. Для построения библиотек используется специальная компонента СП – библиотекарь ( **LIB.EXE** или **TLIB.EXE**).

### Компоненты и стадии обработки программ

Система программирования (СП) это большой комплекс программ, включающий в себя следующие основные программы:

- Компилятор (компиляция) системы программирования, который проверяет правильность написания программ (исходных модулей) и формирует объектные модули. Примеры - **tasm.exe** и **masm.exe**.
- Редактор связей (или компоновщик) необходим для объединения множества объектных модулей в единую программу (исполнимый модуль). Редактор связей настраивает связи между отдельными модулями, которые могут быть двух видов: связи по управлению (вызов функций и процедур) и связи по



данным (использование данных из одного модуля в другом). Примеры: **tlink.exe** и **link.exe**.

- Библиотекарь – программа СП (иногда называют такие программы утилита), которая позволяет создавать библиотеки объектных модулей (ОМ). Библиотеки объектных модулей подключают в программные проекты и, тем самым, обеспечивают подключение нужных объектных модулей в исполнимый модуль.
- Менеджер проектов (MAKE), позволяющий создавать проекты, выполнять сборку из многомодульных программ и проводить их отладку. Чаще всего его работа невидима, эта компонента настраивается на специальный файл проекта, который оформляется на специальном языке.
- Отладчик СП, который обеспечивает возможности эффективной проверки программ и исправления ошибок в программах (TD.EXE, CV.EXE).
- Другие сервисные утилиты предназначены для упрощения процесса программирования и обслуживания создаваемых проектов: текстовые редакторы, справочные системы, примеры использования разных технологий и т.д.

### Фазы подготовки и создания программ на Ассемблере

Для построения исполнимой программы необходимо пройти следующие основные фазы:

- Подготовка алгоритма, кодирование и создание исходных модулей проекта (\*.ASM).
- Компиляция и синтаксическая отладка исходных модулей, формирование объектных модулей проекта (\*.OBJ).
- Редактирование связей для всех объектных модулей проекта, включая и модули из объектных библиотек, и формирование исполнимого модуля проекта (\*.EXE или \*.COM).
- Отладка исполнимого модуля (\*.EXE или \*.COM) с помощью отладчика, если эти модули специально подготовлены для отладки на этапах компиляции и редактирования связей (TD.EXE, CV.EXE).
- Если программа не дает нужный (планируемый/прогнозируемый) результат, то следует “многократное” повторение всех предыдущих этапов (начиная, с любого предшествующего), пока правильный результат не будет получен.

Рассмотрим эти фазы на простейшем примере программы для языка Ассемблер в системе программирования TASM.

## 3.1. Ввод и редактирование текста программы Ассемблера

Ввод и редактирования исходного текста программы, очевидно, самая простая операция процесса подготовки программ, однако она является, с другой стороны, достаточно трудоемкой и, если можно так сказать, “ошибкоемкой” (Исследования, проведенные очень давно, показывают, что на каждые 100 вводимых символов, даже самый опытный оператор,

делает в среднем одну ошибку!). Поэтому выбор и освоение удобного текстового редактора серьезная задача. Текстовый редактор для Ассемблера должен обладать следующими свойствами:

- Обеспечивать ввод в кодировках однобайтовыми символами (не UNICODE).
- Не включать в текст специальные служебные символы (возможно скрытые) для форматирования текста (исключая символ табуляции и конца строки).
- Обеспечивать ввод данных в кодировке ASCII (кодировка ДОС – для отладки программ) и кодировке ANSI (кодировка WINDOWS – для подготовки отчетов по ЛР и КР). Данный пункт относится к русским символам, так как они по разному кодируются в этих кодах.
- Обеспечивать перевод из одной кодировки в другую (ASCII=>ANSI и ANSI=>ASCII).
- Обеспечивать русификацию клавиатуры и шрифтов дисплея.
- Просто разворачиваться и занимать немного места на диске.
- Быть надежным в эксплуатации и легко осваиваться пользователями.

Учитывая сказанное, можно предложить следующие варианты текстовых редакторов для выполнения ЛР:

- Любой текстовый редактор, удовлетворяющий приведенным выше требованиям.
- Текстовый редактор ASM Editor for Windows (есть на сайте).
- На сайте есть специальная программка перекодировки DOS<=>Windows (trans.exe).
- Редактор NOTEPAD совместно с trans.exe.
- Редактор в оболочке QC25 с trans.exe.
- Редакторы в файловых менеджерах (см. ниже) – DN и FAR совместно с trans.exe и русификатором.

Для редактирования текста в полноэкранном режиме и в режиме эмулятора ДОС необходим русификатор шрифта дисплея и клавиатуры. Его необходимо запустить до запуска текстового редактора. Я рекомендую русификатор RKM (есть на сайте). Переключение раскладки клавиатуры по – умолчанию в нем выполняется клавишей – “правый Shift”.

### 3.2. Запуск на компиляцию программы Ассемблера

Компиляция – это процесс преобразования исходного модуля на языке программирования в объектный модуль (см. рисунок, расположенный выше). Компиляция программ может выполняться специальной отдельной программой - компилятором или программой, встроенной в интегрированную оболочку (Environment) системы программирования. При использовании интегрированных оболочек компиляторы вызываются и настраиваются неявно. В качестве отдельных компиляторов для языка Ассемблер мы выделим: TASM.EXE, MASM.EXE и QCL.EXE. Интегрированными оболочками систем программирования являются: QC25, MASM 6.1, VS , BCCP и многие другие. В данном курсе рекомендуется использование: TASM.EXE, MASM.EXE и QC25.

При компиляции, помимо объектного модуля, компилятор формирует форматированный листинг программы, в котором содержится полная информация о программе, процессе компиляции и возможных ошибках. Листинг программы является очень полезным для программиста при отладке программ и для понимания работы компилятора. Кроме листинга при работе компилятора из командной строки на экран дисплея выдается краткая информация о результатах компиляции: успешность процесса, ошибки и т.д. В интегрированных оболочках эта информация выдается в отдельном окне. Процесс компиляции может быть запущен:

- Непосредственно из командной строки.
- Из командного файла, запускаемого в командной строке.

- Неявно, в интегрированной оболочке при нажатии нужной клавиши или выбора пункта меню, соответствующего операции компиляции.

В общем случае запуск компилятора TASM имеет следующий синтаксис:

>TASM[.EXE] [<множество режимов>] <исходный модуль> [, <объектный модуль>] [, <файл листинга>] [, <файл XREF>]

Где:

- TASM[.EXE] - имя исполнимого файла компилятора, он должен быть доступен по путям при запуске или вызове из командного файла.
- <множество режимов> - множество режимов - параметров работы компилятора, разделенных пробелами. Каждый режим начинается служебным символом ‘/’ и имеет символьное значение.
- <исходный модуль> - имя исходного модуля (текстового файла) для компиляции. Если расширение у имени файла отсутствует, то оно воспринимается как ‘.asm’. Это единственный обязательный параметр для запуска компилятора.
- <объектный модуль> - имя объектного модуля, формируемого компилятором. Если это имя отсутствует при запуске компилятора, то оно формируется автоматически на основе имени исходного модуля с заменой расширения на ‘.obj’.
- <файл листинга> - имя файла листинга Ассемблера. Если это имя отсутствует при запуске компилятора, то оно формируется автоматически на основе имени исходного модуля с заменой расширения на ‘.lst’.
- <файл XREF> - файл имен программы и их адресов. Если это имя отсутствует и задан режим компиляции “/с”, то эта информация поступает в файл листинга. Расширение, задаваемое по-умолчанию, - ‘.xrf’.

Пример запуска TASM из командной строки выполняется так (на примере TASM):

```
C:\BORLANDC\TASM>tasm.exe /l /zi /c first.asm
```

Здесь в строке запуска компилятора введены: имя исполнимого модуля компилятора (**tasm.exe**), настройки-режимы компиляции (**/l /zi /c**) и имя модуля подлежащего компиляции (в нашем случае - **first.asm**).

При успешной компиляции на экран дисплея мы получим:

```
Turbo Assembler Version 3.1 Copyright (c) 1988, 1992 Borland International
```

```
Assembling file:   first.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  414k
```

В первой строке выводится название и версия компилятора (**TASM v. 3.1**). Далее представлено имя компилируемого модуля - **first.asm**, число ошибок и предупреждений (у нас нет ошибок - **None**), количество фаз-проходов компилятора (важно при использовании макрокоманд) и размер оперативной памяти, необходимый при размещении объектного модуля в оперативной памяти (**Remaining memory: 414k**). Параметры режимы для нашего случая имеют следующее назначение:

**/l** - формирование компилятором нормального листинга программы

**/zi** – включение в объектный модуль информации, необходимой для работы отладчика

**/c** - включение в листинг таблицы соответствия XREF (перекрестных ссылок) типа команда – адрес ОП

**/z** – выдача на дисплей информации о строке команды, в которой обнаружена ошибка

Более подробно режимы компилятора можно получить, вызвав его в режиме справки (>tasm /?) или обратившись к документации по системе программирования. Эти параметры мы рассмотрим ниже.

### 3.3. Запуск на выполнение и отладку программы Ассемблера

В различных системах программирования для отладки предусмотрены разные программы. В TASM – это Turbo Debugger (TD). В MASM - это Code View (CV). В QC25 отладчик встроен в интегрированную оболочку. Справедливости ради скажу, что QC встроена версия CV.

Процесс отладки заключается в поиске ошибок и причин их возникновения. Для полноценной работы отладчика в исполнимый модуль должна быть включена отладочная информация. Для TASM компиляция и редактирование связей должны быть выполнены с установкой следующих режимов:

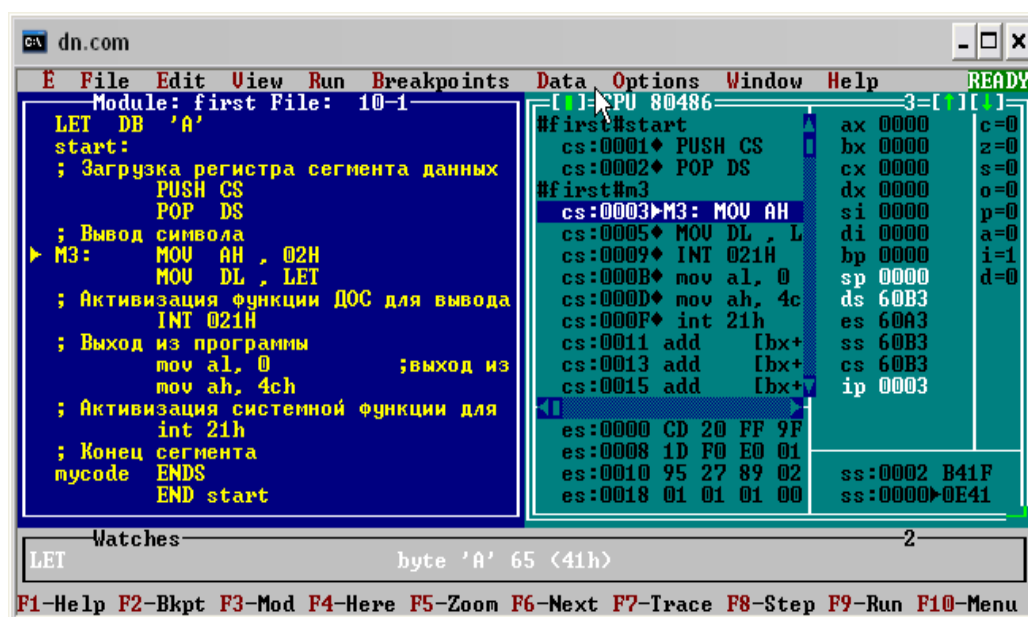
- TASM.EXE – режим “/zi” – включение отладочной информации в объектный модуль.
- TLINK.EXE – режим “/v” - включение отладочной информации в исполнимый модуль.

Если это выполнено, то запуск на отладку может быть выполнен следующим образом:

ЗОМ:

```
C:\BORLANDC\TASM>TD.EXE first.exe
```

Отладчики работают в режиме диалога, поэтому на экране появиться специальное окно отладчика, показанное ниже.



В левой части окна отладчика расположен исходный текст отлаживаемой программы (окно - **Module**), а в правой части ее машинное представление (окно - **CPU**). Кроме того в этом окне показаны: регистры микропроцессора (AX-IP), основные флаги (с - d), область сегмента данных (ES) и область сегмента стека (SS). В нижней части экрана показано

окно просмотра данных (окно - **Watches**), в котором могут быть представлены любые переменные программы.

В целом отладчик позволяет выполнять следующие операции:

- Выполнять программу по шагам (трассировка по отдельным командам). Существуют несколько возможностей трассировки: F8, F7, F4, F9.
- Просматривать регистры, переменные, области памяти по ходу выполнения программы.
- Останавливать программу в заданном месте (точки останова – **BreakPoint**) или по заданному условию.
- Выполнять изменение данных в процессе такой трассировки.
- Увидеть результаты работы программы (Alt+F5).

Аналогичные действия можно выполнить в среде QC25, однако запускать для этого отдельного отладчика не нужно, он интегрирован в оболочку. Для запуска отладки достаточно установить параметры компиляции и редактирования с возможностью отладки, создать программу и нажать клавишу F8. Мы получим следующее окно (см. рисунок ниже), в котором доступны практически те же возможности, что и в TD..

The screenshot shows a window titled "dn.com" with a menu bar (File, Edit, View, Search, Make, Run, Debug, Utility, Options, Help). Below the menu is a status bar showing "DEBUG: <unknown>". The main window is divided into two panes. The top pane, titled "REGISTERS", displays the following values:

AX:0000	BX:0000	CX:0000	DX:0000	FL:"NU UP EI PL NZ NA PO NC"
SI:0000	DI:0000	SP:2EDE	BP:0000	IP:0002
DS:3EA8	ES:3EA8	SS:3EB8	CS:3EB8	

The bottom pane shows assembly code for "MYCODE SEGMENT 'CODE'". The code includes comments in Russian and assembly instructions:

```

MYCODE SEGMENT 'CODE'
ASSUME CS:MYCODE, DS:MYCODE
LET DB 'A'
start:
; Загрузка регистра сегмента данных
PUSH CS
POP DS
; Вывод символа
M3: MOV AH, 02H
MOV DL, LET
; Активизация функции DOS для вывода
INT 021H
; Выход из программы
  
```

At the bottom of the window, there is a status bar with keyboard shortcuts: <F1=Help> <F6=Window> <F5=Run> <F8=Trace> <F10=Step> and a counter showing "00007:001".

В данном окне мы видим основную программу, которая может быть выполнена по шагам. В пункте меню “**Run**” перечислены возможности такого выполнения (клавиша - “**F8**”). Текущая строка программы, которая должна быть выполнена на очередном шаге, выделена цветом (в нашем случае - **зеленым**). В специальном окне (“**Registers**”) показаны все регистры программы и флаги (“**FL**”). Данное окно может быть вызвано так: “**View**” => “**Windows**” => “**Registers**”. Окно “**Debug**” предназначено для индикации текущих значений переменных и выражений программ. Это окно может быть вызвано так: “**Debug**” => “**Watch values ...**” => “<имя переменной> - **Add**”. В нижней строке окна дана подсказка для возможных операций трассировки программы. Результаты работы программы (вывод на консоль) можно посмотреть в специальном окне “**View**” => “**Windows**” => “**Output**” или нажать в любой момент клавишу “**F4**”.

### 3.4. Запуск на выполнение программы Ассемблера

Запуск на выполнение построенной исполнимой программы выполняется таким же образом, как и запускается любая программа. Возможны следующие варианты:

- Запуск из командной строки;
- Запуск под управлением отладчика;
- Запуск из файлового менеджера;
- Запуск в среде WINDOWS.

При запуске из командной строки мы должны точно знать имя формируемого исполнимого модуля и обеспечить путь к программе в среде операционной системы (Команды CD или переменная PATH). При запуске из командной строки нужно ввести

```
C:\TASM>first.exe.␣
A
```

Результат работы программы мы увидим следующей строке (“**A**” выделено шрифтом).

При запуске под управлением отладчика необходимо воспользоваться специальными клавишами. В TD – это клавиша “**F9**”, а QC25 - “**F5**”. Результат работы программы мы можем посмотреть: в TD – “**Window**”=> “**User screen**” (или комбинация Alt+F5), а в QC25 “**View**” => “**Windows**” => “**Output**” (или нажать клавишу “**F4**”).

При запуске из файлового **менеджера** запуск производится простым нажатием клавиши “**Enter**” после выделения конкретной программы в панели просмотра файлов. Для программы, при необходимости, будет запущено окно командной строки, а результаты можно наблюдать в этом окне.

При запуске в среде WINDOWS (любым известным Вам способом) будет автоматически запущена командная строка, а в ней программа. Если в программе не предусмотрено ожидание ввода клавиши, то окно после этого также автоматически закроется. Поэтому желательно в программе предусмотреть еще несколько команд вида перед завершением программы:

```
; Ожидание завершения программы
MOV AH, 01H
INT 021H
```

В этом случае окно командной строки не будет закрыто автоматически, но после нажатия на любую клавишу программа и окно командной строки закроется.

### 3.5. Применение командных файлов для работы с Ассемблером

Если Вы уже попробовали отладить простейшую программу в режиме командной строки, а даже под управлением файлового менеджера, то вы убедились, что необходимо многократно вводить команды (“пальчиками!!!”), что достаточно трудоемко и приводит часто к ошибкам ввода. Наиболее удобно работать в интегрированной среде, освоив возможности управления клавишами или мышкой. Не все программисты желают использовать интегрированные оболочки для работы, хотя трудоемкость их, несомненно, не устраивает. Для преодоления этого факта можно использовать командные файлы (см. лабораторную работу № 2). Можно создать командный файл, который настраивается на параметры (например, на имя исходного модуля), и использовать его для отладки программ. Такой файл может содержать различные настройки, меню для управления и включать все фазы работы с программой, перечисленные выше. Пример самого простого командного файла, который я назвал “COMPTASM.BAT” приведен ниже:

```
REM фаза редактирования исходного модуля
notepad.exe %1.asm
REM фаза компиляции
tasm /l /zi /c %1.asm
REM фаза редактирования связей
tlink /v /m /l %1.obj
REM фаза отладки
PAUSE
td %1.exe
```

Для запуска такого командного файла достаточно указать один параметр, который определяет имя модуля для компиляции и редактирования. Кроме того, должны быть определены пути (PATH) ко всем выполняемым в командном файле программам. В этом командном файле я вызываю стандартный редактор текста “notepad.exe” и вставил команду “PAUSE” для контроля предыдущих шагов. Можно здесь “пофантазировать” и создать файл: без отладчика, зациклить его выполнение, сделать контроль вводимого параметра и т.д. Файл с зацикливанием и выполнением без отладчика показан ниже:

```
:START
RKM
IF (%1) ==() GOTO FINISH
REM фаза редактирования исходного модуля
notepad.exe %1.asm
REM фаза компиляции
tasm /l /zi /c %1.asm
REM фаза редактирования связей
tlink /v /m /l %1.obj
PAUSE
REM фаза выполнения
%1.exe
PAUSE
GOTO START
:FINISH
ECHO НЕТ ПАРАМЕТРА
```

В данном случае необходимо обратить внимание на кодировку русских букв и в программе и в командном файле. Кодировка (ANSI или ASCII) должна соответствовать режиму командной строки, в котором вы работаете, и при необходимости нужно в командном файле запустить русификатор (RKM). К сожалению, приведенный командный файл приходится завершать с помощью кнопки завершения окна в WINDOWS (“X”).

## Формальное описание синтаксиса в БНФ

Во многих учебниках, пособиях, электронных справочниках и документации Вы можете встретить формальное описание синтаксиса языков программирования и правил запуска программ. Такое описание необходимо для однозначного понимания правил записи операторов и команд, исключения ошибок и, в конечном счете, снижения сроков освоения языков и написания программ. Существует много различных языков формального описания других, извините за тавтологию, формальных языков, в частности языков программирования. Они делятся на две группы: текстовые и графические. В графических языках используются специальные диаграммы и правила их построения (пример графического языка для описания Ассемблера Вы найдете в учебнике Юрова).

Более четкое описание можно получить на основе текстовых формальных языков. Одним из примеров распространенных языков формального описания является БНФ. Данная аббревиатура имеет две расшифровки: Бекуса Нормальная Форма (более ранняя) и Бекуса Наура Форма (более поздняя). Разные названия связаны с тем, что второй автор (Наура) внес уточнения в первоначальные правила. В этом пособии мы рассмотрим основные правила применения БНФ для оформления документации ЛР и изучения Ассемблера.

### 4.1. Назначение и состав языка БНФ

Язык БНФ является распространенным языком описания формализованных конструкций других языков. Такие языки называют также метаязыками. Можно дать математическое описание этого языка на основе теории множеств и теории формальных языков, но здесь мы остановимся на более простом смысловом и текстовом его описании. Язык БНФ включает в себя следующие основные элементы и понятия:

- Понятие **терминального** символа и множества терминальных символов. Под терминальным символом понимается такой элемент описания, который буквально входит в синтаксические конструкции (предложения) языка. Эти символы не могут быть раскрыты далее в виде правил. Примером терминальных символов могут служить: знаки операций (“+”, “-“ и т.д.), ключевые слова языка ( for, loop, case и др.) и служебные символы (“=”, “;” и т.д.). Отметим, что любое правило должно быть раскрыто (порождать), в конечном счете, цепочку терминальных символов.
- Понятие **нетерминального** символа и множества нетерминальных символов. Под нетерминальными символами мы понимаем такой элемент языка, который представляет собой некоторое понятие языка и подлежит раскрытию с помощью специальных правил. Нетерминальные символы называют также переменными языка. При конкретном порождении нетерминальные символы должны, с помощью правил быть раскрыты в виде терминальных символов. Примером нетерминального символа является понятие идентификатора (или имени переменной). По существующим правилам записи идентификаторов мы можем построить очень большое число различных вариантов имен переменных. Другим примером нетерминального символа является понятие целого числа или числа вообще, которое можно использовать в виде констант при написании операторов.
- Понятие **правил** грамматики языка и множества этих взаимосвязанных правил, составляющих, по сути, саму грамматику формального языка. Правило записывается с помощью метасимволов и предназначено для раскрытия нетерминальных символов с помощью комбинации других нетерминальных символов и терминальных символов. Предполагается, что существует некоторое первоначальное правило, с которого должно проводиться описание грамматики и, что более важно, грамматический разбор конкретной конструкции.



- Множество **метасимволов** языка БНФ, которые используются для записи правил грамматики описываемого формального языка. Это символы: “:=”, “<”, “>”, “[”, “]”, “{”, “}”, “...”. Более подробно о назначении метасимволов мы поясним ниже. Отметим также, что некоторые символы (а именно, “[”, “]”, “{”, “}”, “...”) входят в расширенный вариант БНФ и не являются обязательными.

## 4.2. Правила, нетерминальные переменные и метасимволы

Нетерминальный символ на языке БНФ записывается следующим образом: в угловые кавычки (<...>) помещается произвольный текст, который используется для обозначения понятия языка. Нетерминальный символ и его обозначение должны быть уникальными в пределах описываемой грамматики, то есть не повторяться. Примерами нетерминальных символов могут служить следующие понятия:

<Переменная> - понятие, описывающее переменную в программе,

<Оператор> - понятие, описывающее оператор в языке программирования.

Правило грамматики записывается в следующем виде (правила похожи на операторы присваивания, но нельзя их путать):

<Нетерминальный символ>:= <Выражение на основе терминальных символов, метасимволов и нетерминальных символов>

Для пояснения самого языка БНФ нам приходится использовать также язык БНФ. В левой части правила должен быть расположен только один нетерминальный символ. Выражение является комбинацией терминальных и нетерминальных символов, а также метасимвола “|”, который позволяет упростить написание правил и, в конечном счете, не является обязательным. Так, например, два правила вида (где, НС – нетерминальный символ):

<НС1>:= <НС2> и

<НС1>:= <НС3> можно объединить в правило вида:

<НС1>:= <НС2> | <НС3> , где НС1 ÷ НС3 нетерминальные символы.

Правила на языке БДН могут быть рекурсивными, это означает, что один и тот же нетерминальный символ, задаваемый правилом, может присутствовать в правом выражении этого же правила. Например:

<НС1>:= <НС2> | <НС2> <НС1>

В выражении правил могут располагаться терминальные символы (ТС). Они могут располагаться в любом порядке. Например:

<НС1>:= ТС1 <НС2> ТС2 ТС3 <НС3> ТС4

При описании конструкций на языке Ассемблера возникает проблема с использованием угловых скобок, так как они являются также терминальными символами языка. В этом случае, либо делается словесное пояснение, либо эти символы выделяются жирным шрифтом, либо заключаются в круглые кавычки “(“ - ”)”. Например:

<Описание записи>:= <Имя записи> <Шаблон записи> (<)<Инициализация записи>(>)

Заключенные в круглые скобки символы рассматриваются как терминальные символы языка. Для упрощения записи правил могут дополнительно использоваться квадратные скобки - “[”, “]”. Применение их означает, что заключенное в них подвыражение может отсутствовать. Например, правил вида:

<НС1>:= <НС2> [<НС1>]

Заменяет более сложное правило вида:

<НС1>:= <НС2> | <НС2> <НС1>

При перечислении повторов можно также использовать и фигурные скобки метасимволы - “{”, “}”, если это не усложняет запись правила. Так, например, правило вида:

<НС1>:= {<НС2> , } ...

Означает сокращенную форму записи правила вида:

$\langle \text{HC1} \rangle := \langle \text{HC2} \rangle \mid \langle \text{HC2} \rangle , \langle \text{HC2} \rangle \mid \langle \text{HC2} \rangle , \langle \text{HC2} \rangle$  и так далее.

Если обратиться к строгой записи на языке БНФ, то использование круглых и фигурных скобок можно исключить. В документациях на многие программные продукты Вы найдете описания на этом языке, включая и расширенное его толкование.

При формальном описании на языке БНФ необходимо помимо формальных правил давать текстовые пояснения основных понятий – нетерминальных символов. Это выполняется в текстовом режиме.

### 4.3. Примеры описания на БНФ

Рассмотрим несколько примеров описаний на языке БНФ.

Пример 1. Описание в БНФ синтаксиса целого числа со знаком:

```
<целое число со знаком>:= <целое число> | + <целое число> | - <целое
число>
<целое число>:= <цифра> | <цифра> <целое число>
<цифра>:= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Таким синтаксическим правилам соответствуют числа: +15, -35, 01, 12345678788 и т.д. Отметим, что для написания программ, недостаточно одних синтаксических правил, нужно еще дополнительное семантическое (понятийное или смысловое) описание языка (семантические правила). Даже в этом простом примере нужно указать максимальное число знаков целой константы, которое допустимо в конкретной системе программирования. Если данная константа соответствует типу `int`, то для записи ее можно указать только 5 значащих цифр ( $2^{16} = 65536$ ).

В нашем примере нетерминальными символами являются: `<целое число со знаком>`, `<целое число>` и `<цифра>`. Терминальными символами являются: “+”, “-”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”.

Отметим также, что один и тот же синтаксис может быть описан разными способами. Так первое правило может быть описано так:

```
<целое число со знаком>:= <знак> <целое число>
<знак>:= <пусто> | + | -
```

Где `<пусто>` часто используемый для описания правил нетерминальный (фактически терминальный) символ, означающий пустую строку (отметим – не пробел, который является значащим терминальным символом. Если при описании правил требуется указать терминальный символ –пробел, то его обозначают специальным символом, например, – “\_”).

Пример 2. Инструкция командной строки для запуска программ.

```
<запуск программы>:= <имя программы> _ <список параметров программы>
<имя программы>:= prog.exe | prog
<список параметров программы>:= <Параметр 1>[_<Параметр 2>] [_<Параметр
3>]
<Параметр 1>:= <Имя файла>
<Параметр 2>:= /L
<Параметр 3>:= /P:<Пароль>
```

Отметим, что и здесь необходимо пояснить смысловое (семантика правил) значение правил синтаксиса. Так `<Имя файла>` - например, стандартный текстовый файл операционной системы (в некоторых случаях не нужно далее раскрывать значение нетерминального символа, сославшись на общепринятые правила в операционной системе, но можно и раскрыть). Нужно раскрыть также, что означает использование параметра 2 (/L), например установку специального режима работы программы. Для параметра 3 нужно пояснить

число символов, которые нужно ввести для пароля. Символ “\_” в нашем описании обозначает пробел, а параметры 2 и 3 могут быть опущены при запуске. Если допускается изменять последовательность параметров (пусть 2 и 3), то нужно в описание грамматики запуска добавить дополнительное правило вида:

<список параметров программы>:= <Параметр 1>[\_< Параметр 3 >] [\_< Параметр 2 >]

Для описания инструкций командной строки во 2-й, 6-й лабораторных работах и в курсовой работе необходимо использовать подобное описание.

Пример 3. Рассмотрим также для примера описание команды IF в формате БНФ. Во-первых, не будем учитывать возможности расширенного режима работы CMD, и, кроме того, сократим при этом немного синтаксис и семантику описания.

```

<Оператор IF>:= IF_ [NOT] <Варианты конструкции оператора IF>
<Варианты конструкции оператора IF>:= <конструкции ERRORLEVEL> | <кон-
струкции со строками> | <конструкции EXIST>
<конструкции ERRORLEVEL>:= ERRORLEVEL_<число>_<команда>
<конструкции со строками>:= <строка>==< строка >_<команда>
<конструкции EXIST>:= EXIST_<имя файла>_<команда>
<строка>:=<любая последовательность символов до пробела _> | <переменная>
| %<переменная>% | <параметр командной строки>
<параметр командной строки>:=%<десятичная цифра>
<десятичная цифра>:=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
9

```

Где:

<команда> - любая команда языка командных файлов, исключая IF. Данная команда выполняется в случае, если условие, заданное в операторе IF, является истинным.

<число> - целое число без знака, которое сравнивается со значением переменной ERRORLEVEL, определяющий код возврата последней выполненной в командном файле программы или встроенной команды. Между этими значениями подразумевается знак отношения (>=).

<имя файла> - стандартное имя файла операционной системы, существование которого проверяется в условии EXIST. Условие считается истинным, если файл найден в пределах текущей директории или по заданному пути (PATH).

<переменная> - любая текстовая переменная командного файла, включая и системные переменные окружения установленные командой SET. Если не указаны символы процентов (%), то имя переменной рассматривается как строка, а противном случае рассматривается значение этой переменной.

<любая последовательность символов до пробела > - любая последовательность символов, которая ограничивается пробелом. Необходимо следить, чтобы в группу символов случайно не попадали служебные слова.

Примеры использования оператора командных файлов IF:

1. IF ERRORLEVEL 3 GOTO MET1 – переход на метку MET1, если ERRORLEVEL >= 3
2. IF (%1) == () SET VAR=NODOSTUP – переменной VAR присваивается значение NODOSTUP если первый параметр не задан.
3. IF NOT EXIST FIRST.EXE GOTO FINISH – переход на метку FINISH, если файла FIRST.EXE не существует.

## Командные файлы и их применение

Язык командных файлов – это простейший язык программирования с усеченными возможностями: ограниченное число операторов, ограниченное число типов переменных и ограниченные возможности по отладки программ. Язык командных файлов предназначен

для написания программ управления запуском других программ (заданий). Поэтому его ранее называли, а это Вы можете встретить в литературе, Языком Управления Заданиями (ЯУЗА). Командные файлы (Batch Files) имеют специальное файловое расширение (\*.BAT), которое не может быть изменено. Благодаря этому расширению операционная система автоматически запускает режим командной строки (см. выше) и выполняет командный файл под управлением командного процессора (CMD.EXE или COMMAND.COM).

Выше было отмечено, что командный процессор CMD.EXE может работать в двух режимах: стандартном и расширенном. В расширенном режиме некоторые команды имеют дополнительные возможности. При рассмотрении особенностей командных файлов и директив, мы будем ориентироваться на стандартный режим. При необходимости Вы можете включить расширенный режим и получить справки о работе команд в этом режиме.

Запуск интерпретатора CMD.EXE в расширенном режиме выполняется так:

```
>CMD .EXE /E:ON↵
```

Выключение расширенного режима выполняется так:

```
>CMD .EXE /E:OFF↵
```

### 5.1. 5.1 Язык командных файлов

Программа на языке командных файлов может быть представлена так:

```
[:<метка>]<инструкция командного файла>
[:<метка>]<инструкция командного файла>
[:<метка>]<инструкция командного файла>
...
[:<метка>]<инструкция командного файла>
```

Метка для языка командных файлов, если она используется, должна располагаться в начале строки, и иметь символьное обозначение, которое, кстати, может начинаться и с цифрового символа (Например, - “:MET1”, “:10”, “:LAB”). Для обозначения меток допускается использовать только латинские символы. В качестве инструкций командных файлов можно использовать:

- Директивы командных файлов;
- Внутренние команды операционной системы;
- Внешние команды (утилиты) операционной системы;
- Любые программы доступные для запуска;
- Вызов других командных файлов;
- Пустые строки с меткой.

На одной строке в программе командных файлов (за исключением оператора IF) может располагаться только одна инструкция. В качестве основных директив командных файлов выделим следующие директивы, пояснив кратко их назначение:

- Директива **IF** – для организации ветвления в командных файлах, организации циклов и проверки условий.
- Директива **SET** – для присваивания переменным окружения значений.
- Директива **SHIFT** – для изменения порядка следования (сдвига) параметров командной строки, заданных при запуске программы.
- Директива **FOR** – для организации выполнения однородных команд применительно к множеству определенных переменных.
- Директива **ECHO** – для организации вывода на консоль и переключения режимов вывода информации.

- Директива **GOTO** – для выполнения безусловных переходов на конкретные метки программы данного командного файла.
- Директива **PAUSE** – для организации паузы при работе командного файла с заданием информационного сообщения.
- Директива **REM** – для комментирования командных файлов.
- Директива **CALL** – Для вызова других командных файлов с возвратом (вложенных командных файлов) в данный командный файл (аналогично вызову функций и процедур).
- Директива **EXIT** – для завершения выполнения командного файла. Если данная команда выполняется во вложенном командном файле без параметра (/b), то выполняется завершение работы всех файлов (фактически завершается CMD.EXE). Если параметр (/b) установлен, то выполняется возврат в основной файл.
- Директива **CLS** – для очистки экрана дисплея (консоли) в момент выполнения данной директивы.
- Модификатор **@** - для отключения вывода директивы на экран дисплея.

Более подробную информацию о директивах командных файлов вы найдете в справочниках, литературе по ДОС и документации на операционную систему. Можно получить справку о всех командах и директивах вызывая утилиту HELP:

```
>HELP↵
```

Или для конкретной команды вызов справки выполняется так:

```
> HELP <команда>↵
```

Кроме того, справку по конкретной команде можно получить так:

```
><команда> /?↵
```

И наконец, в среде WINDOWS всегда можно нажать клавишу “F1” при активном рабочем столе (Desktop) и ввести название команды в поле для поиска.

Набор доступных внешних команд Вы можете уточнить в справках и документации, а также посмотреть в каталогах ОС (для XP – SYSTEM32), которые доступны в каталоге WINDOWS. Для этой цели удобнее использовать файловые менеджеры (см. ниже).

## 5.2. Разработка, ввод, отладка и исполнение командных файлов

Создание командных файлов – это процесс программирования, который включает все основные этапы: разработка алгоритмов, построение блок-схем программы, написание программы на языке командных файлов, ввод ее в виде электронного документа, отладка программ командных файлов и их исполнение (запуск).

В зависимости от сложности поставленной задачи при создании командного файла, создание алгоритма программы может быть и простым и сложным. Этап разработки алгоритма должен завершаться разработкой блок-схемы программы (см. раздел ниже). Командный файл вводится в простом текстовом редакторе (см. раздел по вводу и редактированию программ на языке Ассемблер). Имя командного файла должно иметь специальное расширение (\*.BAT). Желательно не использовать длинных имен (> 8-ми символов), так как могут возникнуть проблемы при работе с программой в режиме командной строки.

К сожалению, для отладки командных файлов нет специальных и удобных отладчиков, поэтому этот процесс может быть трудоемким. Основные рекомендации для отладки программ командных файлов:

- Не отключайте преждевременно режим вывода команд и директив на экран дисплея (нужен режим ECHO ON).
- Не отключайте режим вывода отдельных команд с помощью модификатора – “@”.

- При необходимости используйте команду ECHO для вывода промежуточных значений переменных программы.
- При необходимости используйте команду PAUSE при отладке частей программы.
- Изучите хорошо и грамотно используйте директивы командных файлов.
- Корректно используйте директиву комментирования REM, для пояснения частей программы и для выделения критических мест в тексте программы.
- Используйте вложенные командные файлы для выделения фрагментов, которые имеют отдельное функциональное значение и могут быть отлажены отдельно.
- Сохраняйте старые копии программ командных файлов при существенных изменениях, сделанных в процессе отладки.

Исполнение (запуск) командных файлов возможно в режиме командной строки и при запуске из других командных файлов. Если подразумевается возврат в основной командный файл при таком запуске, то необходимо использовать директиву CALL для запуска нового командного файла и директиву EXIT [/b] для корректного возврата или завершения работы. Пример запуска командного файла:

```
>МАУВАТЧ.БАТ.↓
```

или

```
>МАУВАТЧ.↓
```

Во втором случае расширение опущено, но при этом нужно быть точно знать, что в данном каталоге нет файлов \*.EXE или \*.COM с таким же именем.

При запуске из другого командного файла:

```
CALL МАУВАТЧ.БАТ PAR1 PAR2
```

И возврат из файла МАУВАТЧ.БАТ:

```
EXIT /b
```

Параметр определяет в нашем случае возврат в основной файл. Завершения работы CMD.EXE не выполняется. Режим работы команды EXIT зависит от выбора командного процессора для использования командной строки (CMD.EXE или COMMAND.COM).

### 5.3. Переменные командных файлов

Язык командных файлов является очень ограниченным с точки зрения использования переменных. Допускаются только символьные переменные, причем их число ограничено объемом буфера, выделяемого для переменных окружения. Можно использовать только одну стандартную переменную целого типа – ERRORLEVEL (об этой переменной речь пойдет ниже), которая содержит код возврата исполняемых программ и директив в командном файле и формируется автоматически.

Размер области под системные переменные может быть установлен в специальных системных переменных, которые устанавливаются при загрузке ОС. Это переменные:

```
SHELL= [<имя и путь к command.com>] /E:<число> <параметры ...>
```

Для файла config.sys/nt. Или:

```
command [<имя и путь к command.com>] /E:<число> <параметры ...>
```

Для файла config.sys/nt. Или:

```
SET COMSPEC = [<имя и путь к command.com>] /E:<число> <параметры ...>
```

Для переменной окружения COMSPEC, которая может быть переопределена в режиме командной строки. Значение параметра <число> определяет размер буфера для переменных окружения программы, оно может находиться в пределах от 160 до 32768. По умолчанию оно равно 256 байт.

Переменные командных файлов могут быть использованы в операторах командных файлов, могут быть переопределены, выведены на экран дисплея. Основным оператором изменения переменных это оператор SET. Например:

```
SET VAR=MET1
```

Переменные имеют строковый тип. Каждая переменная имеет имя (название, идентификатор) и значение. Для использования значения переменных их необходимо окружить знаком процент (%VAR%). В следующем примере показано как использовать значение и имя:

```
ECHO OFF
...
SET VAR=MET1
ECHO VAR
ECHO %VAR%
...
```

После выполнения данного фрагмента на экране дисплея мы получим на экране дисплея:

```
>VAR
>MET1
```

Переменные в командных файлах могут использоваться в различных директивах командных файлов, в том числе и в директиве IF:

```
IF (%VAR%) == (FIN) GOTO MFINISH
```

Здесь проверяется значение переменной и выполняется переход на метку MFINISH. Или в операторах GOTO:

```
GOTO %VAR%
```

Переход выполняется по значению переменной VAR. Для перехода можно использовать и переменную ERRORLEVEL:

```
GOTO M%ERRORLEVEL%
```

В этом случае будет выполнен переход на метки: M1 (При ERRORLEVEL=1), M2 (При ERRORLEVEL=2) и т.д. Такие метки должны быть объявлены в данном командном файле. Можно использовать и составное значение двух и более переменных:

```
SET VAR=MET
GOTO %VAR%ERRORLEVEL%
```

В этом случае будет выполнен переход на метки: MET1 (При ERRORLEVEL=1), MET2 (При ERRORLEVEL=2) и т.д. Ниже мы рассмотрим и другие примеры использования переменных командных файлов.

## 5.4. Параметры командных файлов

Выше было отмечено, что командные файлы могут запускаться с параметрами. Рассмотрим требования к такому запуску. Параметры передаются в программу командного файла и могут в нем использоваться наравне с переменными (параметры имеют обозначение - %0, %1, ... %9). Синтаксис запуска с параметрами на языке БНФ имеет вид:

```
<запуск с параметрами>:= <имя программы>_ <параметр> |
<запуск с параметрами> <разделитель> <параметр>
<разделитель> := _ | , | ;
```

Или так:

```
<запуск с параметрами>:= <имя программы>_<параметр>
[ {<разделитель><параметр> ...} ]
```

Пример запуска командного файла с параметрами:

```
>MYBATCH.BAT PAR1 , PAR2 , , PAR4
>MYBATCH.BAT PAR1 PAR2 PAR3
```

Из примеров видно, что разделителями параметров могут быть: пробелы, запятые и точки с запятой. Могут использоваться запятые и точки с запятой, в этом случае они аналогичны пробелам. Возможное число параметров, задаваемых при запуске командного файла, определяется возможной длиной запуска программ в режиме командной строки. Если вы работаете в режиме эмуляции ДОС, под управлением файлового менеджера, то строка за-



пуска командного файла вместе с параметрами не должна превышать 128 символом. Нетрудно подсчитать, что предельное число параметров при этом может быть 63 (проверьте сами). Кстати, такое ограничение действительно и для программ на Ассемблере, запускаемых в режиме эмуляции. Если мы работаем под управлением CMD.EXE, то число параметров и длина отдельного параметра командного файла практически неограниченны.

Задание параметров можно проиллюстрировать на следующем примере. Пусть у нас создан командный файл следующего вида (Test\_par.bat):

```
ECHO OFF
ECHO Programm Name - %0
ECHO First Parameter - %1
ECHO Second Parameter - %2
ECHO Third Parameter - %3
```

При запуске:

```
>test_par.bat par1 par2 par3
```

Мы получим:

```
Programm Name - test_par.bat
First Parameter - par1
Second Parameter - par2
Third Parameter - par3
```

При запуске:

```
>test_par.bat par1 , , par3
```

Или:

```
>test_par.bat par1 ; ; par3
```

Мы получим:

```
Programm Name - test_par.bat
First Parameter - par1
Second Parameter - par3
Third Parameter -
```

Так как запятые и точки с запятой эквивалентны пробелам, то фактически рассматривается только два параметра.

Если перечень необходимых параметров превышает девять (%1 - %9), то для доступа к старшим параметрам можно воспользоваться директивой SHIFT. Эта директива сдвигает параметры влево: 2-й (%2) параметр становится 1-м (%1) и так далее. Покажем это на примере:

```
@ECHO OFF
ECHO Programm Name - %0
ECHO First Parameter - %1
ECHO Second Parameter - %2
SET Name =%0
SHIFT
ECHO.
ECHO Programm Name - %0
ECHO First Parameter - %1
ECHO Second Parameter - %2
ECHO %Name%
```

После запуска командного файла:

```
>test_par.bat par1 par2
```

Получим:

```
Program Name - test_par.bat
First Parameter - par1
Second Parameter - par2

Program Name - par1
First Parameter - par2
Second Parameter -
test_par.bat
```



Параметры, включая и параметр %0, теряются безвозвратно, если их предварительно не сохранить в другой переменной окружения (как показано в примере). Команда ECHO с одной точкой (“ECHO.”) используется в командных файлах для пропуска строки.

## 5.5. Создание простого командного файла

Теперь мы знаем достаточно, для того чтобы создать свой командный файл. Назовем его FIRST.BAT. Пусть он имеет следующее содержание:

```
ECHO OFF
ECHO BATCH FILE
ECHO %0
SET VAR=MET1
ECHO VAR
ECHO %VAR%
ECHO.
REM This is Comment =====
IF (%1) == () GOTO %VAR%
PAUSE TEST
ECHO FINISH
EXIT
REM =====
:MET1
ECHO FINISH - MET1
PAUSE NO PARAMETER
EXIT /b
```

В этом файле показано как:

- Отключить вывод выполняемых команд на экран (**ECHO OFF**);
- Выводить на экран текст и параметры (**ECHO BATCH FILE** , **ECHO %0**);
- Присваивать переменным значения (**SET VAR=MET1**);
- Выводить на экран значение переменной (**ECHO %VAR%**);
- Выводить пустую строку (**ECHO.**);
- Использовать комментарии в командном файле (**REM**);
- Проверять параметр и переходить по метке (**IF (%1) == () GOTO %VAR%**). Для проверки пустого параметра нужно использовать вспомогательные символы иначе конструкция оператора будет ошибочной. Можно и по другому – (**IF .%1 == . GOTO %VAR%** );
- Задавать приостановку выполнения командного файла (**PAUSE TEST** и **PAUSE NO PARAMETER**) с выдачей сообщения;
- Выходить из программы с завершением командного файла и командного процессора (**EXIT**);
- Выходить из программы с завершением только командного файла (**EXIT /b**);

Данный командный файл можно запустить с параметром и без параметра. При запуске с параметром:

```
>first.bat par1
```

На экране мы получим:

```
>ECHO OFF
BATCH FILE
FIRST.bat
VAR
MET1
```

```
Для продолжения нажмите любую клавишу . . .
FINISH
```

После проверки условия при наличии параметра переход не выполняется. Так как задан оператор ECHO OFF, то для оператора PAUSE текстовое сообщение тоже не отображается. Если был запущен CMD.EXE, то окно командной строки закроется. Если мы хотим получить это сообщение от команды PAUSE, то нужно включить вывод - ECHO ON и для других директив подавить вывод на экран команд (@). Например:

```
ECHO ON
@ECHO BATCH FILE
PAUSE TEST
@ECHO FINISH
@EXIT
```

Для такого файла получим на экране:

```
>ECHO ON
BATCH FILE
> PAUSE TEST
Для продолжения нажмите любую клавишу . . .
FINISH
```

При запуске командного файла без параметра, например:

```
>first.bat
```

На экране мы получим:

```
>ECHO OFF
BATCH FILE
FIRST.bat
VAR
MET1

FINISH - MET1
Для продолжения нажмите любую клавишу . . .
```

Здесь выполняется переход на метку по значению переменной и выводится другая информация. После завершения программы окно командной строки (CMD.EXE) не закроется.

## 5.6. Вложенные командные файлы

Вложенные командные файлы, по сути, являются процедурами. Они могут разрабатываться и отлаживаться автономно, могут иметь параметры. При выполнении вложенного командного файла операционная система запускает дополнительный командный процессор (CMD.EXE или COMMAND.COM). При завершении командного файла с помощью команды EXIT /b <число> можно передать код возврата ERRORLEVEL (значение определяется значением параметра <число>). Кроме того, из вложенного командного файла в основной могут быть переданы значения переменных, установленных командой SET. Ниже приведены примеры основного командного файла (SECOND.BAT) и вложенного командного файла (SUB.BAT). На этих примерах иллюстрируются возможности вызова вложенных файлов. В среде DosBox команда EXIT имеет упрощенный вариант (см. справку!)

Файл SECOND.BAT:

```
ECHO OFF
REM Call sub file
CALL SUB.BAT PAR1 , PAR2
ECHO Return from sub file
ECHO ERRORLEVEL = %ERRORLEVEL%
ECHO %VAR%
PAUSE
EXIT /b
```

Файл SUB.BAT:

```
REM Вложенный файл
ECHO Name batch=%0
ECHO Parameter 1 =%1
```

```
ECHO Parameter 2 =%2
REM Return Code = 5
Set VAR=Sample
EXIT /B 5
```

При запуске основного файла:

```
>second.bat
```

Получим следующий результат:

```
>ECHO OFF
Name batch=SUB.BAT
Parameter 1 =PAR1
Parameter 2 =PAR2
Return from sub file
ERRORLEVEL = 5
Sample
Для продолжения нажмите любую клавишу . . .
```

В данном примере показано как:

- Вызвать вложенный командный файл и передать в него параметры (**CALL SUB.BAT PAR1 , PAR2**);
- Установить переменную во вложенном файле (**Set VAR=Sample**) и прочитать ее значение в основном файле (**ECHO %VAR%**);
- Установить код возврата (**ERRORLEVEL**) во вложенном командном файле (**EXIT /B 5**) и прочитать его значение в основном файле(**ECHO ERRORLEVEL = %ERRORLEVEL%**).

Число вложений командных файлов практически не ограничивается под управлением CMD.EXE.

## 5.7. Примеры командных файлов

Ниже приведены два примера простых командных файлов для организации экранного меню. Для организации интерактивного взаимодействия здесь используются утилиты BE и CHOICE. Описание этих программ приведено в разделе данного пособия (см. раздел 6. Утилиты BE и CHOICE ).

В первом примере: экран сначала очищаем, затем выводим текстовое меню (пункты 1,2 и 3), запрашиваем ввод клавиши, выполняем ветвление (IF - GOTO), проверяя переменную ERRORLEVEL. По отдельным пунктам меню выполняем простые действия (ECHO и PAUSE). Если выбраны пункты 1 или 2 то повторяем все действия снова. По пункту 3 завершаем работу командного файла.

```
echo off
:menu
cls
echo 1. Режим 1
echo 2. Режим 2
echo 3. Выход
REM ЗАПРОС НАЖАТИЯ КЛАВИШИ
be ask "Выберете пункт (1,2,3)" '123' default=2 timeout=10

if ERRORLEVEL 3 goto 3
if ERRORLEVEL 2 goto 2
if ERRORLEVEL 1 goto 1
goto fin
:1
echo 1
pause
goto menu

:2
echo 2
```

```

pause
goto menu

:3
echo 3
pause
goto fin

:fin
ЕCHO Завершение программы

```

Блок схема данной программы приведена в разделе общего пособия [7] - 21.3.21.3  
Примеры блок-схем программ.

Во втором примере: экран сначала очищаем, затем выводим текстовое меню (пункты a, b и c), запрашиваем ввод клавиши, выполняем переход по переменной, построенной на основе ERRORLEVEL (GOTO %met%). По отдельным пунктам меню выполняем простые действия (ECHO и PAUSE). Если выбраны пункты “a” или “b” то повторяем все действия снова. По пункту “c” завершаем работу командного файла.

```

echo off
:menu
cls
echo a. One
echo b. Two
echo c. Exit
choice /C:abc /S Введите номер пункта меню

Set met=p%ERRORLEVEL%
goto %met%

:p1
echo a
pause
goto menu

:p2
echo b
pause
goto menu

:p3
echo c
pause
goto fin

:fin
ЕCHO Завершение программы

```

Результаты работы данных программ получите самостоятельно в режиме командной строки.

#### Практика.

1. Выполните все примеры данного раздела.
2. Создайте командный файл для 2-й лабораторной работы и оформите отчет в соответствии с требованиями.

В командных файлах отсутствуют операторы и директивы ввода информации. Это создает проблемы при создании интерактивных командных файлов. Существует несколько простых возможностей запросить информацию от пользователя, применяющего данный командный файл, для ввода данных. Это:

- Использование интерактивной утилиты BE (Batch Enhancer – расширитель командных файлов), созданной специально для этих целей,
- Использование специальной команды CHOICE, которая ранее входила в состав ОС, хотя и теперь может использоваться в режиме командной строки,
- Разработка и использование собственной программы, которая может быть вызвана в командном файле, и может передавать в его среду информацию.

Во всех перечисленных случаях ввод данных выполняется только с помощью одной переменной целого типа **ERRORLEVEL**, которая, по сути, представляет код возврата вызванной программы. Если разрабатывается собственная программа, то возврат может быть выполнен функцией `exit()` для языка Си или подфункцией `4Ch` прерывания `021h`. Данная переменная может быть использована в операторе **IF**, что обеспечивает ветвление в программе командного файла. Для этого предусмотрена специальная конструкция оператора **IF**. При формировании переменной **ERRORLEVEL**, а это единственная переменная целого типа в командных файлах, стандартного типа, мы можем использовать ее в следующих операторах:

```
IF ERRORLEVEL 5 GOTO M2
IF ERRORLEVEL 3 GOTO M3
IF ERRORLEVEL 4 GOTO M4
...
...
:M2
...
:M3
```

Переход на метку M2 выполняется в том случае, если значение **ERRORLEVEL** ≥ 5.

```
IF ERRORLEVEL 3 SET MET=M1
...
GOTO %MET%
...
:M1
```

Переход на метку M1 выполняется в том случае, если значение **ERRORLEVEL** ≥ 3.

Переменная **ERRORLEVEL** может быть использована и качестве составного имени оператора:

```
GOTO M%ERRORLEVEL%
```

Переход на метку M2 выполняется в том случае, если значение **ERRORLEVEL** = 2.

## 6.1. Утилита BE , назначение и использование

Справка для утилиты BE, полученная после выполнения команды - BE /?, содержит следующую информацию:

```
Batch Enhancer, Norton Utilities 6.2, Copyright 1992 by Symantec Corporation
New commands to enhance batch files.
BE command [parameters] [/DEBUG]
BE pathname [ [GOTO] label]
commands are:
    ASK          PRINTCHAR
    BEEP         REBOOT
    BOX          ROWCOL
    CLS          SA
    DELAY        SHIFTSTATE
```

EXIT	TRIGGER
GOTO	WEEKDAY
JUMP	WINDOW
MONTHDAY	
/DEBUG	Display the ERRORLEVEL code.
pathname	Name of a BE command file.
label	Label in file to start execution.

For more help on a specific command type:  
BE command ?

Для выполнения ЛР нам нужны две команды: ASK и BEEP. Команда ASK выводит на экран подсказку, ожидает нажатия клавиши и позволяет запросить код возврата ERRORLEVEL после ввода нужной клавиши. Справка BE ASK ? имеет вид:

```
BE ASK "< подсказка >" ['<список символов>'] [DEFAULT=<символ>]
[TIMEOUT=<число>] [ADJUST=< число >]
```

Где:

<prompt> – подсказка, выдаваемая перед запросом нажатия клавиши.

< список символов > – список символов для выбора альтернатив.

< символ > – клавиша, выбираемая по умолчанию.

< число > – для TIMEOUT время ожидания в секундах, через которое срабатывает клавиша, выбираемая по умолчанию.

#### Примеры.

```
be ask "Выберете пункт меню (1,2,3)" '123' default=1 timeout=4
```

- запрашивается выбор из набора (1,2,3), а через 4 секунды по умолчанию срабатывает нажатие “1”.

```
be ask "Введите символ (a,b,c)" 'abc' default=c timeout=10
```

- запрашивается выбор из набора (a,b,c), а через 10 секунд по умолчанию срабатывает нажатие “c”.

Второй режим работы утилиты – вывод определенного количества звуковых сигналов заданной частоты, длительности и длительности промежутков между ними. Справка BE BEEP ? имеет следующий вид:

```
Batch Enhancer, Norton Utilities 6.2, Copyright 1992 by Symantec Corporation

BEEP [switches]
or
BEEP command-file [/E]

Switches
/Dn Duration of the tone in n/18 seconds
/Fn Sound a tone of frequency n
/Rn Repeat the tone n times
/Wn Wait between tones n/18 seconds
/E Echo text in quotes following notes
```

Где:

/D<число> - длительность в <число>/18 seconds

/F<число> - частота звукового сигнала <число>

/R<число> - повтор звуков <число> раз

/W<число> - ожидание между звуками <число>/18 seconds

Во второй разновидности конструкции характеристики нот и звуков записываются в отдельный командный файл.

#### Пример.

```
be.exe beep /D18 /F200 /R5 /W18 /E
```

Выдача 5 звуковых сигналов с частотой 200 и длительностью в 1 секунду.

## 6.2. Утилита CHOICE, назначение и использование

Другая утилита CHOICE предназначена для только для запроса ввода клавиши. Подсказка по ней имеет вид:

```
CHOICE [/C[:]<символы выбора>] [/N] [/S] [/T[:]<клавиша>,<число>]
[<текст>]

/C[:]<символы выбора> - Specifies allowable keys. Default is YN
/N                      Do not display choices and ? at end of prompt string.
/S                      Treat choice keys as case sensitive.
/T[:]<клавиша>,<число>  Default choice to <клавиша> after ,<число>
seconds
<текст>                Prompt string to display

ERRORLEVEL is set to offset of key user presses in choices.
```

Где:

<символы выбора> - последовательность символов для выбора.

/N – не выводить подсказку доступных клавиш после текста подсказки

/S – рассматривать ввод символов без учета регистра

/T – использовать клавишу по умолчанию по истечению заданного времени.

<число> - количество секунд ожидания для использования клавиши по умолчанию.

<клавиша> - клавиша по умолчанию.

<текст> - текст подсказки для ввода клавиши

Примеры.

```
CHOICE /C:123 /T:2,05
```

Запросить ввод из трех альтернатив (1,2,3) использовать “2” после пяти секунд ожидания.

```
CHOICE /C:ABC /NВведите (А,В,С) :
```

Запросить ввод из трех альтернатив (А,В,С), не различать срочные и прописные, ввод выполняется после подсказки “ Введите (А,В,С):”.

## 6.3. Утилита GREP, назначение и использование

Очень полезная утилита для программистов – GREP.COM. Она входит в состав пакета VC++ 3.1. Эта утилита позволяет выполнить поиск информации в строках в текстовых файлах с выдачей имен и номеров строк, где эта информация найдена. Стандартная справка о работе и параметрах этой утилиты приведена ниже.

```
Turbo GREP Version 3.0 Copyright (c) 1992 Borland International
Syntax: GREP [-rlcnvidzuwo] searchstring file[s]

Options are one or more option characters preceeded by "-", and optionally
followed by "+" (turn option on), or "-" (turn it off). The default is
"+".

-r+ Regular expression search          -l- File names only
-c- match Count only                  -n- Line numbers
-v- Non-matching lines only           -i- Ignore case
-d- Search subdirectories                -z- Verbose
-u- Update default options             -w- Word search
-o- UNIX output format                 Default set: [0-9A-Z_]

A regular expression is one or more occurrences of: One or more characters
optionally enclosed in quotes. The following symbols are treated spe-
cially:

^ start of line                        $ end of line
. any character                        \ quote next character
* match zero or more                  + match one or more

[aeiou0-9] match a, e, i, o, u, and 0 thru 9
[^aeiou0-9] match anything but a, e, i, o, u, and 0 thru 9
```

Для поиска можно задавать маску, “дикие символы” и т.д. Можно выполнять поиск во множестве файлов (\*.cpp) и задавать разные режимы поиска.

Пример.

```
> GREP nr+ "class Point" *.hpp
```

Поиск описания класса “class Point” во множестве заголовочных файлов (\*.hpp). Отметим, что большинство файловых менеджеров имеет возможности такого поиска, причем характеристики поиска в них определяются настройками в оконном режиме.



Вы наверно уже поняли, что при работе в режиме командной строки приходится вводить много команд и выполнять операций переключения между каталогами, дисками. При работе в среде Windows также приходится выполнять аналогичные действия, но в этом случае Вы можете воспользоваться стандартным приложением – Windows Explorer. И в первом и во втором случае работать не очень удобно, трудоемко и возможны ошибки при выполнении операций над файлами. Для большего удобства предусмотрены специальные программы, которые называются файловыми менеджерами. Эти программы обеспечивают выполнение множества полезных операций и очень удобны в работе. Надеюсь, что даже при первом знакомстве Вы в этом сможете убедиться.

Разновидностей файловых менеджеров много. Они обеспечивают работу как в режиме эмуляции ДОС, так и в среде WINDOWS. Перечислим основные возможности файловых менеджеров:

- Управление файлами (копирование, перемещение, поиск и многие другие);
- Управление программами (запуск, настройка и т.д.);
- Редактирование текстовых файлов;
- Сортировка файлов;
- И многие другие возможности.

В данном разделе мы кратко познакомимся с тремя такими программами: **DosNavigator**, **FarManager** и. Кроме этих вариантов Вы можете встретить и другие файловые менеджеры: **Norton Commander**, **Volkov Commander** и другие. Функционально и по интерфейсу эти программы очень похожи друг на друга. Самое существенное отличие – это набор функциональных клавиш, которые используются для управления.

Общим для всех файловых менеджеров является то, что в окне программы обычно представлены две панели (два подокна), одна из которых является активной. Активность обычно выделяется цветом. Переключение между панелями осуществляется клавишей **TAB**. В активном окне выделен блочный курсор (выделяется цветом или рамкой). Не нужно путать этот курсор с курсором мыши. Блочный курсор выделяет объект (файл), над которым могут быть выполнены операции (копирования, перемещения, удаления и т.д.). Подсказка о возможных операциях (функциональных клавишах) обычно помещается в нижней строке окна файлового менеджера.

Второй особенностью файлового менеджера, является наличие командной строки, в которой можно вводить команды и запускать программы. Кстати, обычно запуск программы может быть выполнен простым нажатием клавиши ENTER, после выделения нужной программы с помощью блочного курсора. Командная строка аналогична командной строке в режиме эмуляции ДОС, под управлением командного процессора. Если файловый менеджер работает в среде WINDOWS, командная строка соответствует режиму ввода команд в меню пуск: ПУСК/Start=> Выполнить/Run => Ввод команды.

## 7.1. Dos Navigator

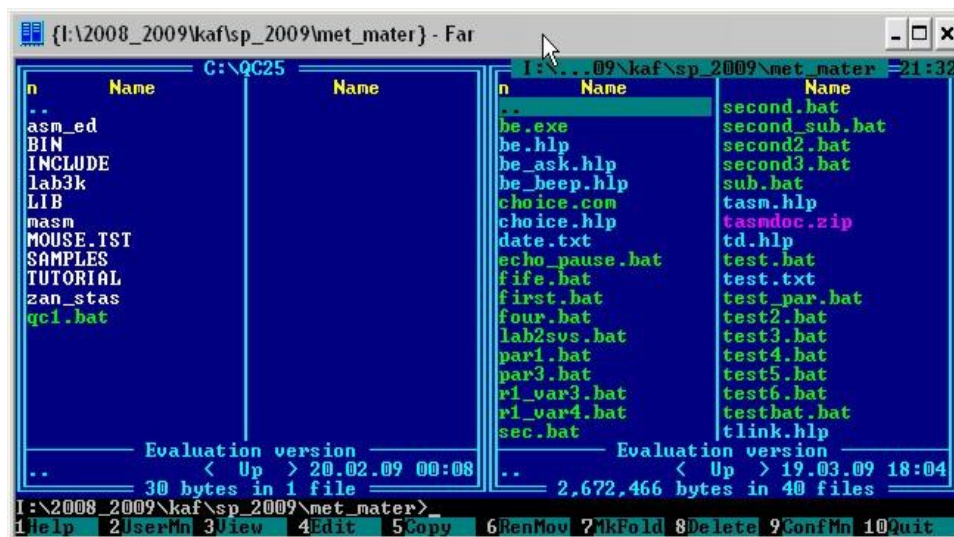
Окно файлового менеджера **Dos Navigator** (есть на сайте) представлено ниже. На левой панели представлен каталог (директория, папка) диска (c:)- c:\QC25\BIN. На правой панели каталог диска (i:).



Для выхода в меню можно использовать клавише **F10**. Для завершения программы необходимо воспользоваться клавишами **Alt+X**. Назначение функциональных клавиш для выполнения основных операций показано в нижней строке. При выполнении операций копирования и перемещения файлы будут копироваться из активной панели менеджера в пассивную панель. Вторая строка снизу представляет собой командную строку для ввода команд. Программа позволяет делать множество специальных настроек, которые значительно упрощают работу пользователя. Например, можно создать пользовательское меню (**F2**) для быстрого вызова нужных программ и выполнения команд.

## 7.2. Far manager

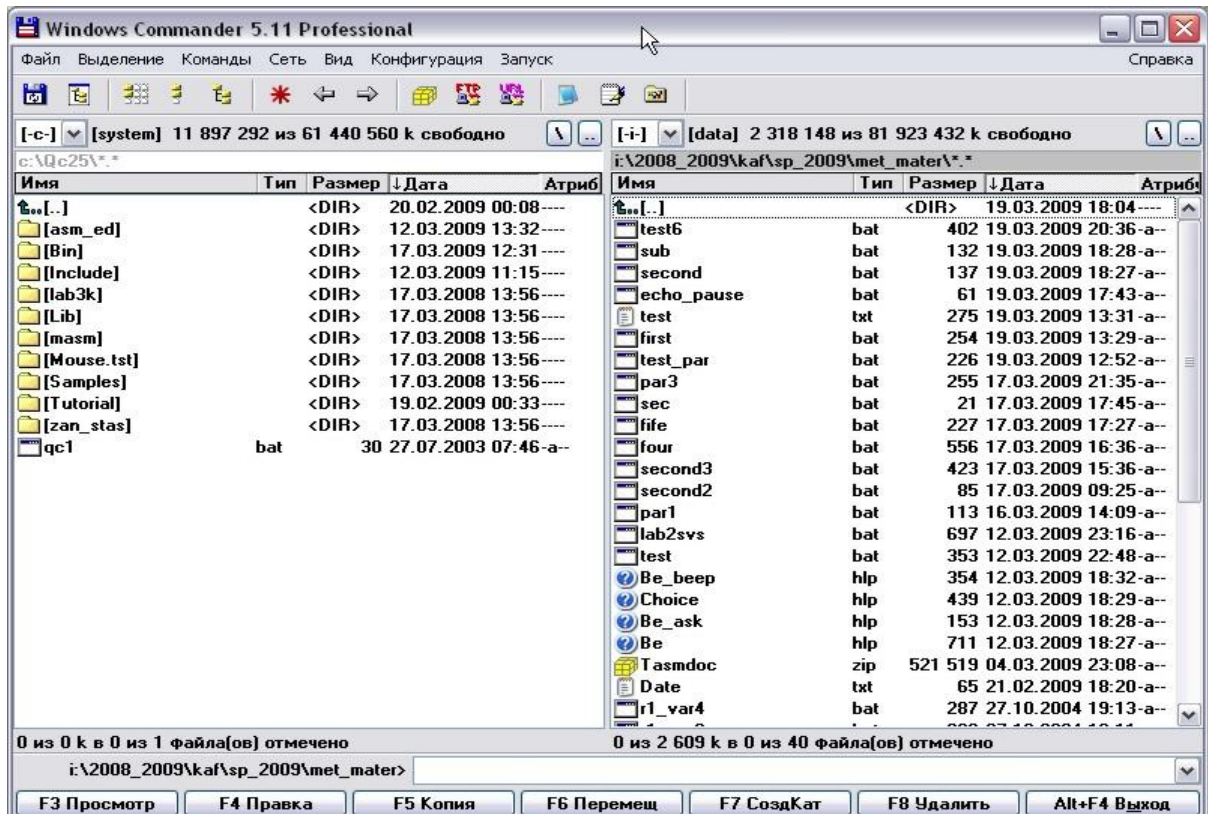
Окно файлового менеджера **Far manager** (есть на сайте) представлено ниже. На левой панели представлен каталог (директория, папка) диска (c:)- c:\QC25. На правой панели каталог диска (i:).



Завершение работы FAR выполняется клавишей **F10**. Для запуска меню используется **F9**. После нескольких сеансов работы в файл менеджере FAR Вы, несомненно, освоитесь и будете его использовать в своей работе.

### 7.3. Windows/Total Commander

Для работы в среде WINDOWS можно использовать **Windows Commander** или **Total Commander** (синонимы). Окно этой программы показано на рисунке ниже.



Достаточно внимательно посмотреть, чтобы увидеть сходство с другими файловыми менеджерами. Особенностью Windows Commander по сравнению с другими рассмотренными менеджерами является более активное использование мыши. Кроме того здесь соблюдены все стандарты Windows программ (меню и клавиатура).

В программах на языке Ассемблер можно и нужно использовать процедуры. Процедуры позволяют сделать программу более наглядной и компактной.

## 10.1. Процедуры, их назначение и применение

Процедуры целесообразно использовать даже тогда, когда к ней выполняется одно обращение. Это позволяет сделать основную программу более обзримой. При вызове процедур в нее необходимо передавать параметры. К сожалению, команда вызова процедура CALL не позволяет передавать параметры в процедуру явно. Для работы с процедурами необходимо выполнить следующие шаги:

- Спроектировать процедуру, что включает как минимум: определение функционального назначения процедуры, перечень входных и выходных параметров процедуры.
- Создать программу процедуры и оформить ее на языке Ассемблера.
- Отладить процедуру автономно.
- Написать в основной программе и отладить команды вызова процедуры.

Рассмотрим эти основные составляющие шаги для создания и применения процедур.

## 10.2. Описание процедур

Описание процедур выполняется на основе следующего синтаксиса:

```
<Имя>    PROC [ FAR | NEAR ]
           <Команда>
           <Команда>
           <Команда>
           <Команда>
           ...
           <Команда>
           RET[N | F]

<Имя>    ENDP
```

Процедуры могут быть двух типов: NEAR и FAR. Это указывается при описании процедуры в директиве PROC. По-умолчанию процедур является NEAR. Параметры <имя> в операторах PROC и ENDP должны быть одинаковыми. Они отличаются от меток, на них нельзя передавать управление. Оператор RET (RETN , RETF) осуществляет корректный выход из процедуры (для NEAR и FAR соответственно). Таких операторов выхода может быть несколько, и они не обязательно должны стоять в конце процедуры. Но должен быть, по крайней мере, один выход из процедуры. Описание процедуры можно разместить практически в любом месте программы. Но лучше их размещать в конце программы. Число процедур не ограничивается. Не разрешается вложенное описание процедур, хотя допустим вызов вложенных процедур.

## 10.3. Параметры процедур и вызов процедур

Вызовы процедур выполняются командой CALL:

```
; Вызов процедуры
CALL    <имя процедуры>
```

При вызове процедуры параметры не задаются. Параметры в/из процедуры входные и возвращаемые могут быть переданы следующими способами:

- Через глобальные переменные программы.
- Через регистры при вызове процедуры
- Через стек программы.

Передача через глобальные переменные выглядит так (глобальная переменная LET):

```
; Глобальная переменная
```

```

LET DB 'A'
...
; Вызов процедуры
CALL PUTCH
...
; Процедура вывод символа
PUTCH PROC
    MOV DL , LET
    MOV AH , 02H
    INT 021H
    RET
PUTCH ENDP

```

Передача через регистры выполняется так (регистр DL):

```

; Вызов процедуры
    MOV DL , LET
    CALL PUTCH
...
; Процедура вывод символа
PUTCH PROC
    MOV AH , 02H
    INT 021H
    RET
PUTCH ENDP

```

Передача через стек несколько сложнее для одного параметра и выполняется так :

```

; Вызов процедуры
    MOV DL , LET
    PUSH DX
    CALL PUTCH
    POP DX
...
; Процедура вывод символа
PUTCH PROC
    MOV BP , SP
    MOV DX , 2 + [BP]
    MOV AH , 02H
    INT 021H
    RET
PUTCH ENDP

```

Так как команды CALL и RET используют стек, то приходится самостоятельно получать данные из стека (MOV DX , 2 + [BP]), предварительно получив значение BP на основе SP.

Как и процедуры, бывают короткие и дальние вызовы. Короткие вызовы мы уже рассмотрели выше, так как процедуры были по умолчанию объявлены как короткие. Длинный вызов может быть сделан так:

```

; Поле для хранения длинного адреса
PA DW ?
...
; Подготовка длинного адреса
    LEA BX , TESTP
    MOV PA , BX
    MOV PA + 2 , DS
; Задание параметра
    MOV DL , 'B'
    PUSH DX
; Дальний вызов
    CALL DWORD PTR CS:[PA]
    POP DX
...
; Процедура для дальнего вызова

```

```

TESTP  PROC  FAR
        MOV  BP ,  SP
        MOV  DX ,  4 + [BP]
        MOV  AH ,  02H
        INT  021H
        RETF
TESTP  ENDP

```

Для дальнего вызова процедуры можно воспользоваться следующей конструкцией оператора CALL:

```

; Дальний вызов
CALL FAR PTR TESTP

```

## 10.4. Вложенные вызовы процедур

Из одной процедуры можно вызывать другие процедуры. Ограничений числа вложенных вызовов практически нет. Число вызовов ограничивается размером стека. Пример вложенных вызовов процедур приведен ниже:

```

; Процедура перевода строки
CRLF  PROC
        MOV  DL ,  10
        CALL PUTCH
        MOV  DL ,  13
        CALL PUTCH
        RET
CRLF  ENDP

```

В этой процедуре для вывода на экран символов перевода строки (10) и возврата каретки (13) дважды используется процедура PUTCH.

Примечание. При вложенных вызовах нужно следить за регистрами и стеком. При необходимости регистры нужно сохранять. Число записей в стек должно чётко соответствовать числу выборки данных из стека.

## 10.5. Пример программы с процедурами

Рассмотрим пример программы с процедурами. Ниже приводится исходный текст этого примера:

```

MYCODE SEGMENT 'CODE'
    ASSUME CS:MYCODE, DS:MYCODE
    LET  DB  'A'
    PA   DW  ?
start:
; Загрузка регистра сегмента данных
    PUSH CS
    POP  DS
; Вывод символа
    MOV  DL ,  LET
    CALL PUTCH
; Перевод строки
    CALL CRLF
; Подготовка длинного адреса
    LEA  BX ,  TESTP
    MOV  PA ,  BX
    MOV  PA + 2 ,  DS
; Задание параметра
    MOV  DL ,  'B'
    PUSH DX
; Дальний вызов
    CALL DWORD PTR CS:[PA]
    POP  DX

```



```

; Перевод строки
CALL CRLF
; Ожидание завершения программы
MOV AH, 01H
INT 021H
; Выход из программы
MOV AL, 0 ;выход из программы с возвращением errorlevel 0
MOV AH, 4CH
; Активизация системной функции для завершения программы
INT 21H
; Процедура перевода строки
CRLF PROC
MOV DL , 10
CALL PUTCH
MOV DL , 13
CALL PUTCH
RET
CRLF ENDP
; Процедура вывод символа
PUTCH PROC NEAR
MOV AH , 02H
INT 021H
RETN
PUTCH ENDP
; Процедура для дальнего вызова
TESTP PROC FAR
MOV BP , SP
MOV DX , 4 + [BP]
MOV AH , 02H
INT 021H
RETF
TESTP ENDP
; Конец сегмента
mycode ENDS
END start

```

Блок-схема данной программы приведена в разделе 21.3.21.3 Примеры *блок-схем программ*. Щелкните на номере раздела и найдете блок-схему.

В данном примере рассмотрены разные процедуры и их вызовы, которые мы уже рассмотрели ранее. Здесь продемонстрированы различные способы вызова коротких и дальних процедур (условно удаленных).

## 10.6. Листинг программы с процедурами

После компиляции мы получим следующий листинг программы. Действия по созданию программы были описаны выше в разделе 3. Ввод текста, компиляция, редактирование и отладка. С содержанием специальной информации листинга программы мы познакомимся позже (см. в разделе ).

```

Microsoft (R) QuickAssembler for QuickC Version 2.51      03/23/9

0000                                MYCODE SEGMENT 'CODE'
                                ASSUME CS:MYCODE, DS:MYCODE
0000 41                                LET DB 'A'
0001 0000                            PA DW ?
0003                                start:
                                ; Загрузка регистра сегмента данных
0003 0E                                PUSH CS
0004 1F                                POP DS
                                ; Вывод символа
0005 8A 16 0000 R                    MOV DL , LET
0009 E8 0030                        CALL PUTCH
                                ; Перевод строки

```

```

000C E8 0022 CALL CRLF
;Подготовка длинного адреса
000F 8D 1E 0041 R LEA BX , TESTP
0013 89 1E 0001 R MOV PA , BX
0017 8C 1E 0003 R MOV PA + 2 , DS
; Задание параметра
001B B2 42 MOV DL , 'B'
001D 52 PUSH DX
; Дальний вызов
001E 2E: FF 1E 0001 R CALL DWORD PTR CS:[PA]
0023 5A POP DX
; Перевод строки
0024 E8 000A CALL CRLF
; Ожидание завершения программы
0027 B4 01 MOV AH, 01H
0029 CD 21 INT 021H
; Выход из программы
002B B0 00 MOV AL, 0 ;выход из программы
с возвращением errorlevel 0
002D B4 4C MOV AH, 4CH
; Активизация системной функции для завершения программы
002F CD 21 INT 21H
; Процедура перевода строки
0031 CRLF PROC
0031 B2 0A MOV DL , 10
0033 E8 0006 CALL PUTCH
0036 B2 0D MOV DL , 13
0038 E8 0001 CALL PUTCH
003B C3 RET
003C CRLF ENDP
; Процедура вывод символа
003C PUTCH PROC NEAR
003C B4 02 MOV AH , 02H
003E CD 21 INT 021H
0040 C3 RETN
0041 PUTCH ENDP
; Процедура для дальнего вызова
0041 TESTP PROC FAR
0041 8B EC MOV BP , SP
0043 8B 56 04 MOV DX , 4 + [BP]
0046 B4 02 MOV AH , 02H
0048 CD 21 INT 021H
004A CB RETF
004B TESTP ENDP
; Конец сегмента
004B mycode ENDS
END start
Microsoft (R) QuickAssembler for QuickC Version 2.51 03/23/9

```

## Segments and Groups:

N a m e	Length	Align	Combine Class
MYCODE . . . . .	004B	PARA	NONE 'CODE'

## Symbols:

N a m e	Type	Value	Attr
CRLF . . . . .	N PROC	0031	MYCODE
= 000B			



```

      LET . . . . . L BYTE      0000 MYCODE
      PA . . . . . L WORD      0001 MYCODE
      PUTCH . . . . . N PROC    003C MYCODE      Length
= 0005

      START . . . . . L NEAR    0003 MYCODE
      TESTP . . . . . F PROC    0041 MYCODE      Length
= 000A

      @CPU . . . . . TEXT      0101h
      @FILENAME . . . . . TEXT  fourd
      @VERSION . . . . . TEXT  520

Line Numbers:
fourd.asm
(1) 1:3 2:4 3:5 4:6 5:7 6:8 7:9 8:10 9:11 10:12 11:13 12:14 13:15 14:16
15:17
16:18 17:19 18:20 19:21 20:22 21:23 22:24 23:25 24:26 25:27 26:28 27:29
28:30
29:31 30:32 31:33 32:34 33:35 34:36 35:37 36:38 37:39 38:40 39:41 40:42
41:43
42:44 43:45 44:46 45:47 46:48 47:49 48:50 49:51 50:52 51:53 52:54 53:55
54:56
55:57 56:58 57:59 58:60 (0)

      58 Source   Lines
      58 Total    Lines
      17 Symbols

      41160 Bytes symbol space free

      0 Warning Errors
      0 Severe Errors

```

Результат работы данной программы, которая, кстати, почти соответствует требованиям 3-й ЛР по курсу очень простые:

**A**  
**B**

После вывода 2-х символов, каждый на отдельной строке, программа ожидает нажатия любой клавиши и затем заканчивается.

## 16.1. Команда XLAT

**XLAT** [адрес таблицы символов]

## XLATB

1. В регистр **AL** заносится символ для перекодировки
2. Длинный адрес **DS:BX** определяет адрес таблицы перекодирования символов, длина которой не может превышать 256 символов.
3. После выполнения команды **XLAT** в **AL** перемещается символ из таблицы по адресу **[BX + AL]** или адрес таблицы плюс **AL**.

```
hex      DB      '0123456789ABCDEF'          ; Таблица перекодировки
String   db      '$',0
...
```

```
; XLAT
PUSH DS
POP ES
LEA DI, String ; Строка для копирования String
mov bx, OFFSET hex ; Указатель DS:BX на таблицу
mov ax, 0FAAh ; Число ддля перевода в AX
mov cx, 2 ; Число байтов 2 – параметр цикла

loop1: xchg ah, al ; Байты выводятся в обратном порядке
push ax ; Сохраняем исходное
shr al, 1 ; Сдвигаем вправо на 4 разряда
shr al, 1
shr al, 1
shr al, 1
xlat ; Получить ASCII код в AL
stosb ; Копируем в строку String (DS:[DI]) и DI++
pop ax ; Восстанавливаем число
push ax ; И сохраняем снова
and al, 00001111b ; Маскируем старшую шест. цифру
xlat ; Получить ASCII код в AL
stosb ; Копируем в строку String (DS:[DI]) и DI++
pop ax ; Восстанавливаем число
loop loop1 ; Переходим к младшему байту (он был первым)
mov ax, 'h' ; добавляем 'h' в AX
stosw ; копируем два
байта (h и 0) в строку
```

```

; за-
меной порядка в памяти
; Выводим строку на экран
MOV AH, 09H
LEA DX, STRING
INT 021H

```

В результате работы этого фрагмента программ на экране мы увидим:

```
0FAAh
```

Команда XLAT занимает в памяти всего 1 байт. Подобную операцию перекодировки, при всех прочих равных условиях, можно было бы выполнить и так:

```

MOV AH, 0H
MOV SI, AL
MOV AL, [BX][SI]

```

Или:

```

ADD BX, AL
MOV AL, [BX]

```

Однако, в первом случае размер программы в байтах равен 6-ти байтам, а во втором случае 4-м байтам и портится значение регистра BX. Поэтому применение команды XLAT является предпочтительным.

## 16.2. 16.2 Команда MOVS, префикс REP и команды цепочек

Команда **MOVS** и ее разновидности используется для перемещения цепочек байтовых чисел. Синтаксис команды:

```
MOVS <приемник> , <источник>
```

Или

```
MOVS
```

Операнды в команде при программировании на языке Ассемблер играют роль подсказки и в операции не участвуют. Предусмотрены команды **MOVSB** (байты), **MOVSW** (слова) и **MOVSD** (двойные слова только для 386 МП и старше). Источник определяется адресом на **DS:SI**, а приемник определяется адресом на **ES:DI**. Данные регистры перед выполнением команды должны быть загружены. Для повторения команды используется специальный префикс **REP**, определяющий циклический режим повторения команды и занимающий в ОП 1 байт. Число повторений определяется следующим образом:

- Числом, которое предварительно заносится в регистр **CX**.
- Длинной строки источника (конец строки задается нулевым числом – Null terminated String).

Примечание. Для некоторых подпрограмм ОС символом конца строки считается символ доллара '\$'. Это нужно учитывать при разработке программ. Например, функция вывода строки на экран (AH = 9, INT 21h) завершает вывод строки до первого символа '\$'. Если в программе возникает необходимость вывода на экран этого символа ('\$'), то нужно воспользоваться процедурой (AH = 2, INT 21h).

При выполнении команды **MOVSB** в циклическом режиме выполняется автоматическая коррекция индексных регистров **SI** и **DI**, определяющих смещения источника и приемника. Эти регистры в циклах изменяются на 1, 2 или 4 в зависимости от типа команды. Кроме того, при этом важным является значение флага **DF** (direction flag - флаг направления). Если флаг установлен в 0, то пересылка цепочки выполняется слева направо, начиная со значений младших адресов к старшим. Если флаг установлен в 1, то пересылка цепочки выполняется справа налево, начиная со значений старших адресов к младшим. Значения регистров **SI** и **DI** будут либо увеличиваться (**DF**=0) либо уменьшаться (**DF**=1) на заданное значение. Завершение цикла повторений происходит при достижении значения истины любого из условий (в счетчике **CX** 0 или достигнут конец строки).

Ниже приводится пример использования команды пересылки цепочки символов. Исходные строки в программе описаны так:

```
STRINGS db      '1234567890  $',0
STRINGD db      '                                $',0
```

Для пересылки данных используется следующий фрагмент:

```
CLD                ; Сброс флага направления
PUSH DS
POP  ES            ; Обязательная загрузка ES
LEA DI,STRINGD    ; загрузка адреса приемника
LEA SI,STRINGS    ; загрузка адреса источника
MOV CX, 10        ; Максимальное число повторений
; Команда MOVSB
REP  MOVSB
```

В программе первоначально устанавливается в 0 флаг направления. Далее формируется значение сегментного регистра **ES**. В нашем случае пересылка выполняется в пределах одного сегмента, поэтому **ES = DS**. В принципе значение регистра **ES** может быть любым, то есть пересылка может быть между сегментами. Далее с помощью команды **LEA** формируются начальные адреса - смещения источника (**SI**) и приемника (**DI**) и заполняется число циклов **CX=10**. В нашем случае нулевой символ в строке находится на 14-й позиции, поэтому первым сработает счетчик на регистре **CX**. Если вывести исходные данные в отладчике QC25, то увидим:

```
&StringS,s ♦ "1234567890  $"
&StringD,s ♦ "                                $"
```

Отладчик заканчивает вывод по нулевому символу. После работы программы получим:

```
&StringS,s ♦ "1234567890  $"
&StringD,s ♦ "1234567890      $"
```

Было скопировано ровно 10 байт. Если установить счетчик циклов **CX = 15**, то получим:

```
&StringS,s ♦ "1234567890  $"
&StringD,s ♦ "1234567890  $"
```

```
...
    MOV CX, 15
...
```

В этом случае циклы будут завершены по условию конца строки в источнике (14 символ равен 0). Если **CX = 3**, то получим:

```
&StringS,s ♦ "1234567890  $"
&StringD,s ♦ "123          $"
```

```
...
    MOV CX, 15
...
```

Если признак направления установлен в 1 (командой **STD**), то при пересылке цепочки регистры **DI** и **SI** на каждом шаге будут уменьшаться. Пусть исходные строки для перемещения имеют вид:

```
STRINGS db      '1234567890$',0
STRINGD db      '                                $',0
```

Тогда можно выполнить следующий фрагмент программы:

```
;    MOVSB        байты обратное направление STD
    STD
;    Направление STRINGS => STRINGD
    LEA DI, STRINGD + 9
    LEA SI, STRINGS + 9
    MOV CX, 8
REP  MOVSB
```

Так как установлен признак направления в 1 (**STD**) перемещение будет выполняться с младших байтов цепочки. Результат будет таким:

```
&STRINGS,s ♦ "1234567890  $"
&STRINGD,s ♦ " 34567890      $"
```

Если префикс **REP** перед командой **MOVSB** отсутствует, то команда будет выполнена один раз. Например:

```
CLD
```

```

...
MOV CX, 10
; Если REP нет, то 1 раз
MOVSB

```

Результат будет таким:

```

&STRINGS,s ♦ "1234567890 $"
&STRINGD,s ♦ "1          $"

```

Даже при установке регистра CX в 10. Для управления командой MOVSB могут использоваться и другие префиксы: REPE/REPNE и REPZ/REPNZ. Эти признаки ориентируются на значения передаваемых данных и для команды MOVS не действуют.

Команда **COMPSB** и ее разновидности используется для сравнения цепочек байтовых чисел. Синтаксис команды:

**COMPSB <приемник> , <источник>**

Или

**COMPSB**

Пусть мы имеем описание двух строк, которые равны:

```

Strs  db  '123$'
      db  0
Strd  db  '123$'
      db  0
...

```

Тогда, после выполнения фрагмента программы:

```

; Команда CMPSB
    PUSH DS
    POP  ES          ; Обязательная загрузка ES
    CLD
    LEA  DI, Strd
    LEA  SI, Strs
    MOV  CX, 10
REP  CMPSB
    JE  M5
    XOR AX,AX

```

**M5:**

Будет выполнен переход на метку M5, так как будут сравниваться строки до появления в исходной строке нулевого символа. Если мы специально нарушим равенство строк, как это сделано в следующем фрагменте:

```

; CMPSB REP
    CLD
    LEA  DI, Strd
    LEA  SI, Strs
    MOV  Strd + 1 , '*'
    MOV  CX, 10
REP  CMPSB
    JE  M6
    XOR AX,AX

```

**M6:**

то переход на метку M6 не произойдет. При использовании префикса REPNE процесс проверки остановиться на первом отличном символе:

```

; CMPSB REPNE
    CLD
    LEA  DI, Strd
    LEA  SI, Strs
    MOV  Strd + 1 , '*'
    MOV  CX, 10
REPNE CMPSB
    JNE M61
    XOR AX,AX

```

**M61:**

в регистре CX мы получим значение 9, которое показывает, что в строках равны только первые символы (10 - 9). После восстановления строки (возвращаем '2' на место) при префиксе REPE ( проверка эквивалентности строк) получим переход на метку M62:

```
; CMPSB REPE
    CLD
    LEA DI, Strd
    LEA SI, Strs
    MOV Strd + 1, '2'
    MOV CX, 10
REPE    CMPSB
    JE M62
    XOR AX, AX
```

**M62:**

Команда **SCASB** и ее разновидности используется для поиска в цепочке значения помещенного в регистре AL(AX). Синтаксис команды:

**SCASB <приемник>**

Или

**SCASB**

При поиске символа или числа (SCASW) в регистр AL (AX) предварительно заносим значение для поиска и выполняем команду с префиксом повторения:

```
; SCASB REP
    CLD
    LEA DI, Strd
    MOV CX, 10
    MOV AL, '2'
    MOV CX, 10
REP    SCASB
```

После ее выполнения в регистре DI получим адрес найденного символа. Для следующей программы с префиксом REPNE:

```
; SCASB REPNE
    CLD
    LEA DI, Strd
    MOV CX, 10
    MOV AL, '3'
    MOV CX, 10
REPNE    SCASB
```

В DI получим значение адреса символа следующего после '3'.

```
; SCASB REPE
    CLD
    LEA DI, Strd
    MOV AL, '2'
    MOV CX, 10
REPE    SCASB
```

При использовании REPE также в регистре DI получим адрес найденного символа.

Команда **LODSB** и ее разновидности используется для загрузки в AL(AX) значения из цепочки символов. Синтаксис команды:

**LODSB <источник>**

Или

**LODSB**

```
; LODS
    CLD
    LEA SI, Strs
    LODSB ; Загрузка 1-го байта строки в AL
    ...
    LODSB ; Загрузка 2-го байта строки в AL
```

Такие команды можно использовать в цикле для последовательной обработки отдельных символов (можно с успехом применить в 5-й ЛР) для обработки введенной строки.

Команда **STOSB** и ее разновидности используется для записи в цепочку значения из AL(AX). Синтаксис команды:

**STOSB <приемник>**

Или

**STOSB**

Пример фрагмента программы с повторением для записи в строку:

```
; STOSB
    CLD
    LEA DI, Strd
    MOV AL, '#'
    MOV CX, 4
REP     STOSB
```

После выполнения данного фрагмента в строке **Strd** мы получим:

**&Strd,s ♦ "####"**

Такие команды можно использовать в цикле для последовательной обработки отдельных символов (можно с успехом применить в 5-й ЛР) для запоминания строки.

### 16.3. 16.3 MUL/IMUL и DIV/IDIV

Команды **MUL/IMUL** и **DIV/IDIV** используются для выполнения операций умножения и деления целых чисел. Особенностью команды умножения является то, что полученный результат может иметь двойную размерность (8 -> 16 и 16 -> 32). Особенностью команды деления является наличие двух результатов (частного и остатка). Это влияет на выполнение команд, в частности на использование регистров. Парные команды **MUL/IMUL** и **DIV/IDIV** отличаются учетом (**IMUL, IDIV**) или не учетом (**MUL, DIV**) знаков чисел при выполнении операций. Для примеров программ выполнения операций нам нужны следующие описания:

```
WORD1  DW 100
BYTE1  DB 100
REZ     DW 0
REZO    DW 0
REZD    DD 0
```

Команда для умножения **MUL** слов (16 бит) без учета знака имеет вид:

```
; Тест команды MUL ( на слово)
    MOV AX, 1000
    MOV BX, 1000
    MUL BX           ; умножение
    MOV WORD PTR REZD, AX
    MOV WORD PTR REZD+2, DX
```

Результат операции получается на двух регистрах: в **AX** – младшие байты, а на **DX** – старшие байты. Напомню, что в оперативной памяти они записаны в обратном порядке! Результат получается в виде:

**REZD,d ♦ 1000000**

Умножение слов можно выполнить с использованием переменной из памяти:

```
; Память WORD
    MOV AX, 1000
    MUL WORD1       ; 100
    MOV WORD PTR REZD, AX
    MOV WORD PTR REZD+2, DX
```

Результат получается в виде:

**REZD,d ♦ 100000**

При умножении на байтовую переменную в регистре (BL) результат должен поместиться в регистр слова (AX), при этом множимое расположено в регистре AL:

```
; умножение на регистр BL
    XOR AX,AX
    MOV AL , 200
    MOV BL , 200
    MUL BL
    MOV REZ , AX
```

Результат получается в виде:

```
REZ,u * 40000
```

Если после выполнения операции старший регистр не нулевой (AH != 0), то формируются флаги OF (переполнения) и CF (переноса). Значения этих флагов можно использовать в командах условной передачи управления. Можно умножать на байт из памяти, например:

```
; Память BYTE
    MOV AL , 200
    MUL BYTE1 ; 100
    MOV REZ , AX
```

Результат получается в виде:

```
REZ,u * 20000
```

Если при умножении важен знак, то нужно использовать команду IMUL. Например:

```
; IMUL со знаком
    MOV AX , -100
    IMUL WORD1 ; 100
    MOV REZ , AX
```

Результат получается в виде:

```
REZ,d * -10000
```

При знаковом умножении слов

```
; IMUL слова
    MOV AX , -1000
    IMUL WORD1 ; 100
    MOV WORD PTR REZD , AX
    MOV WORD PTR REZD+2 , DX
```

Результат получается на регистрах AX:DX в виде:

```
REZD,d * -100000
```

Команда деления может выполняться без учета знака (DIV) и с учетом знака (IDIV). При операции короткого деления делимое располагается на регистре AX (16 бит), а делитель на заданном регистре размером 8 бит (в примере BL) или в памяти. Результат получаем в регистрах AL(частное) и AH(остаток). В примере ниже:

```
; DIV короткое
    MOV AX , 105
    MOV BL , 10
    DIV BL
    MOV REZCH , AL
    MOV REZOC , AH
```

Получим результат:

```
REZCH,u * 10
```

```
REZOC,u * 5
```

При операции длинного деления делимое располагается на регистрах AX:DX (32 бит), а делитель на заданном регистре размером 16бит (в примере BX). Результат получаем в регистрах AX(частное) и DX(остаток). В примере ниже:

```
; DIV длинное
    MOV AX , WORD PTR REZD ; 100005 - DD
    MOV DX , WORD PTR REZD+2
    MOV WORD PTR REZD , AX
    MOV WORD PTR REZD+2 , DX
```



```
MOV BX, 100
DIV BX
MOV REZ OCD, DX
MOV REZ CHD, AX
```

Получим результат:

```
REZ CHD, u ♦ 1000
REZ OCD, u ♦ 5
```

Операции деления выполняются и над знаковыми числами (IDIV), при этом знак устанавливается у частного. Надеюсь, что Вы сами придумаете и отладите примеры для изучения работы этой команды. Нужно также учитывать, что в современных микропроцессорах (старше 386) появились и новые возможности команд деления и умножения. Изучите эти возможности самостоятельно.

## 16.4. 16.4 Команды: TEST, CMP и AND

Команды **TEST**, **CMP** и **AND** используются часто для подготовки ветвления в программах на языке Ассемблер. Команда сравнения **CMP** позволяет сравнить два значения и подготавливает флаги для команды, следующей за ней. Это обычно команда условного перехода (JE, JNE, JL и других). Пример ниже иллюстрирует сравнение двух байтовых чисел (5 и 7) на регистре AL:

```
; Команда CMP
XOR AX, AX
MOV AL, 5
CMP AL, 7
JNE M3
MOV AH, 5
```

M3:

Так как числа неравны, то выполняется переход на метку M3. В следующем примере:

```
; Команда CMP
XOR AX, AX
MOV AL, 5
CMP AL, 5
JE M4
MOV AH, 5
```

M4:

Числа равны (5 и 5) и в этом случае тоже выполняется переход на метку M4.

Команда **AND** является командой поразрядного логического умножения. В примере логического умножения:

```
; Команда AND
XOR AX, AX
MOV AL, 00Ah
AND AL, 0A0h
JZ M1
MOV AL, 0Ah
```

M1:

После выполнения на регистре AL получится 0, поэтому выполнится переход на метку M1. Команда **TEST** работает также как и команда **AND**, но значение регистра не изменяется (не портится). Поэтому в примере:

```
; Команда TEST
XOR AX, AX
MOV AL, 5
TEST AL, 2
JZ M2
MOV AH, 5
```

M2:

Переход на метку M2 выполнен не будет, а значение регистра AL останется прежним.

## 16.5. 16.5 Команды: AAA , CBW и CWD

Команда AAA используется для десятичной коррекции для специального представления чисел в неупакованной десятичной форме. В этой форме каждый байт используется для хранения одной десятичной цифры.

```
; Команда AAA
MOV AX , 0Ch
AAA
```

До выполнения команды AAA на регистре AX мы имеем 000Ch, а после выполнения получим 0102h.

Команды CBW и CWD используются для преобразования форматов представления чисел с учетом знака (байта в слово и слова в двойное слово). Для пояснения команд нам потребуется переменная типа двойного слова (result).

```
result DD 0
...
```

Пример команды CBW:

```
; Команда CBW
MOV AH, 7
MOV AL, 5
CBW ; байт в слово
```

До выполнения команды на регистре AX мы имеем 0705h, а после выполнения команды получим 0005h. Для отрицательных чисел знаки и представление чисел тоже сохраняется:

```
; Команда CBW
MOV AH, 7
MOV AL, -5
CBW ; байт в слово
```

До выполнения команды на регистре AX мы имеем 07FBh, а после выполнения команды получим FFFBh.

В команде CWD используется дополнительно к AX и регистр DX. Поэтому для фрагмента программы, показанного ниже:

```
; Команда CWD
MOV DX, 7
MOV AX, -5
CWD
MOV WORD PTR result[2], DX
MOV WORD PTR result[0], AX
```

в регистре AX получим FFFBh, в регистре DX FFFFh, а значение переменной будет таким:  
result, d ← -5

## 16.6. 16.6 Команды: ADC и SBB

Команды ADC и SBB используются для сложения и вычитания с учетом флага переноса от предыдущей операции. Для сложения (ADC) добавляется перенос, если он был, а для вычитания (SBB) вычитается. Ниже приводятся примеры для сложения и вычитания четырех-байтовых чисел.

Сложение:

```
long1 DD 10000
long2 DD 60000
result DD 0
...
```

```
; Команда ADC
MOV AX, WORD PTR long1[0] ; AX = low word, long1
MOV DX, WORD PTR long1[2] ; DX = high word, long1
ADD AX, WORD PTR long2[0] ; Add low word, long2
ADC DX, WORD PTR long2[2] ; Add high word, long2
```

```
MOV     WORD PTR result[2], DX
MOV     WORD PTR result[0], AX
```

Получим:

```
result,d * 70000
```

Вычитание:

```
long1 DD 40000
long2 DD 10000
result DD 0
```

...

; Команда SBB

```
MOV     AX, WORD PTR long3[0]    ; AX = low word, long1
MOV     DX, WORD PTR long3[2]    ; DX = high word, long1
SUB     AX, WORD PTR long4[0]    ; Add low word, long2
SBB     DX, WORD PTR long4[2]    ; Add high word, long2
MOV     WORD PTR result[2], DX
MOV     WORD PTR result[0], AX
```

Получим:

```
result,d * 30000
```

## 16.7. 16.7 Команды: LOOP/LOOPE/LOOPNE

Команды организации циклов позволяют повторить фрагмент программы заданное число раз. Это число задается на регистре CX. На каждом шаге цикла значение CX уменьшается на 1 и проверяется на нуль. При достижении нуля цикл заканчивается. Фрагмент программы с циклом:

```
; Команда LOOP
CLD
LEA DI, Strd
MOV CX, 10
MOV AL, '*'
MLOOP:
STOSB
LOOP MLOOP
; конец цикла
MOV AL, 0
STOSB
```

В результате получим:

```
&Strd,s * "*****"
```

Другие команды циклов проверяют дополнительно значение флага нуля (ZF), который устанавливается арифметическими командами. Должны быть выполнены одновременно два условия. Следующий цикл выполняется всего один раз:

```
; Команда LOOPE LOOPZ
CLD
LEA DI, Strd
MOV CX, 10
MOV AX, 0
MLOOPE:
STOSB
ADD AX, 1
LOOPE MLOOPE
```

После команды ADD флаг нуля устанавливается в 1 (не равно нулю) и цикл завершается. Аналогично для обратной команды LOOPNE цикл выполнится один раз:

```
; Команда LOOPNE LOOPNZ
CLD
LEA DI, Strd
MOV CX, 10
MOV AX, 0
```

**MLOOPNE :****STOSB****ADD AX, 0****LOOPNZ MLOOPNE**

так как после добавления в AX нуля флаг нуля будет сброшен в 0. Проверьте также работу команд циклов: LOOPZ и LOOPNZ.

Практика.

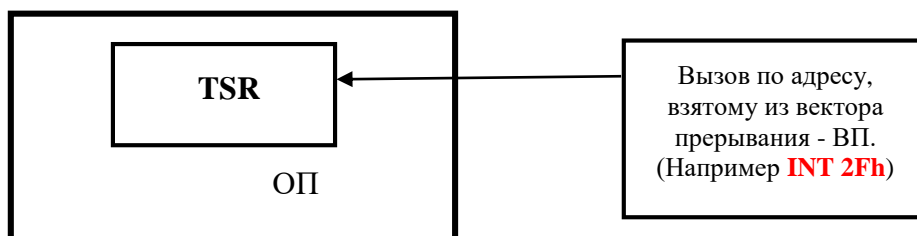
1. Выполните в режиме отладки фрагменты программ для команды XLAT.
2. Примените команду XLAT при разработке программы 4-й ЛР.
3. Выполните в режиме отладки фрагменты программ для команды MOVSB.
4. Примените команду MOVSB при разработке программы 8-й ЛР.
5. Выполните в режиме отладки фрагменты программ для команд DIV и MUL.
6. Выполните в режиме отладки фрагменты программ для команд TEST и AND.
7. Выполните в режиме отладки фрагменты программ для команд CBW и CWD.
8. Выполните в режиме отладки фрагменты программ для команд ADC и SBB.
9. Выполните в режиме отладки фрагменты программ для команд LOOP и LOOPE/LOOPNZ.

## 18.1. 18.1 Курсовая работа по СП

В текущем семестре (теперь в 4-м) студенты выполняют курсовую работу, связанную с разработкой резидентной программы (драйвера) и оформлением комплекта документации на программный продукт. Требования к КР и варианты КР размещены на сайте преподавателя. В этом семестре в рамках планового домашнего задания студенты должны разработать техническое задание (ТЗ) на основе требований к разработке документации также представленных на сайте. В данном разделе дается дополнительная полезная информация для разработки о реализации курсовой работы.

## 18.2. 18.2 Резидентные программы

Резидентные программы (**TSR** – “**T**erminate and **S**tay **R**esident”) – это программы, которые остаются в оперативной памяти (ОП) после их специального завершения. Эти программы могут работать параллельно с другими программами ДООС (псевдопараллельно). Обращение к этим программам выполняется через механизм прерываний (программные или аппаратные прерывания).



Для правильной работы резидентной программы (TSR) необходимо выполнить следующие действия и процедуры:

- Проверить повторную загрузку данного нового резидента.
- Загрузить резидентную программу в оперативную память.
- Выдать сообщение о корректной загрузке резидента в память.
- Записать в вектор прерываний (см. ниже) адрес резидента.
- Прочитать и правильно использовать параметры запуска резидента.
- Обеспечить вызов старого обработчика прерываний по данному адресу, если такое необходимо.
- Правильно отработать в процедуре резидента заданные функции.
- Обеспечить связь с резидентной программой с помощью клавиатуры, программой или другим способом.
- Обеспечить, при необходимости корректную выгрузку резидентной программы и освобождение занимаемой резидентом памяти.
- Корректно восстановить работу старых обработчиков данного прерывания, восстановить адреса старых обработчиков в векторе прерываний.
- Выдать сообщение о завершении работы резидентной программы.
- Вызвать справку о работе программы
- Выполнить варианты требования КР СП.

В данном разделе мы рассмотрим основные действия и процедуры, необходимые для построения резидентной программы и ее корректной работы.

### 18.3. 18.3 Вектор прерываний

Вектор прерываний (ВП) – это область оперативной памяти, содержащей адреса обработчиков прерываний. Каждый адрес занимает 4 байта (far адрес обработчика, включая адрес сегмента – 2б и адрес смещения – 2б). Область ВП имеет размер 1024 байта (1 Кб), поэтому максимально можно записать 256 адресов, т.е. возможно обрабатывать 256 различных прерываний. Вектор прерываний находится в начале ОП (для реального режима) по адресам 0000:0000 – 0000:03FF (см. рисунок ниже).

<far адрес обработчика>	} <b>0000:0000</b> – начало области <b>ВП</b> Каждый адрес - 4 байта ... <b>← ОБЛАСТЬ ВЕКТОРА ПРЕРЫВАНИЙ</b>
...	
...	
...	
<far адрес обработчика или блока>	} <b>0000:03FF</b> – $1024 \div 4 = 256$ – конец области <b>ВП</b>
<far адрес обработчика>	
<системные области >	} 0000:0400 – буферы, данные ОС и т.д.
<области п/п ДОС и BIOS>	
Пользовательские резидентные программы и драйверы	
Пользовательские программы	
...	
...	
Видеобуфер	} B800:0000 – буфер дисплея (видеопамять)
<системные области >	
<Буферы экрана для разных режимов>	
<системные области >	
...	
...	
Идентификационный № ЭВМ (2б)	} F000:FFFF – IBM Computer Type
HMA – High Memory Area	
	} FFFF:FFF0 – конец области HMA

Вычисление адреса конкретного обработчика в векторе прерываний выполняется умножением его номера (прерывания) на четыре: например, для прерывания 09H мы получим – 0000:0024H (или десятичное смещение 36). Доступ к адресам обработчиков прерываний может быть прямым (вычисляем адрес) или с помощью системных функций (25h и 35h прерывания ДОС 021h). Эти возможности рассмотрены ниже.

Все остальные детали распределения ОП вы можете посмотреть в справочниках ДОС – General Memory Map, которые расположены на сайте или в литературе (ЛР № 1).

### 18.4. 18.4 Структура резидентной программы (2-е части)

Обычно резидентная программа (далее мы будем также пользоваться термином резидент) первоначально состоит из двух частей. Они показаны на рисунке ниже.

<b>Резидентная часть программы</b>	- <b>ЧАСТЬ 1</b> находится постоянно в памяти и вызывается только посредством прерываний программных и аппаратных.
<b>Временная часть программы, используемая при инициализации резидента</b>	- <b>ЧАСТЬ 2</b> работает только при запуске резидента или проверке повторного его запуска или при выполнении процедуры выгрузки резидента.

При завершении установки резидента оперативная память под временную часть обычно освобождается, поэтому команды и данные из этой части нельзя использовать в процедурах резидентной части. В состоянии проверки повторности загрузки резидентной программы в ОП может находиться одновременно две части резидентной, так как программа запускается повторно.

## 18.5. 18.4 Понятие прерывания и назначение прерываний

**Прерывания** – это специальный механизм, позволяющий операционной системе выполнять свои функции (управление ресурсами). Они обеспечивают связь аппаратной части компьютера и обрабатываемых программ. Кроме того, механизм прерываний обеспечивает выполнение множества стандартных функций для управления программами и устройствами.

**Прерывания** – это автоматически инициируемый программный процесс, временно переключающий микропроцессор на выполнение другой программы. После обработки прерывания прерванный процесс автоматически будет продолжен.

В момент возникновения прерывания в стеке запоминается точка возврата (следующая после прерывания команда) и содержание регистра флагов. После завершения обработки прерывания из стека восстанавливается регистр флагов (Flags) и прерванная программа переключается на выполнение следующей команды.

Прерывания подразделяются на программные и аппаратные. Данная классификация основана на причине инициирования прерывания. В случае программных прерываний они возникают после выполнения специальной команды (INT – interrupt - прерывание), которая выполняется в самой прерываемой программе. Программа фактически сама себя прерывает. При аппаратных прерываниях срабатывают специальные схемы контроля компьютера и сигнал прерывания поступает через специальное устройство (КП – контроллер прерываний) в центральный процессор (CPU) после чего работающая программа прерывается. Кстати это может быть любая программа, прерывания которой разрешены. Аппаратные прерывания могут возникать от любых устройств, которые могут связываться с процессором (клавиатура, мышь, диски и т.п.). Устройства с помощью прерываний сообщают процессору о своем состоянии, выполнении действий и исправности/неисправности.

Прерывания подразделяются также на маскируемые и немаскируемые прерывания. Если прерывание может быть временно запрещено, то оно относится к группе маскируемых прерываний. Программные прерывания подразделяются на группы: прерывания DOS и прерывания BIOS. Такая классификация зависит от места расположения процедур обработки прерываний. Прерывания могут внутренними и внешними, в зависимости от способа инициирования прерывания (в данном компьютере или извне). Внешние прерывания немаскируются.

При инициировании программных прерываний в программе выполняется специальная машинная команда:

**INT <номер прерывания> ,**

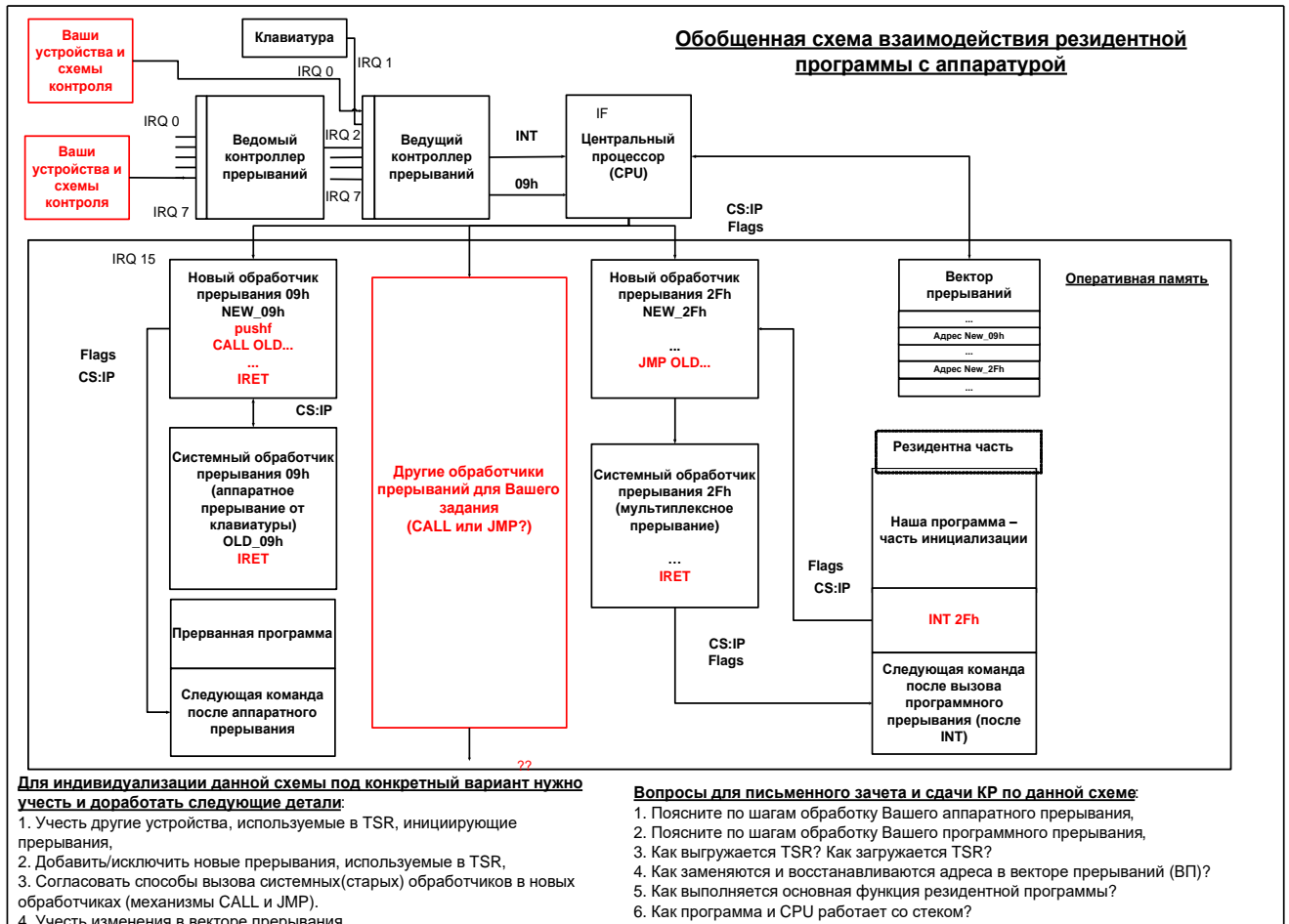
где <номер прерывания> - число в пределах от 0 до 255 (один байт). Данный номер определяет адрес процедуры обработки прерывания в векторе прерываний (см. выше).

Аппаратные прерывания инициируются устройствами, подключенными к компьютеру, для этого они подключены к специальной микросхеме – контроллеру прерываний КП. Эта микросхема в свою очередь подключена к центральному процессору и позволяет по специальным линиям передать: сигнал прерываний и специальный номер прерывания.

В курсовой работе по дисциплине СП Вам необходимо разобраться с механизмом инициации и обработке аппаратных и программных прерываний и написать собственные процедуры обработки прерываний в виде резидентных модулей (часть 1).

## 18.6. 18.6 Обработка прерываний в процессоре (программных и аппаратных)

На рисунке, расположенном ниже представлена обобщенная и укрупненная схема обработки программных и аппаратных прерываний. Данная схема должна быть доработана в рамках курсовой работы по СП для каждого конкретного варианта задания студента.



На данном рисунке дополнительно выделены: перечень возможных изменений для индивидуализации задания по конкретному варианту и перечень основных вопросов задаваемых при защите курсовой работы по дисциплине СП по данной схеме (данная схема должна быть разработана в виде одного из трех листов курсовой работы).

Поясним кратко последовательность действий при обработке аппаратных и программных прерываний.

**Аппаратные прерывания** (в нашем случае обработка прерывания от клавиатуры – номер – 09H):

- Нажата любая клавиша на клавиатуре.
- Иницируется прерывание (IRQ1) и сигнал поступает в контроллер прерывания

**Примечание 1.** Для обработки 16-ти прерываний в цепочку, на нашей схеме, соединены 2 контроллера прерываний (КП), каждый из которых способен обработать только по 8 сигналов прерываний.

- Из контроллера прерываний в центральный процессор (CPU) поступает сигнал INT и номер прерывания 09H.
- По номеру прерывания процессор выбирает адрес обработчика прерывания из вектора прерывания, в нашем случае этот сдвиг в ВП равен 36 (9\*4).



- Процессор запоминает в стеке регистр флагов (Flags) и дальний адрес возврата в прерванную программу (CS:IP).
- На основе адреса обработчика устанавливаются новые CS:IP, что фактически приводит к вызову новой процедуры обработчика прерываний.
- Новый обработчик прерываний выполняет необходимые (по варианту задания) действия.
- В новом обработчике необходимо запомнить в стеке регистры общего назначения, которые используются в его программе.
- При необходимости может быть вызван старый обработчик прерываний по методу CALL или методу JMP. При использовании вызова старого обработчика с возвратом нужно в стек поместить регистр флагов (команда PUSHF) для того, чтобы корректно работала команда IRET в этом обработчике. Вызов с возвратом может быть выполнен в начале нового обработчика, при его окончании или в середине. При вызове с командой JMP выполняется безусловный переход и в стек ничего не заносится. Перед вызовом по методу JMP нужно восстановить все запомненные регистры общего назначения.
- Восстанавливаются регистры общего назначения в новом обработчике.
- Выполняется возврат в прерванную программу командой IRET, которая из стека выбирает регистр флагов и дальний адрес возврата CS:IP.
- Продолжает работать прерванная программа.

**Программные прерывания** (в нашем случае обработка прерывания запускается командой INT 2FH):

- Выполняется команда INT 2FH в части инициализации резидентной программы.
- Процессор, выполняя эту команду, определяет адрес нового обработчика прерывания ( $2F_{16} = 47_{10}$ , смещение  $47 \cdot 4 = 188$ ). Получаем дальний адрес нового обработчика.
- Запоминается в стеке регистр флагов и текущие значения CS:IP в качестве точки возврата в прерванную программу.
- На основе адреса обработчика устанавливаются новые CS:IP, что фактически приводит к вызову новой процедуры обработчика прерываний.
- Новый обработчик прерываний выполняет необходимые (по варианту задания) действия.
- В новом обработчике необходимо запомнить в стеке регистры общего назначения, которые используются в его программе.
- Восстанавливаются регистры общего назначения в новом обработчике.
- При необходимости может быть вызван старый обработчик прерываний по методу CALL или методу JMP. При использовании вызова старого обработчика с возвратом нужно в стек поместить регистр флагов (команда PUSHF) для того, чтобы корректно работала команда IRET в этом обработчике. Вызов с возвратом может быть выполнен в начале нового обработчика, при его окончании или в середине. При вызове с командой JMP выполняется безусловный переход и в стек ничего не заносится. Перед вызовом по методу JMP нужно восстановить все запомненные регистры общего назначения.
- Выполняется возврат в прерванную программу из старого обработчика прерывания командой IRET, которая из стека выбирает регистр флагов и дальний адрес возврата CS:IP.
- Продолжает работать прерванная программа в части инициализации.

В курсовой работе студенты должны приспособить данную схему для своего варианта и описать последовательность обработки прерываний в документации (Техническое описание).

### 18.7. 18.7 Установка резидента

В конце части инициализации резидентной программы необходимо указать операционной системе, что часть программы останется резидентной (Часть 1), то есть выполнить установку резидента.

Для установки резидента используется специальное прерывание DOS (Terminate and Stay Resident - TSR):

**031H – 21H.**

Или BIOS прерывание

**027H**

Эти варианты отличаются способом задания размера памяти резидента ("обрезания") и отдельными свойствами – см. более подробно в справочниках DOS:

**1- способ( ОП >64kb и старшие версии ДОС):**

```
MOV DX , <размер резидента в параграфах>
MOV AX, 3100H
INT 21H
```

**2-способ:**

```
MOV DX , <адрес конца резидента от PSP>
INT 27H
```

Для правильной установки необходимо рассчитать в программе рассчитать его размер.

### 18.8. 18.8 Расчет размера резидента

Для случая с прерыванием 031H – 21H нужно определить в параграфах размер оставляемой памяти. Он определяется для программы:

```
ORG 100h
BEGIN: ...
```

...

```
INIT:
```

Так:

```
MOV DX, (INIT - BEGIN + 10fh)/16
```

Размер вычисляется как разница начала части инициализации (INIT), которая будет удалена (Часть 2) и адреса точки входа в программу (BEGIN). Кроме того, добавляется размер области PSP и корректировочная константа. Полученное значение нужно разделить на 16 для получения размера в параграфах (16 байт). Причем:

- 100h – размер PSP, а

0Fh – коррекция при делении нацело (16) для округления целого числа параграфов.

Для прерывания 027H указывается адрес конца резидента как адрес начала части инициализации (INIT):

```
MOV DX, OFFSET INIT
INT 027h
```

### 18.9. 18.9 Запуск части инициализации

Резидентная программа обычно оформляется в формате \*.COM. Поэтому точка входа должна быть расположена непосредственно после области выделяемой под PSP про-

цесса (ORG 100H). Поэтому точка входа должна быть помечена в первой строке после предполагаемой области PSP(метка **BEGIN**). В директиве конца программы **END** эта метка также должна быть задана. После запуска программы из командной строки выполняется безусловный переход на метку инициализации (**JMP INIT**). Начало резидента может быть оформлено и в виде процедуры (**BEGIN PROC ... BEGIN ENDP**), но это не имеет особого значения, так как в эту процедуру возврат не осуществляется.

```
; Начало резидента
CODEPR SEGMENT PARA
ASSUME CS:CODEPR , DS:CODEPR
; Начало части резидента (Часть 1)
    ORG 100H ; установка счетчика команд (байт) на 100H для последующей
            ;вставки PSP
BEGIN:
    JMP INIT      ; это метка начала части инициализации
; Определение данных резидента
...
; Начало части инициализации (Часть 2)
INIT:
...
END BEGIN      ; Определение точки входа при запуске в командной строке
```

Область данных резидента обычно располагается после этой метки, хотя эти данные можно расположить в любом месте резидентной части (Часть 1 – располагается между метками **BEGIN** и **INIT**).

## 18.10. 18.10 Определение и запоминание старого обработчика

Для каждого прерывания, которое будет обрабатываться в резиденте, нужно определить и запомнить адрес старого обработчика прерывания. Это делается для того, чтобы после завершения работы резидентной программы можно было восстановить старый обработчик и, при необходимости, вызвать его из собственного обработчика для выполнения стандартных действий (например, при обработке прерывания от клавиатуры). Эти действия могут быть выполнены либо с помощью специально прерывания (35H – 21H) либо с помощью непосредственного обращения в область вектора прерывания (см. ниже).

Адрес обработчика считывается в поле длиной в 4 байта (длинный адрес). Нужно помнить, что двойные слова сохраняются в памяти в обратном порядке (сначала младшее слово – смещение, а затем старшее слово – сегмент). После обращения к прерыванию 035H ,в регистрах мы получим дальний адрес из вектора прерывания (ES:BX). При этом на AL указывается номер прерывания. Для приведения адресного выражения к слову используется спецификация **WORD PTR**.

```
; область данных части резидента
OLD_09 dd ?          ; адрес старого обработчика 09h
...
; часть инициализации
...
MOV AH, 35h
MOV AL , 09h  ; номер обработчика
INT 21h
MOV WORD PTR CS:OLD_09, BX
MOV WORD PTR CS:OLD_09+2, ES
```

Такие действия необходимо выполнить в программе для каждого обработчика прерывания, задаваемого в программе.

## 18.11. 18.11 Задание нового обработчика прерывания

Для каждого прерывания, которое обслуживает резидентная программа нужно определить новый обработчик прерывания. Это делается с помощью функции **025h-21h** или с

помощью непосредственной записи в область вектора прерывания (см. ниже). В приведенном ниже фрагменте предполагается, что в регистре CS записан адрес сегмента кода программы. Новый обработчик располагается в резидентной части программы. На регистре AL должен быть задан номер нового обработчика прерывания, а на регистре DX смещение обработчика - **NEW\_09** (для CS). Ниже приведены фрагменты для части резидентной и части инициализации для установки нового прерывания.

```

; часть резидента
NEW_09 PROC
...
IRET
NEW_09 ENDP
...
; часть инициализации
MOV AH, 25h
MOV AL, 09h ; номер прерывания обработчика
LEA DX, NEW_09
INT 21h

```

## 18.12.18.12 Вызов старого обработчика прерывания

Вызов старого обработчика прерывания может быть выполнен как с помощью безусловного перехода (**JMP**), так и как перехода с возвратом (**CALL**). И в первом и во втором случае адресом вызова должен быть запомненный в резидентной части адрес старого обработчика (например: **OLD\_09**).

При безусловном переходе вызов выполняется так:

```

; тело нового обработчика прерывания
...
; вызов старого обработчика прерывания без возврата
JMP DWORD PTR CS:OLD_09

```

При переходе с возвратом:

```

; тело нового обработчика прерывания
...
; вызов старого обработчика прерывания с возвратом
PUSHF ; !!!!
CALL DWORD PTR CS:OLD_09
; продолжение тела нового обработчика прерывания
...
IRET

```

Различие в этих вызовах заключается, в том числе, в наличии команды **PUSHF**, записывающей в стек регистр флагов. Это необходимо для того, чтобы при выполнении команды **IRET** стек корректно освобождался. Кроме этого, при вызове без возврата нет необходимости задавать команду **IRET**, так как завершение работы резидента и возврат к прерванной программе выполняется в старом обработчике.

## 18.13.18.13 Пример простейшего резидента

На основе материала изложенного выше можно построить простейшую резидентную программу. В этой программе не будут некоторых важных возможностей (проверки повторности, выгрузки резидент, обработки параметров и т.д.), но для простоты понимания задачи пока их опустим.

**Задача:** Написать резидентную программу, которая при нажатии клавиши **ESC** выводит на экран цепочку одинаковых символов (в программе ‘**B**’). При загрузке резидента должно выдаваться сообщение “**Start TSR**”.

Текст программы такого резидента приведен ниже.

```

; Символы по нажатию ESC

```

```

CODEPR SEGMENT PARA
    ASSUME CS:CODEPR , DS:CODEPR
    ORG 100H ; Область PSP
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Резидентная часть
BEGIN:  JMP  INIT ; Переход к части инициализации
; Данные резидента
SAVEINT9 DD ? ; Сохранение старого обработчика
; Новый обработчик прерывания 09H
NEWINT9: PUSH AX ; Сохранение используемых регистров
        PUSH CX
        PUSH BX
; Запрос сканкода из клавиатуры
        IN AL, 60H ; Взять из порта 60 H на регистр AL
        CMP AL, 1 ; Проверка сканкода ESC он равен 1
        JNE EXIT ; Переход если не ESC
; Вывод на экран 10-ти символов 'B' с помощью BIOS функции 0AH - 010H
        PUSH AX
        PUSH BX
        PUSH CX
        MOV AH , 0AH
        MOV AL, 'B'
        MOV BH , 0
        MOV CX, 10 ; Число символов
        INT 010H
        POP CX
        POP BX
        POP AX
; Восстановление регистров и вызов старого обработчика без возврата
EXIT:   POP BX
        POP CX
        POP AX
        JMP CS:SAVEINT9 ; Вызов старого обработчика
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Часть инициализации
INIT:   CLI ; Запрет прерываний
; Установка DS
        PUSH CS
        POP DS
; Получение адреса старого обработчика
        MOV AH, 35H
        MOV AL, 09H ; Номер прерывания
        INT 21H
; Сохранение адреса старого обработчика
        MOV WORD PTR SAVEINT9 , BX
        MOV WORD PTR SAVEINT9+2 , ES
; Установка нового обработчика в вектор прерывания
        MOV AH,25H
        MOV AL, 09H ; Номер прерывания
        MOV DX, OFFSET NEWINT9
        INT 21H
; Вывод сообщения о загрузке резидента
        MOV AH , 09H
        MOV DX, OFFSET MSG
        INT 21H
; Завершить и оставить резидентной (TSR)
        MOV AX, 3100H
        MOV DX, (init - begin +10FH)/16 ; Размер резидента
        STI ; Разрешение прерываний
        INT 21H
; Данные части инициализации
MSG DB 'Start TSR', 10,13,'$'

```

```
CODEPR    ENDS
          END      BEGIN
```

В данной программе вывод сообщения о старте резидента выводится с помощью функции DOS (09H – 21H), а вывод цепочки символов с помощью прерывания BIOS (0AH – 10H), так как использование функций DOS в резидентной части ограничено и практически недопустимо (!!!).

Компиляция такой программы (fast.asm) выполняется командным файлом, содержащем следующие директивы:

```
tasm /l /zi fast.asm
tlink /v /t /l fast.obj
PAUSE
```

Директива PAUSE необходима для контроля наличия ошибок в программе. После запуска резидента при нажатии клавиши **ESC** на экране будет выведена следующая строка:

```
C:\work>fast.com
Start TSR
...
```

После нажатия клавиши ESC на экран в командную строку будет выведено:

```
C:\work>\BBBBBBBBBB
```

Должно было быть выведено 10 символов, а на самом деле всего 9. Это объясняется тем, что при нажатии ESC на экран выводится символ '\', а так как в BIOS прерывании с повторением курсор не перемещается, то символ '\' заменяет первую букву 'B'.

Чтобы исправить эту “ошибку” необходимо выполнить полностью обработку прерывания, включая выдачу сигналов клавиатуре о прочитанном символе, а контроллеру прерывания сигнала о завершении обработки прерываний. Для обработки клавиатурных прерываний это существенно. В следующем фрагменте показано, как это сделать.

```
; Сохранение регистров перед выводом на экран цепочки
    PUSH AX
    PUSH BX
    PUSH CX
; Выдача сигнала через порт контроллера 8255 о чтении сканкода
    IN  AL, 61H
    OR  AL, 10000000b      ; установим бит 7 порта B
    OUT 61H, AL
    AND AL, 01111111b
    OUT 61H, AL           ; сбросим бит 7 порта B (символ прочитан)
; Вывод цепочки 10 символов
    MOV AH, 0AH
    MOV AL, 'B'           ; символ 'B'
    MOV BH, 0
    MOV CX, 10            ; Число символов
    INT 010H              ; прерывание BIOS DOS dghzve. ytkmpz
    POP CX                ; восстановление регистров
    POP BX
    POP AX
; Сигнал контроллеру прерываний через порт 20H сигнал (EOI = 20H)
    MOV AL, 20H
    OUT 20H, AL           ; в порт ведущего контроллера 8259A
    POP AX
    IRET                  ; завершение нового обработчика при вводе ESC
```

В результате мы получим строку, которая расположена ниже, причем курсор будет располагаться в первой позиции, так как 0A-010H не переводит курсора.:

```
C:\work>\BBBBBBBBBB
```

Теперь продолжим рассмотрение других особенностей построения и реализации резидентных программ.

### 18.14. 18.14 Работа с вектором прерываний напрямую

Работать с вектором прерываний можно напрямую. В реальном режиме доступна вся память, достаточно задать длинный адрес (сегмент смещение). Адрес расположения зависит от номера прерывания, достаточно этот номер умножить на 4 и получим необходимое смещение. Значение сегментного регистра (ES) в этом случае должно быть нулевым. Необходимо также учесть, что в памяти двойное слово по словам располагается в обратном порядке, с начала смещение, а затем сегмент. Текст этих фрагментов программы приведен ниже:

Запоминание старого обработчика:

```
MOV AX, 0
PUSH AX
POP ES                ; задание в ES значения 0
MOV BX, WORD PTR ES: 09h*4    ; считаем первое слово
MOV WORD PTR CS:OLD_09, BX    ; запомним его в поле старого обработчика
MOV BX, WORD PTR ES: 09h*4 + 2 ; считаем второе слово
MOV WORD PTR CS:OLD_09+2, BX  ; запомним его в поле старого обработчика
```

И запись нового обработчика:

```
MOV AX, 0
PUSH AX
POP ES
LEA BX, NEW_09
MOV WORD PTR ES: 09h*4, BX
MOV BX, CS
MOV WORD PTR ES: 09h*4 + 2, BX
```

Несмотря на рассмотренную возможность, более корректно использовать для этих целей функции 035H и 025H прерывания 21H. Подумайте почему.

### 18.15. 18.15 Обработка в одном резиденте нескольких прерываний

Если в резиденте используется несколько обработчиков, то в отдельных полях нужно запомнить адрес каждого, и для каждого нужно определить процедуру (адрес) его обработки, а резидентной части создать процедуру для его обработки.

<Метка 1:>Обработчик 1
<Метка 2:>Обработчик 2
<Метка 3:>Обработчик 3
...

← вызов 1 из вектора прерываний  
 ← вызов 2 из вектора прерываний  
 ← вызов 5 из вектора прерываний

Таким образом, для каждого нового обработчика нужно: зарезервировать процедуру обработки этого прерывания, обозначив вход в нее меткой (или именем этой процедуры), сохранить в специальном поле дальний адрес старого обработчика, запомнить в векторе прерывания адрес нового обработчика. В программе это будет выглядеть так:

```
; резидентная часть
...
OLD_9H DD ?
OLD_2FH DD ?
OLD_OAH DD ?
...
; Обработчики прерываний
NEW_9H: ...
...
NEW_2FH: ...
...
OLD_OAH: ...
...
```

```

; часть инициализации
...
; сохранение старых обработчиков
...
; установка новых обработчиков
...

```

Сохранение и запоминание старых обработчиков рассмотрена выше, она выполняется для каждого обработчика отдельно. Установка новых обработчиков производится аналогично.

### 18.16. 18.16 Связь с резидентом из других программ

Связь с резидентной программой может быть выполнена разными способами: посредством специальных или общих прерываний и непосредственным вызовом процедур резидента с предварительным определением адреса из вектора прерываний. Рассмотрим способ использования специальных прерываний.

Специальные прерывания 2F и 2D, называемые мультиплексными могут использоваться для связи с резидентными процедурами (в ряде случаев, если возникают необъяснимые ошибки целесообразно использовать прерывание 2D или любое другое свободное прерывание в ВП). Главная проблема в этом случае – точно установить, что Вы обращаетесь именно к своей резидентной программе. Для этого задаются правила взаимодействия.

#### Мультиплексное прерывание 2F:

1. Первоначально необходимо сохранить старый вектор и установить собственный обработчик в резидентном модуле (см. выше).

2. Далее в резиденте описывается процедура типа:

```

NEW_2F      PROC FAR
             CMP AH, 0EEh      ; собственный код
             JE  MY_OBR
             JMP OLD_OBR
;
MY_OBR:      CMP AL, 1          ; код наличия в памяти
             JE  M1
;   Проверка выгрузки
             CMP AL, 2          ; код выгрузки
             JNE M2
; команды и данные для выгрузки
...
             MOV AL, 0FFh      ; сигнал об успешной выгрузке
             IRET
;   Установка признака наличия в памяти
M1:          MOV AL, 0FFh      ; по договоренности код информирования о наличии
M2:          IRET
;   Вызов старого обработчика
OLD_OBR:     JMP DWORD PTR CS:OLD_2F
..

```

При вызове обработчика **NEW\_2F** первоначально проверяется уникальный код собственного обработчика, который мы задаем сами (у нас это ‘0EEH’). Если код не соответствует то выполняется переход на вызов старого обработчика прерывания (метка - OLD\_OBR), в противном случае выполняется переход в тело процедуры нашего обработчика (метка – MY\_OBR). В обработчике мы проверяем код заданный на регистре AL, 1-режим проверки повторности, 2 – режим выгрузки резидента. В режиме выгрузки резидента выполняются действия по восстановлению вектора прерывания на основе запомненных ад-



ресов старых обработчиков (см. ниже) и возвращается признак правильной выгрузки на регистре AL (0FFH). Далее выполняется команда возврата из обработчика - IRET. В режиме проверки наличия нашего резидента в памяти также возвращается признак правильной выгрузки на регистре AL (0FFH) и выполняется возврат (IRET).

3. В основной программе для различных действий вызывается свой обработчик в части инициализации или отдельной программе может быть вызван новый обработчик. Это выполняется так:

Для проверки наличия:

```
MOV AH, 0EEh
MOV AL, 1    ; Проверка наличия
INT 2Fh
CMP AL, 0FFh
JE IMET1    ; переход если резидент в памяти
; код если нет в памяти
...
```

Для передачи сигнала выгрузки, если при завершении резидент должен выгружаться сам, возможен следующий код:

```
MOV AH, 0EEh
MOV AL, 2    ; Сигнал выгрузки
INT 2Fh
CMP AL, 0FFh
JE IMET1    ; переход если выгружено нормально
; код если не выгружен
...
; код после выгрузки
IMET:
```

Можно выполнять и другие действия для передачи информации резиденту из других программ.

## 18.17. 18.17 Выгрузка резидента

При выгрузке резидентной программы из оперативной памяти должны быть выполнены следующие действия:

- Выполняются необходимые действия для завершения резидента (заккрытие файлов, сброс системных флагов и другие необходимые действия).
- Восстановлены все старые обработчики в векторе прерывания.
- Освобождена оперативная память, выделенная под резидентную программу и под сопровождение процесса (окружение программы).

Отдельные действия конкретного резидента зависят от существа задачи, поэтому здесь мы не будем рассматривать. Действия по восстановлению векторов прерываний аналогичны действиям по установке резидента и базируются на сохраненных адресах старых обработчиков. Они выполняются так:

```
PUSH DS
; Восстановление 9H
mov AX, 2509h    ; Восстановление обработчика прерывания 05H
lds DX, CS:SAVEINT9
int 21h
; Восстановление 2FH
mov AX, 252Fh    ; Восстановление обработчика прерывания 2FH
lds DX, CS:SAVEINT2F
int 21h
POP DS
```

Команда **LDS** позволяет загрузить одновременно на основе длинного адреса регистры DS и DX, необходимые для выполнения функции 25H. Так как регистр DS затирается его нужно сохранить в стеке и восстановить.

Действия по освобождению памяти выполняются с помощью функции 49H прерывания DOS. При этом на регистре ES должен быть установлен сегментный адрес начала выделенной области (или блока – ОС выделяет память блоками, которые идентифицируются по началу блока). В начале резидента располагается область PSP (Program Segment Prefix см. в справочниках), которая содержит сегментный адрес области окружения DOS со смещением 2CH (DOS environment). Выгрузка самой программы выполняется на основе сегментного регистра кода, который сформирован при вызове самого резидента. Код ниже иллюстрирует эти операции.

; получим из PSP адрес собственного окружения и выгрузим окружение DOS

```
MOV ES,CS:2CH
MOV AH,49H
INT 21H
; выгрузим теперь саму программу
PUSH CS
POP ES
MOV AH,49H
INT 21H
IRET
```

После действий по освобождению памяти никакие действия в обработчике нельзя считать корректными.

### 18.18. 18.18 Разбор параметров командной строки

В курсовой работе по СП есть требование использования при инициализации параметров командной строки. Разбор параметров из области PSP вы освоили в 6-й ЛР. Здесь приведу основные сведения:

1. Размер командной строки задан в программе со смещением 80H (область PSP). Это однобайтовое поле. Если его значение равно 0, то параметров в командной строке нет. Максимальный размер этого поля параметров равен 127 байт.
2. Начиная с CS:81H расположена сама строка параметров, которые нужно проверять, включая, кстати, и сброс пробелов. Просмотр строки лучше выполнять в цикле. В алгоритмах рекомендую использовать команды LOASB и STOSB.
3. Алгоритм просмотра реализуется в цикле и с учетом строгой и нестрогой последовательности параметров в задании, а так же разных регистров.
4. Для доступа к параметрам окружения DOS (SET переменные, подробнее в справочнике) необходимо использовать дальний адрес, расположенный в PSP со смещением 2CH (точнее CS:2CH).

Ниже приводится фрагмент программы для анализа наличия параметров в командной строке.

```
MOV AL, CS:80H
CMP AL, 0 ; проверить число байт параметров командной строки
JNE INIT_UND ; перейти на метку если параметров нет
```

### 18.19. 18.19 Контроль наличия резидента (другой способ)

Проверка наличия может быть выполнена и другим способом: по адресу резидента проверяем заданную константу - пароль. Адрес поля пароля можно вычислить на основе адреса обработчика прерывания (например, INT\_09 - 2)

; это часть резидента

```
...
PASSW DW 62627
INT_09: ...
```

Далее в программе инициализации вычисляем адрес резидента с помощью функции DOS 35H. Адрес получаем на регистрах - ES:BX.

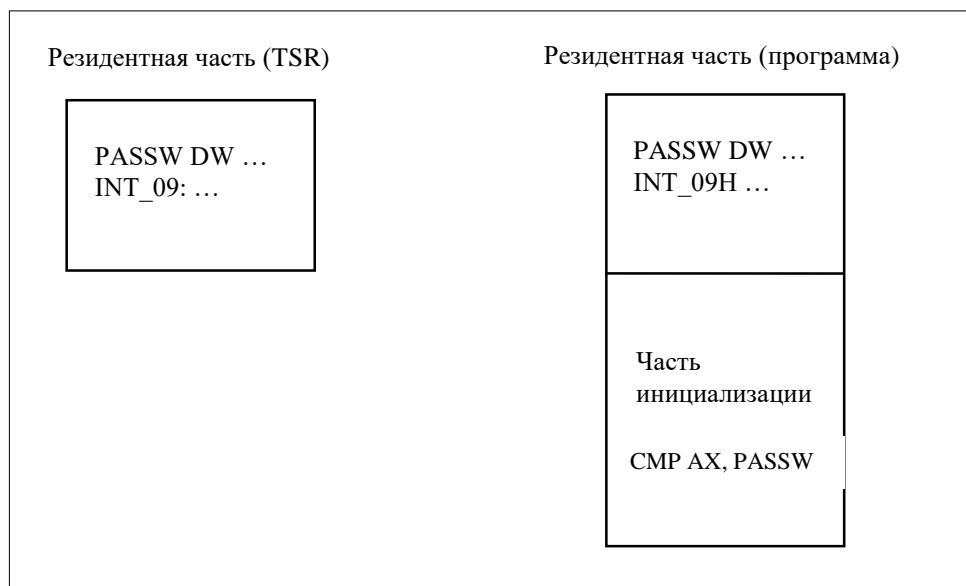
**; это часть инициализации**

```
...
MOV AH, 35h
MOV AL, 09h ; номер обработчика
INT 21h
SUB BX, 2
MOV AX, WORD PTR ES:BX
CMP AX, PASSW
JNE NOLOAD ; метка обработки отсутствия в памяти
...
```

Если код записанный в поле PASSW и код полученный из резидента по этому прерыванию совпадают, то с большой долей вероятности можно считать, что резидент уже в памяти. Если совпадения нет, то резидент отсутствует.

Такая проверка возможна потому, что в оперативной памяти может одновременно находиться две резидентные части: одна – загруженный резидент, а вторая часть основной программы.

Рисунок расположенный ниже иллюстрирует ситуацию, когда в оперативной памяти присутствуют две резидентные части программы.



## 18.20. 18.20 Связь с резидентом с помощью клавиатуры

В каждом варианте курсовой работы по СП предусмотрена обработка прерывания от клавиатуры. Прерывание от клавиатуры является аппаратным (BIOS прерывание – 09H), для его обработки необходимо установить собственный обработчик. Обработка этого прерывания может быть выполнена:

- С использованием портов ввода/вывода (60H , 61H и 20H), или
- С использованием стандартного обработчика прерываний ,или
- С помощью комбинированного использования портов и стандартного обработчика прерываний.

При использовании стандартного обработчика прерываний можно использовать специальный буфер клавиатуры и специальные функции прерывания BIOS для работы с клавиатурой (16H).

Для инициации обработки клавиатуры необходимо, основываясь на изложенной выше информации, установить собственный обработчик для прерывания 09H. При работе с буфером клавиатуры можно использовать область системной памяти, которая расположена в начале ОП с сегментным адресом 040H (смещение - 0400H) сразу после вектора прерываний. Буфер клавиатуры является циклическим, каждый его элемент содержит 2 байта. Таким образом максимально в нем может быть расположено 16 символов ( точнее нажатий клавиш, включая и их комбинации). Такое необходимо, так как обработчики прерывания могут не успеть обработать все нажатия. При переполнении циклического буфера выдается сигнал через динамик и новые нажатия клавиш в нем не фиксируются. Для работы с буфером клавиатуры предусмотрено два указателя на начало (голову списка -Head) и на конец (хвост списка - Tail). Если эти указатели равны, то список пуст.

Кроме того, в системной области фиксируется информация о нажатии управляющих клавиш (таких как Alt, Ctrl и др.) Расположение специальных полей в системной области (0400H), позволяющих работать с буфером клавиатуры (более подробно содержание системной области смотрите в справочниках – General Memory Map) показано ниже:

По адресу **0417h** – находится информация (размером в 2 байта) о нажатии специальных управляющих клавиш (Ctrl, Alt, Shift и др.), подробнее о содержании смотрите в справочниках.

По адресу **041Ah** – находится адрес головы циклического буфера (2 байта)

По адресу **041Ch** – находится адрес головы циклического буфера (2 байта)

По адресу **041Eh** (32 - байта) – циклический буфер клавиатуры

На основе этой информации можно установить код символа, стоящего в очередь на обработку, и информацию о нажатии управляющих клавиш. Использование буфера клавиатуры возможно после вызова стандартного обработчика прерывания от клавиатуры.

**Работа через порт.** При использовании порта ввода/вывода можно прочитать код нажатой клавиши в порта А контроллера 8255 (60H). Это может быть выполнено в программе собственного обработчика с помощью следующей команды:

**IN AL , 060H ; чтение скан-кода клавиши**

Полученный скан-код (см. раздел - 23.5. SCAN – коды), можно использовать в программе. Для получения признака нажатия управляющих клавиш (Shift, Ins и др. см. в справочнике) можно использовать специальное прерывание BIOS. В примере, расположенном ниже проверяется нажатие клавиши Shift совместно с клавишей ‘Q’.

```
IN    AL, 60H          ; Взять из порта 60H сканкод на регистр AL
CMP   AL, 10H          ; Проверка нажатия 'Q', скан код не зависит от регистра
JNE   OLD              ; Переход на стандартный обработчик
; проверка клавиши Shift (первый бит в байте состояния управляющих клавиш)
MOV   AH, 02H
INT   016H             ; Получим управляющий байт на AL
TEST  AL, 00000010b     ; Проверим на 1
JZ    OLD              ; Переход на стандартный обработчик, если 0
```

Для завершения самостоятельной обработки необходимо через порт В контроллера 8255 (61H) передать информацию о считывании скан-кода. Для этого нужно выполнить следующие команды:

```
IN    AL, 61H          ; Считаем старое состояние
OR    AL, 10000000b     ; Установим 7-й бит в 1
OUT   61H, AL          ; Запишем в порт
AND   AL, 01111111b     ; Сбросим 7-й бит
OUT   61H, AL          ; Запишем в порт
```

Данные команды позволяют подтвердить прием скан-кода из клавиатуры. Если далее нам не нужно обрабатывать стандартным обработчиком данное прерывание, то необходимо передать сигнал (20H – EOI END OF INTERRUPT) в порт контроллер прерываний. Это делается командами:

```
MOV AL, 20H
OUT 20H, AL
POP AX
IRET
```

Если нужно завершить обработку с системным обработчиком прерываний, то нужно ему передать управление:

```
OLD: POP AX
STI
JMP CS:SAVEINT9 ; Вызов старого обработчика
```

**Работа с буфером клавиатуры.** При работе с буфером клавиатуры целесообразно вызвать старый обработчик предварительно. Это можно сделать так:

```
NEWINT9:
PUSH AX ; Сохранение используемых регистров
PUSH CS ; для работы с данными
POP DS
PUSHF ; Это обязательно для корректного возврата
CALL CS:SAVEINT9 ; Вызов старого обработчика
```

Для полноты работы с клавиатурой нужно иметь возможность получить информацию об управляющих клавишах и значение введенного кода( скан-кода, ASCII кода и расширенного ASCII кода). Для проверки нажатия управляющих клавиш можно прочесть специальный байт по адресу 0417 из системной области. Это можно сделать так:

```
; проверка левого шифта
MOV AX , 40H ; сегментный адрес системной области
MOV ES , AX
MOV AL , ES:17H ; смещение для управляющего байта состояния клавиш
TEST AL , 00000010b ; проверка второго бита
JZ PROD ; переход если левый шифт не нажат
```

Такие действия можно выполнить и по-другому, основываясь на прерывании BIOS 02H – 16H:

```
MOV AH , 02H
INT 16H ; BIOS прерывание для получения управляющего байта
TEST AL , 00000010b ; проверка второго бита
JZ PROD ; переход если левый шифт не нажат
```

Если символ прочитанный с клавиатуры не должен появляться на экране (например, в окне COMMAND.COM) необходимо обнулить буфер клавиатуры. Это можно сделать, установив одинаковые значения указателей циклического буфера. Например, так:

```
MOV AX , 40H
MOV ES , AX
MOV BX , ES:1AH
MOV ES:1CH , BX
```

После такой операции введенный символ не будет выводиться на экран командного процессора. Это можно сделать и так:

```
MOV AH , 00H ; код команды чтения символа и выборки из буфера
INT 16H ; BIOS прерывание для получения управляющего байта
```

Если буфер клавиатуры пуст, то программа будет ожидать нажатия клавиши. Из буфера клавиатуры можно прочесть коды нажатой клавиши или их комбинации напрямую. Это может быть сделано так:

```
MOV AX , 40H
MOV ES , AX
MOV BX , ES:1AH
MOV AX , ES:[BX]
```

После такой операции на регистрах АН и АL заносится комбинация кодов, на основе которых можно выделить следующее: ASCII символ, скан-код или расширенный ASCII код. Это можно сделать и так:

```
MOV AH , 01H ; код команды чтения символа и без выборки из буфера
INT 16H ; BIOS прерывание для получения управляющего байта
```

Если регистр АL содержит 0, то это значит, что в регистр АН занесен расширенный ASCII код, который можно анализировать. В противном случае регистр АL содержит ASCII введенного символа, а регистр АН содержит скан-код нажатой клавиши.

Предварительно необходимо вызвать системный обработчик прерываний. Анализ введенного кода для проверки нажатой клавиши может быть проведен так:

```
MOV AX , 40H
MOV ES , AX
MOV BX , ES:1AH
MOV AX, ES:[BX] ; на AX 2 байта из буфера клавиатуры
; Проверка расширенного кода
CMP AL , 0 ; проверка отсутствия ASCII
JE PRINT_EXT ; переход, если расширенный код
CMP AL , 61H ; Проверка символа 'a'
JNE EXIT
CMP AH , 1EH ; Проверка нужного скан-кода для клавиши 'A'
JNE EXIT
...
PRINT:
...

; Проверка комбинации расширенного кода
PRINT_EXT:
CMP AH , 5EH ; Ctrl + F1
JNE EXIT ; Переход если не наша комбинация
...
JMP PRINT
```

В данном примере сначала на АХ из буфера клавиатуры получаем два байта. Затем проверяем АL на 0. Нулевое значение свидетельствует о наличии расширенного кода ASCII и мы переходим на метку для его анализа (PRINT\_EXT). Там мы проверяем код комбинации Ctrl + F1( код - 5E). Если код ASCII (AL != 0), то проверяем нужный код в регистре АL (у нас 61H – код строчной буквы ‘a’). Для иллюстрации наличия скан-кода клавиши ‘A’ делаем дополнительную проверку регистра АН, куда помещен скан-код этой клавиши – 1EH. Если код не соответствует, то уходим на выход из процедуры.

Практике программирования и в курсовой работе по СП Вы можете выбрать любой из вариантов для обработки прерываний от клавиатуры.

## 18.21.18.21 Освобождение памяти внешнее из отдельной программы

Освобождение памяти под резидент и восстановление векторов прерываний старых обработчиков прерываний. Вы можете выполнить и вне резидента, в отдельной программе. Для этого нужно вычислить адрес начала резидентной программы и PSP, а затем выполнить действия по восстановлению старых векторов прерываний и освобождению оперативной памяти. Трудности может составить получение адресов сохраненных старых адресов вектора прерываний. Если разместить переменную для сохранения старого адреса обработчика непосредственно перед точкой входа в процедуру обработки прерываний, то получить этот адрес несложно, вычитая из этого адреса четыре.

```
SAVEINT9 DD ? ; Сохранение старого обработчика
; Новый обработчик прерывания 09H
NEWINT9:
PUSH AX ; Сохранение используемых регистров
```

Начальная точка расположения резидентной программы и адрес ее PSP могут быть получены аналогично.

## 18.22. 18.22 Завершение основной программы при проверке повторной загрузки

При проверке повторности загрузки в части инициализации, программа завершается обычным образом, если она не остается резидентной. Аналогичное завершение должно быть при выдаче справки о программе. Завершение основной программы в этих случаях может быть выполнена так:

**Либо:**

```
MOV AX , 4C00H
INT 21H
```

На AL задается код возврата программы (ERRORLEVEL).

**Либо:**

```
MOV AX , 0000H
INT 20H
```

## 18.23. 18.23 Проверка загрузки и выгрузки с помощью утилиты mem.exe

Для контроля наличия резидента в ОП нужно использовать утилиту ОС MEM.EXE. Эта утилита используется несколько раз: с ее помощью уточняется объем свободной памяти до запуска резидента; проверяется наличие резидента в ОП и проверяется освобождение ОП под модули резидента после его выгрузки. Для этого необходимо сначала уточнить параметры ее запуска, так как для Разных ОС они могут отличаться. Поэтому ее нужно запустить в режиме справки ( MEM.EXE /?). Размер свободной памяти до загрузки резидента должен полностью соответствовать размеру свободной памяти после его выгрузки. Запуск в режиме справки для XP:

```
C:\work>mem /?
```

Справка на экране:

Вывод сведений об используемой и свободной памяти.

MEM [/PROGRAM | /DEBUG | /CLASSIFY]

```

/PROGRAM or /P   Вывод статуса программ, загруженных в память.
/DEBUG or /D     Вывод статуса программ, внутренних драйверов и другой
                  дополнительной информации.
/CLASSIFY or /C  Классификация программ по использованию памяти.
                  Вывод сведений о размерах программ, использовании памяти
                  и максимальном свободном блоке памяти.
```

Если предварительно загрузить резидент (TSRKBD.COM) и проверить распределение памяти командой:

```
C:\work>mem /p
```

Мы получим на экране следующую информацию:

Адрес	Имя	Размер	Тип
-----	-----	-----	-----
000000		000400	Вектор прерывания
000400		000100	Область обмена ПЗУ (ROM)
000500		000200	Область обмена DOS
000700	IO	000370	Системные данные
000A70	MSDOS	0017A0	Системные данные
002210	IO	002280	Системные данные
...			
0044A0	COMMAND	000B50	Программа



005000	MSDOS	000070	- Свободно -
005080	COMMAND	0005E0	Окружение
005670	DOSX	0087A0	Программа
00DE20	COMMAND	000510	Данные
00E340	COMMAND	000B50	Программа
00EEA0	COMMAND	0004F0	Окружение
00F3A0	TSRKBD	000510	Окружение
00F8C0	TSRKBD	000160	Программа
00FA30	MEM	000510	Окружение
00FF50	MEM	0174E0	Программа
027440	MSDOS	078BA0	- Свободно -
...			
0D4310	MSDOS	00BCE0	- Свободно -
655360 байт - всего обычной памяти			
655360 байт - доступно для MS-DOS			
589968 максимальный размер исполняемой программы			
1048576 байт - всего непрерывной дополнительной памяти			
0 байт - доступно непрерывной дополнительной памяти			
941056 байт - доступной памяти XMS			
резидентная часть MS-DOS загружена в сегмент HMA			

На этом примере видно, что под область резидентной программы выделяется два фрагмента ОП, которые необходимо освобождать (программа и окружение). Количество свободной памяти в этом режиме определяется строкой: “**589968 максимальный размер исполняемой программы**”.

## 18.24. 18.24 Описание данных и процедур резидента

Описание данных и процедур, используемых в резидентной части, нужно размещать только в области самого резидента.

```

BEGIN: ...
    JMP INIT
; Описания данных
...
; обработчики прерываний и процедуры
...
; Описания процедур резидента
MY_PROC PROC
...
MY_PROC ENDP
...
; Описания данных
...
MSG DB ...
INIT: ...

```

Если данные для резидента случайно описать в области части инициализации, они не будут доступны и программа резидента не сможет работать правильно.

## 18.25. 18.25 Русификация сообщений резидента

Для того чтобы резидент мог выдавать сообщения на русском языке необходимо выполнить следующие условия:

- Загрузить перед запуском резидента русификатор (например, RKM – см. выше) и



- Сообщения резидента закодировать в коде ASCII, для чего их необходимо ввести в текстовом редакторе DOC при включенном русификаторе. В этом случае в текстовом редакторе под WINDOWS данные сообщения правильно читаться не будут.

Кстати, используя русификатор, Вы дополнительно проверяете правильность своей программы в части вызова старого обработчика прерывания. Необходимо корректно вызывать старый обработчик. В тех случаях, когда используется полная обработка прерывания собственным резидентом, ввод русских символов с клавиатуры становится недоступным.

### 18.26. 18.26 ДЗ и ТЗ на КР

В 4-м семестре студенты в виде домашнего задания (ДЗ) разрабатывают и защищают ТЗ на резидентную программу курсовой работы по СП, которая выполняется по плану 5-го семестра. Требования к выполнению ДЗ, варианты КР и требования к документации размещены на сайте и периодически обновляются. Студенты должны изучить эти требования, без ошибок выбрать свой вариант и сдать ДЗ в указанные сроки. Нумерация вариантов КР в 5-м семестре сохраняется из 4-го семестра. Документ должен быть оформлен правильно и без ошибок и сохранен в последней редакции для использования в 5-м семестре (один экземпляр ТЗ остается у преподавателя, а другой у студента).

### 18.27. 18.27 КР

В 4-м семестре студенты выполняют курсовую работу по дисциплине СП. Требования к выполнению КР, варианты КР и требования к документации размещены на сайте и периодически обновляются. Работа заключается в следующих основных заданиях:

- разработка проекта программы резидента по заданному варианту,
- разработка и отладка программы резидента,
- разработка комплекта документации на программный продукт,
- сдача и защита КР на основе предоставленных материалов и проведения приемно-сдаточных испытаний, которые проводятся на основе специальной программы испытаний.

КР работа выполняется и защищается каждым студентом индивидуально. По результатам проверки документации и проведенных испытаний проставляется оценка, которая идет в диплом.

### 18.28.18.28 Автономная программа для выгрузки TSR

В вариантах курсовой работы предусмотрен случай выгрузки резидента с помощью специальной автономной программы. В этом варианте необходимо создать программу, которая обращается к резиденту с помощью прерывания 2D (2F). С помощью этого прерывания сначала нужно проверить наличие резидента в памяти, а затем выдать сигнал выгрузки. Далее показан пример программы, в которой выполняются такие действия.

```
; Программа выгрузки резидента (unload.asm)
MYCODE SEGMENT 'CODE'
    ASSUME CS:MYCODE
START:
; Загрузка сегментного регистра данных DS
    PUSH CS
    POP DS
; Проверка наличия резидента в ОП
    MOV AH, 0EEh
    MOV AL, 1 ; Проверка наличия
```

```

    INT 2Dh
    CMP AL, 0FFh
    JNE NO_LOAD ; переход если резидент не(!) в памяти
; Выгрузка
    MOV AH, 0EEH
    MOV AL, 2
    INT 2Dh
    CMP AL, 0FFH ; проверка ответа от нашего резидента должно быть
0FFH
    JNE ERR
; Сообщение об успешной выгрузке
    MOV AH, 09H
    MOV DX, OFFSET MSG_UNLD
    INT 21H
    JMP EXIT
; Ошибка выгрузки
ERR:
    MOV AH, 09H
    MOV DX, OFFSET MSG_ERR
    INT 21H
    JMP EXIT
; Сообщение о том, что уже загружен
NO_LOAD:
    MOV AH, 09H
    MOV DX, OFFSET MSG_NOUNLD
    INT 21H
; Выход из программы
EXIT:
    MOV AL, 0
    MOV AH, 4CH
    INT 21H
; Сообщения
MSG_NOUNLD DB 'TSR is NO in memory!', 10,13,'$'
MSG_UNLD DB 'TSR was unloaded (Unload programm)!', 10,13,'$'
MSG_ERR DB 'Ошибка выгрузки резидента!', 10,13,'$' ; ASCII!!!
MYCODE ENDS
END START

```

Компиляция данной программы должна быть выполнена в формате .EXE. Мы получим **UNLOAD.EXE**. Запуск этой программы при загруженном резиденте приводит к следующей реакции в командной строке:

```

C:\work> unload
C:\work>TSR was unloaded (Unload programm)!

C:\work>_

```

Если резидента в памяти не обнаружено, то получим:

```

C:\work> unload
C:\work> TSR is NO in memory!

C:\work>_

```

Выгрузку можно выполнить и другим способом, рассмотренным выше, без использования обработчика собственного прерывания (2D), при этом должны быть восстановлены старые обработчики и освобождена оперативная память.

### 18.3 18.3 Пример резидентной программы

Ниже приводится текст простой (функционально) резидентной программы, которая содержит все системные требования, заданные в курсовой работе студентов по дисциплине СП.

Данная программа, после выполнения загрузки резидента, при нажатии на функциональную клавишу F1 выводить на экран дисплея сообщение –“ TSR: F1 - pushed!!!”. Перечень основных функций перечислен в заголовке программы.

```
;      Пример простой резидентной программы  ( tsrpos.asm )-> tsrpos.com
;      По F1 - выводиться строка
;      По Ctrl+F1 - программа выгружается
;      Выполняется проверка повторности запуска TSR
;      Выполняется вызов справки по /H или /h
;      Выполняется выгрузка из части инициализации /U или /u

CODEPR  SEGMENT PARA
    ASSUME CS:CODEPR , DS:CODEPR
    PSP:          ORG  100H                      ; Область PSP
    ;;;;;;;;;;;;;;
; Резидентная часть
BEGIN:    JMP     INIT                          ; Переход к части инициализации
;      Данные резидента
SAVEINT9 DD  ?                                ; Сохранение старого обработчика
SAVEINT2D DD  ?                               ; Сохранение старого обработчика
FLAG DB    0
;      Новый обработчик прерывания 09H
NEWINT9:
    CLI
    PUSH  AX                      ; Сохранение используемых регистров
; Вызов старого обработчика
    PUSH  ES
    PUSH  DS
    PUSH  CS
    POP   DS
    ;
    PUSH  AX
    PUSH  BX
    PUSH  CX
    PUSHF
    CALL  CS:SAVEINT9             ; Вызов старого обработчика      с возвратом
;      Запись в AX и AL кодов из буфера
;      MOV AX , 40H
;      MOV ES , AX
;      MOV BX , ES:1AH
;      MOV AX,  ES:[BX]
;      MOV AH, 1
;      INT  16H
;      JZ  PROD
;      Проверка расширенного кода
;      CMP AL , 0
;      JE  EXT
;      CMP AH, 1DH
; Для чистого CTRL
;      JNE PROD
;      MOV AH,00H
;      INT 16H
PROD:
    POP  CX
    POP  BX
    POP  AX
    JMP  END9
;      Проверка нажатия F1
EXT:    CMP AH ,  3BH ; Код F1
;      JE PRINT1                      ; Печать строки      по F1
; Проверка выгрузки
```

```

        CMP AH , 5EH          ; Нажата Ctrl + F1
        JNE  PROD
UNLD:
; Выгрузка
        PUSH ES
        MOV  DX,OFFSET REZMSG1      ; Вывод сообщение о завершении
        CALL PRINT                  ; резидента
        POP ES
;
        PUSH DX
        PUSH ES
;
; 9H
        mov AX,2509H      ; Восстановление обработчика прерывания 05H
        lds DX,CS:SAVEINT9
        int 21H
; 2FH
        mov AX,252DH      ; Восстановление обработчика прерывания 2FH
        lds DX,CS:SAVEINT2D
        int 21H
; получим из PSP адрес собственного окружения и выгрузим его
        MOV  ES,CS:2CH
        MOV  AH,49H
        INT  21H
; выгрузим теперь саму программу
        PUSH  CS
        POP   ES
        MOV  AH,49H
        INT  21H
; Код в AL для успешной выгрузки
        MOV AH,00H
        INT 16H
;
        POP  ES
        POP  DX
        POP  CX
        POP  BX
        POP  AX
;
        POP  DS
        POP  ES
        POP  AX
        STI
        IRET
; Вывод цепочки F1
PRINT1:
        PUSH ES
        MOV  DX,OFFSET REZMSG2      ; Вывод сообщение о F1
        CALL PRINT
        POP ES
; Очистка буфера клавиатуры
        PUSH ES
        MOV  AX , 40H
        MOV  ES , AX
        MOV  BX , ES:1AH
        MOV  ES:1CH , BX
;
        POP  ES
        JMP  PROD
;

```

```

END9:
        POP DS
        POP ES
        POP AX
        STI
        IRET
;
;
;
NEWINT2D PROC FAR
        CLI
        PUSH DS
;
        PUSH CS
        POP DS
        CMP AH, 0EEh ; собственный код
        JE MY_OBR
        JMP OLD_OBR
; Новый обработчик
MY_OBR:  CMP AL, 1 ; Проверка наличия
        JE EXIT_2F
        CMP AL, 2 ; Выгрузка
        JE UNLOAD
;
EXIT_2F: MOV AL, 0FFh ; Есть в памяти
        STI
        POP DS
        IRET
;
UNLOAD:
; Восстановление обработчиков
        PUSH CS
        POP DS
;
        MOV DX, OFFSET REZMSG1 ; Вывод сообщение о завершении
; CALL PRINT ; резидента
;
;
;
;
; 9H
        mov AX, 2509H ; Восстановление обработчика прерывания 05H
        lds DX, CS:SAVEINT9
        int 21H
; 2FH
        mov AX, 252DH ; Восстановление обработчика прерывания 2FH
        lds DX, CS:SAVEINT2D
        int 21H
; получим из PSP адрес собственного окружения и выгрузим его
        MOV ES, CS:2CH
        MOV AH, 49H
        INT 21H
; выгрузим теперь саму программу
        PUSH CS
        POP ES
        MOV AH, 49H
        INT 21H
; Код в AL для успешной выгрузки
        MOV AL, 0FFh
        POP DX
        STI
        POP ES

```

```

        POP DS
        IRET

OLD_OBR:  MOV AL, 0      ; наш резидент не установлен
          STI
          POP DS
          JMP     CS:SAVEINT2D      ; Вызов старого обработчика      2FH
NEWINT2D ENDP
;;;;;;;;;;;;;
; Процедура печати строки для резидентной программы
; т.к. 21H(09H) нельзя
; DX - adress string NEAR (DS - БАЗА)
; Конец строки - '$'
;
PRINT    PROC
          PUSH DI
          PUSH AX
          PUSH CX
          PUSH DX
          PUSH BX
          MOV     DI,DX
;   Позиция курсора в начале
          MOV     AH,03H
          MOV     BH,00H
          INT     10H
;
          MOV     CX , 70

; цикл вывода и проверки символов строки
LOOP1:   MOV     AL,DS:[DI]
          CMP     AL,'$'          ; Конец строки для вывода
          JE      FIN
;   Печать символа на экран
          INC     DI
          MOV     BH,0
          MOV     AH,0AH
          PUSH CX
          MOV     CX,0001H
          INT     10H
          POP CX
; Новая позиция курсора
NEXT4:   MOV     AH,02H
          INC     DL
          MOV     BH,00H
          INT     10H
          JMP     LOOP1
; Конец цикла вывода
FIN:
          POP BX
          POP DX
          POP CX
          POP AX
          POP DI      ; Восстановление стека
          RET
PRINT    ENDP
REZMSG1 db 'TSR was unloaded (Ctrl+F1 - TSR part)!','$'
REZMSG2 db 'TSR: F1 - pushed!!!', '$'
;;;;;;;;;;;;;
; Часть инициализации
INIT:    CLI          ; Запрет прерываний
; Установка DS
          PUSH CS

```

```

        POP DS
; проверка параметров
        MOV AL, CS:80H
        CMP AL, 0
        JNE INIT_UND ; Есть параметры
; Проверка в памяти
INIT1:    PUSH CS
            POP DS
            MOV AH, 0EEh
            MOV AL, 1 ; Проверка наличия
            INT 2Dh
            CMP AL, 0FFh
            JNE INIT2 ; переход если резидент не(!) в памяти (на загрузку)
; Сообщение о том, что уже загружен
            MOV AH, 09H
            MOV DX, OFFSET MSG_INMEM
            INT 21H
            ; Выход из программы без установки резидента
EXITTSR:
            MOV AL, 0
            MOV AH, 4CH
            STI
            INT 21H
; Выгрузка или справка
INIT_UND:
; Есть параметры
;//////////////////////////////////////// разборка параметров
            MOV SI, OFFSET (PSP + 81H)
            LODSB ; первый пробел
            LODSB
            CMP AL, '/'
            JNE ERR_PAR
            LODSB ; символ параметра
            CMP AL, 'H'
            JE INIT_HELP
            CMP AL, 'h'
            JE INIT_HELP
            CMP AL, 'U'
            JE INIT_UND1
            CMP AL, 'u'
            JE INIT_UND1
            JMP ERR_PAR
; Выгрузка резидента
INIT_UND1:
            MOV AH, 0EEH
            MOV AL, 2
            INT 2Dh
            CMP AL, 0FFH
            JE INIT3
; Сообщение о том что нет выгрузки
            MOV AH, 09H
            MOV DX, OFFSET MSG_NOUNLD
            INT 21H
            JMP EXITTSR
; Сообщение о выгрузке
INIT3:
            MOV AH, 09H
            MOV DX, OFFSET MSG_UNLD
            INT 21H
            JMP EXITTSR
; Справка
INIT_HELP:

```

```

        MOV AH , 09H
        MOV DX, OFFSET MSG_Help
        INT 21H
        JMP EXITTSR
ERR_PAR:
        MOV AH , 09H
        MOV DX, OFFSET MSG_Param
        INT 21H
        JMP EXITTSR
; Получение адреса старого обработчика          9
INIT2:  MOV     AH, 35H
        MOV     AL, 09H      ; Номер прерывания
        INT 21H
; Сохранение адреса старого обработчика
        MOV     WORD PTR SAVEINT9 , BX
        MOV     WORD PTR SAVEINT9+2 , ES
; Установка нового обработчика в вектор прерывания          9
        MOV     AH,25H
        MOV     AL, 09H      ; Номер прерывания
        MOV     DX, OFFSET NEWINT9
        INT 21H
; Получение адреса старого обработчика          2F
        MOV     AH, 35H
        MOV     AL, 2DH      ; Номер прерывания
        INT 21H
; Сохранение адреса старого обработчика
        MOV     WORD PTR SAVEINT2D , BX
        MOV     WORD PTR SAVEINT2D +2 , ES
; Установка нового обработчика в вектор прерывания          2F
        MOV     AH,25H
        MOV     AL, 2DH      ; Номер прерывания
        MOV     DX, OFFSET NEWINT2D
        INT 21H
; Вывод сообщения о загрузке резидента
        MOV AH , 09H
        MOV DX, OFFSET MSG
        INT 21H
; Завершить и оставить резидентной (TSR)
        MOV     AX, 3100H
        MOV     DX, (init - begin +10FH)/16      ; Размер резидента
        STI                      ; Разрешение прерываний
        INT 21H
; Данные части инициализации
MSG      DB 'Start TSR program!', 10,13,'$'
MSG_INMEM DB 'TSR already is in memory!', 10,13,'$'
MSG_UNLD  DB 'TSR was unloaded (Init part)!', 10,13,'$'
MSG_NOUNLD DB 'TSR is NO in memory!', 10,13,'$'
MSG_Help  DB 'Test - sample TSR - 2010', 10,13,'Parameters: /H or /h - help
, /U or /u - unload TSR ', 10,13,'€-д@а- жёпя!',10,13,'$'          ; Последний
текст "Информация" - введен в коде ASCII!!!
MSG_Param  DB 'Error - command line parameters!', 10,13,'$'
CODEPR    ENDS
                END      BEGIN

```

После компиляции и редактирования связей в формате .COM (см. выше) получим исполнимый модуль **TSRPOS.COM**.

Предварительно необходимо проверить размер свободной памяти и его запомнить:

C:\work>mem

...

591664 максимальный размер исполняемой программы

...



При запуске этого модуля без параметров резидент будет загружен в оперативную память:

```
C:\work>tsrpos
C:\work>Start TSR program!

C:\work>_
```

Повторный запуск исполнимого файла **TSRPOS.COM** приводит к выдаче сообщения на экран:

```
C:\work>tsrpos
C:\work>TSR already is in memory

C:\work>_
```

Проверка с помощью утилиты MEM должна показать наличие резидента и его окружения в памяти (напомню, что перед запуском резидента необходимо с помощью утилиты MEM проверить количество свободной памяти и его запомнить!):

```
C:\work>mem /p
```

Адрес	Имя	Размер	Тип
000000		000400	Вектор прерывания
000400		000100	Область обмена ПЗУ (ROM)
...			
00DE20	COMMAND	000510	Данные
00E340	COMMAND	000B50	Программа
00EEA0	COMMAND	0004F0	Окружение
00F3A0	TSRPOS	000510	Окружение
00F8C0	TSRPOS	000260	Программа
00FA30	MEM	000510	Окружение
00FF50	MEM	0174E0	Программа
...			
	589968	максимальный размер исполняемой программы	
...			

При запуске программы в режиме справки резидент не загружается и не удаляется из памяти. На экран выводится следующее сообщение:

```
C:\work>tsrpos /H
Test - sample TSR - 2010
Parameters: /H or /h - help , /U or /u - unload TSR
Информация!
C:\work>_
```

При нажатии на клавишу F1 резидент выдает следующее сообщение в текущую позицию курсора дисплея (курсор расположен после последнего символа):

```
C:\work>TSR: F1 - pushed!!!_
```

При использовании комбинации Ctrl+F1 резидент будет выгружен с помощью своих процедур:

```
C:\work>TSR was unloaded (Ctrl+F1 - TSR part)!_
```

Повторная проверка с помощью MEM должна показать отсутствие резидента в памяти, а размер свободной памяти должен восстановиться к начальному состоянию.

```
C:\work>mem
```

...			
	591664	максимальный размер исполняемой программы	
...			

Числовые значения для различных компьютеров могут отличаться от величин, приводимых здесь (это зависит от конкретной конфигурации ОС и установок конкретного компьютера), но значения, полученные до загрузки резидента и после его выгрузки должны совпадать в точности до единицы (У нас в примере **591664 = 591664**).

Выгрузку резидента можно выполнить в нашем случае и из командной строки:

```
C:\work>tsrpos /U
TSR was unloaded (Init part)!
```

```
C:\work>_
```

Контроль выгрузки с помощью МЕМ здесь выполняется аналогично. При задании ошибочных параметров будет выдано сообщение:

```
C:\work>tsrpos /L
```

```
Error - command line parameters!
```

```
C:\work>_
```

Программа данных проверок должна быть оформлена в специальном документе (**“Программа и методика испытаний”** – см. требования к документации КР на сайте), входящем в состав пакета документации на КР и должна обязательно содержать:

- условия проведения испытаний и
- последовательность проверок, оформленных в виде таблицы, с обязательными ссылками на пункты функциональных требований ТЗ.

Проведение проверок при сдаче КР может быть выполнено выборочно, поэтому отдельные строки таблицы проверок должны обеспечивать последовательность логически завершенных действий, не зависящих от других пунктов проверок.

Важнейшим этапом разработки программ и программных систем является формализация процесса выполнения операций (шагов, действий, вложенных процессов и т.д.). Этот этап также называют разработкой алгоритма программы. Для такой формализации (формализованного описания) должны быть использованы формализованные языки, не содержащие неоднозначностей. Существует много подобных графических языков, но, на мой взгляд, самым удачным является формализованный язык блок-схем.

Нужно иметь в виду, что этот язык подходит не для всех описаний используемых при разработке программного обеспечения, а в первую очередь используется для описания последовательности действий и команд. Например, для описания систем классов применяется язык диаграмм классов, для описания структур баз данных графический язык сущностей и связей. Однако, если Вам необходимо разработать метод класса (процедуру, функцию и т.д.) даже средней сложности, то при разработке ее и ее отладки обойтись без блок-схемы. Я уже не говорю о документировании программного обеспечения, сохранения информации о проекте для последующего сопровождения и модернизации.

Для лабораторных работ по дисциплине Системное программирование, при разработке командных файлов и программ на языке Ассемблер, целесообразно (и требуется) использование блок-схем как основного языка для описания алгоритмов. Поэтому, я в данное пособие включил этот раздел.

Язык блок-схем является графическим языком описания алгоритмов. В общем случае, блок-схема это ориентированный граф из специальных элементов - блоков (вершины) и связей (стрелки и линии).

### **21.1.21.1 Назначение блок-схем программ**

Блок-схемы (я буду использовать название с дефисом, хотя допустимо и другое) позволяют обеспечить следующее:

- Создать формализованное описание алгоритма работы разрабатываемой программы.
- На смысловом уровне и на логическом уровне проверить правильность описанного алгоритма.
- Вручную (по шагам) выполнить создаваемую программу и проверить корректность выполняемых действий и получаемого результата.
- Служить основой для создания программы на конкретном языке программирования. Обращаю Ваше внимание, что только в этот момент может быть принято решение о выборе языка программирования, причем именно характер блок-схемы, или ее вид, позволяет сделать такой выбор более точно.
- Служить подсказкой при отладке программ с помощью универсальных отладчиков.
- Не исключено, что по написанной программе может быть повторно построена блок-схема, а ее можно сравнить с блок-схемой, построенной на этапе проектирования. Это сравнение может привести к следующим выводам: программа построена неверно, блок схема построена неверно или, что не учтены важные особенности разработанного алгоритма.

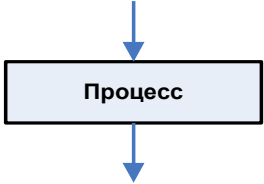
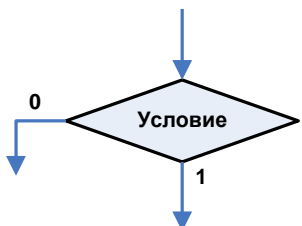

Думаю, что данный перечень может быть продолжен и далее, а здесь я отмечу, что блок-схема программы позволяет представить программу и алгоритм в целом, в виде единого образа, по которому можно дать ее оценку: плохая программа или хорошая. Например, если в блок-схеме много связей (в том числе и избыточных), то, возможно, следует подумать о том, чтобы использовать процедуры и, тем самым, упростить понимание алгоритма и написанной программы. Считаю, что программист, который не умеет правильно оформлять блок-схемы и не использует их в своей работе, не может претендовать на звание профессионального программиста!

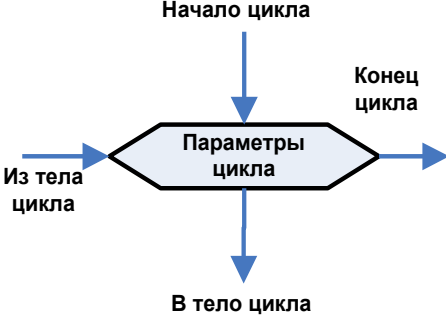
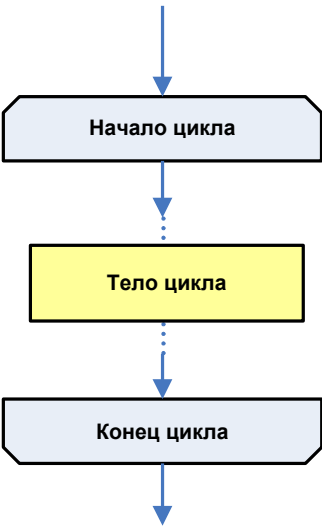
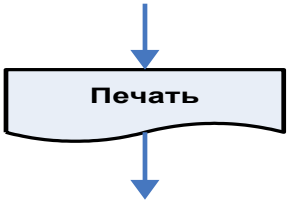
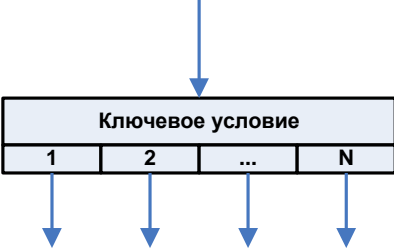
В зависимости от этапа разработки алгоритмов программ, блок-схемы могут иметь разную степень детальности. Более того, технология разработки блок-схем должна предусматривать последовательное раскрытие деталей алгоритма, постепенное уточнение действий и операций, выполняемых для решения задачи. Самый детальный способ описания программы – это описание на уровне команд или операций языка. Хотя этого не всегда нужно, тем не менее, это возможный и самый низкий уровень описания, к которому нужно стремиться, в частности при документировании программ.

Отмечу еще одну особенность блок-схем, которая иногда приводит к недоумению.

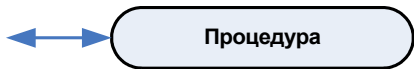
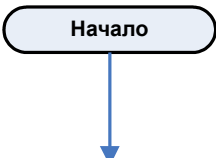
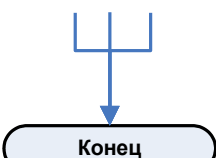
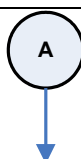
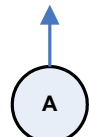
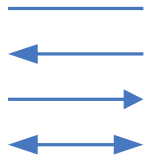
### 21.2.21.2 Элементы блок-схем программ

Блок-схемы, как было уже отмечено, представляют собой ориентированный граф, в вершинах которого размещаются специфические элементы. Каждый элемент имеет особую структуру, свое назначение и обозначение. Самое существенное отличие элементов – число входящих и выходящих стрелок. Логика работы элемента блок-схемы в основном и определяется их числом. Основные элементы (их обозначения и назначение) представлены в таблице расположенной ниже.

№	Обозначение элемента	Назначение	Примечание
1.		Элемент типа процесс описывает последовательность действий и операций. Операции выполняются одна за другой без условий. Имеет одну входящую стрелку и одну выходящую.	Используется как элемент последовательной детализации.
2.		Элемент ветвления, типа условие. Если условие при проверке получает значение истина (1, ДА, TRUE), то деле выполняется переход по стрелке помеченной 1. Если условие получает на конкретном шаге значение ложь (0, НЕТ, FALSE), то выполняется переход по стрелке помеченной 0.	Можно использовать обозначения стрелок (1, ДА, Истина, TRUE) или (0, НЕТ, ЛОЖЬ, FALSE) соответственно.
3.		Это вариант конструкции условия. Описание аналогично. Возможны и другие варианты стрелок. Самое главное: одна входит, а две выходят.	

№	Обозначение элемента	Назначение	Примечание
4.		Данный элемент, элемент описания циклов, имеет две входящие стрелки и две выходящие стрелки. Назначение стрелок показано на рисунке. В поле данного блока записываются условия продолжения цикла и изменяемые на каждом его шаге параметры.	Данный элемент рекомендуется для оформления блок-схем лабораторных работ.
5.		Другой элемент для организации циклических фрагментов программ. Он соответствует ГОСТу. Этот элемент состоит из двух частей, в первой записываются начальные условия цикла, а во второй условия его продолжения.	При организации вложенных циклов не всегда является наглядным. Элемент не является обязательным для ЛР.
6.		Элемент для обозначения вывода информации, в частности на печать. Данный элемент похож на элемент типа процесс и во многих случаях может быть им заменен.	В ЛР лучше использовать процесс.
7.		Элемент типа переключатель. Имеет один вход и множество выходов. По ключевому условию определяется (вычисляется) номер выхода, по которому производится передача управления.	Рекомендуется для выполнения ЛР по курсу.

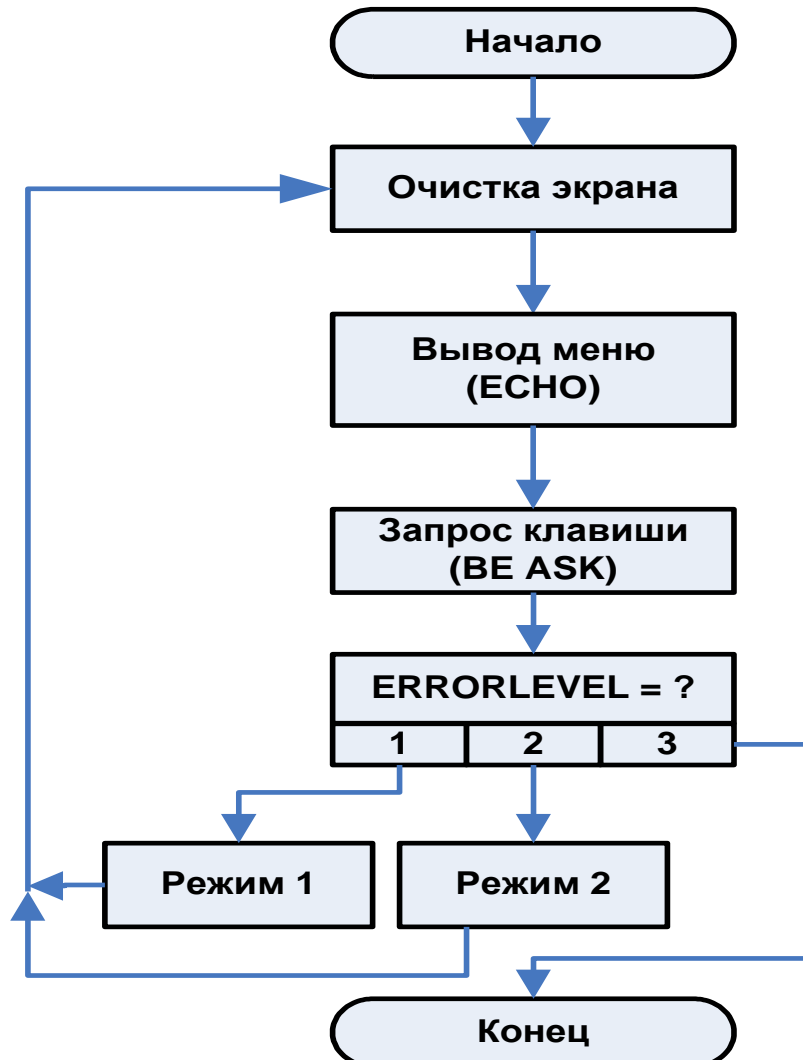
№	Обозначение элемента	Назначение	Примечание
8.		<p>Это другая форма блока переключателя. Но есть и отличия, например, условия, могут быть разными и число выходов больше на единицу, так как предусмотрен вариант, когда ни одно условие не сработало (по умолчанию - default).</p>	<p>Применять разновидности блоков переключателя желательно на основе логики работы программы.</p>
9.		<p>Здесь представлена конструкция, которая не является отдельным элементом блок-схемы, а состоит из трех важных блоков, которые должны присутствовать в любом цикле: задание начальных условий цикла (НУ), тело цикла, проверка условия завершения/продолжения цикла.</p>	<p>Эту простейшую, но обобщенную блок-схему я включил для более четкого понимания циклов. Такую конструкцию можно использовать с оператором IF, if.</p>
10.		<p>Здесь также представлена конструкция, которая не является отдельным элементом блок-схемы, а состоит из трех важных блоков, аналогичных предыдущей конструкции. Отличие заключается в том, что теоретически возможен цикл без единого выполнения тела цикла (Сравни конструкции DO и WHILE).</p>	<p>Эту простейшую, но обобщенную блок-схему я включил для более четкого понимания циклов. Такую конструкцию можно использовать с оператором IF, if.</p>

№	Обозначение элемента	Назначение	Примечание
11.		Такой элемент используется для выделения в блок-схемах участков вызова процедур (функций, методов, подпрограмм и т.д.). Двойная стрелка показывает то, что выполняется возврат из процедуры.	Необязательный элемент блок-схем. В поля блока записывается название процедуры.
12.		Данный элемент определяет начало алгоритма, начало выполнения программы. Чаще всего в одной блок-схеме такой элемент один. Имеет одну выходящую стрелку.	Для программ с несколькими входами начальные элементы должны быть помечены.
13.		Данный элемент определяет конец алгоритма, конец выполнения программы. Таких элементов может быть несколько, в зависимости от логики работы программы. Имеет одну входящую стрелку.	Элемент окончания может иметь один или несколько входов.
14.		Данный (и следующий) элемент используется для условного разрыва соединяющих линий. В поле элемента записывается обозначение (чаще буква или цифра). Это обозначение должно соответствовать обозначению другой точки разрыва.	
15.		Этот элемент является парным к предыдущему. Они позволяют организовать разрыв линий на одном листе или на разных листах.	
16.		Элементы в виде простых линий и линий со стрелками задают связи между блочными элементами блок-схем. Для линий без стрелок подразумевается стрелка: сверху - вниз и слева - направо. Линии с двойными стрелками используются для обозначения вызова процедур.	Линии могут быть ломаными, но только под прямым углом (ГОСТ). Предпочтительнее использовать линии со стрелками.

### 21.3.21.3 Примеры блок-схем программ

В данном разделе мы рассмотрим несколько примеров блок-схем для иллюстрации использования элементов и приемов построения алгоритмов. Примеры простые и соответствуют программам, которые рассмотрены в других разделах настоящего пособия. Перед блок-схемой дана ссылка на раздел, в котором описана программа ей соответствующая. Если щелкнуть мышкой на ссылке (с клавишей Ctrl) можно оперативно перейти к тесту программы.

Пример блок-схемы программы для командного файла, рассмотренного в разделе 5.7. Примеры командных файлов. Данный командный файл соответствует программе, которую нужно разработать в ЛР номер 2. В блок-схеме этого примера продемонстрировано использование переключателя и элементов типа процесс. Отметим, что переключатель в



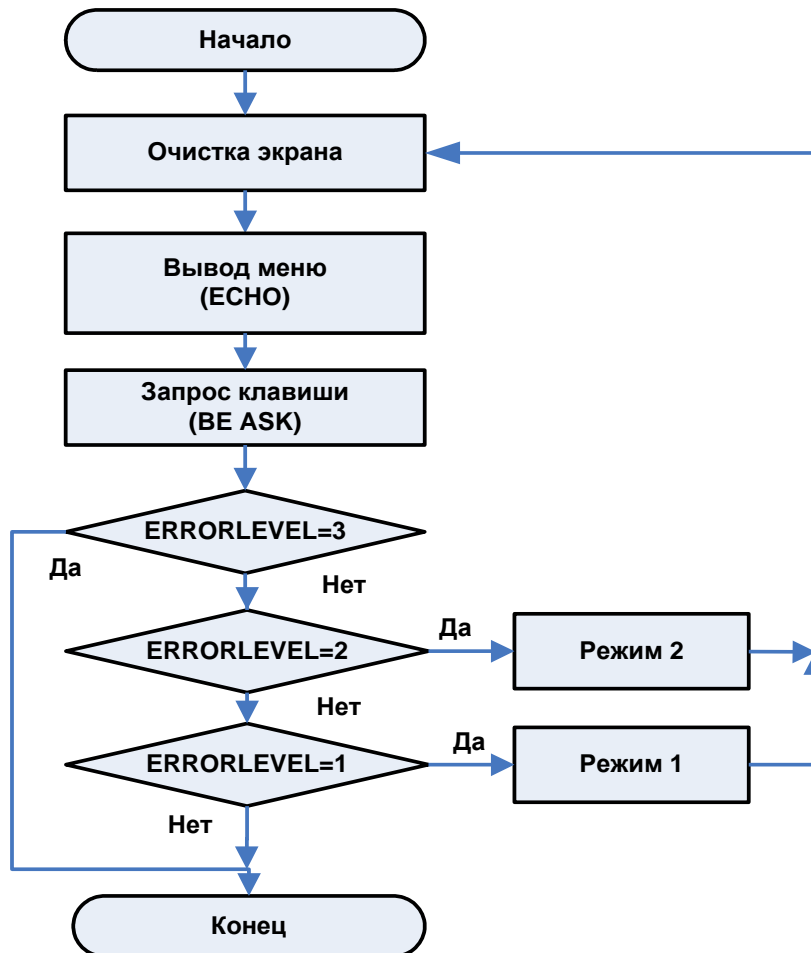
командном файле реализован с помощью директив IF, поэтому, строго говоря, блок-схема не совсем точна, хотя полностью отражает алгоритм программы. Причина неточности данной блок-схемы заключается в том, не ясно куда мы будем двигаться в том случае, когда будет получен результат  $ERRORLEVEL < 1$ . Вспомните, что сравнение выполняется по условию  $\geq$ . При использовании BE ASK допустимо назначить ключ возврата по умолчанию, например 3. Тогда программа завершится даже при невозможности задания



нужного номера с клавиатуры (отличными от 1, 2, 3). При использовании CHOICE такое невозможно, так как ключа по умолчанию задать нельзя.

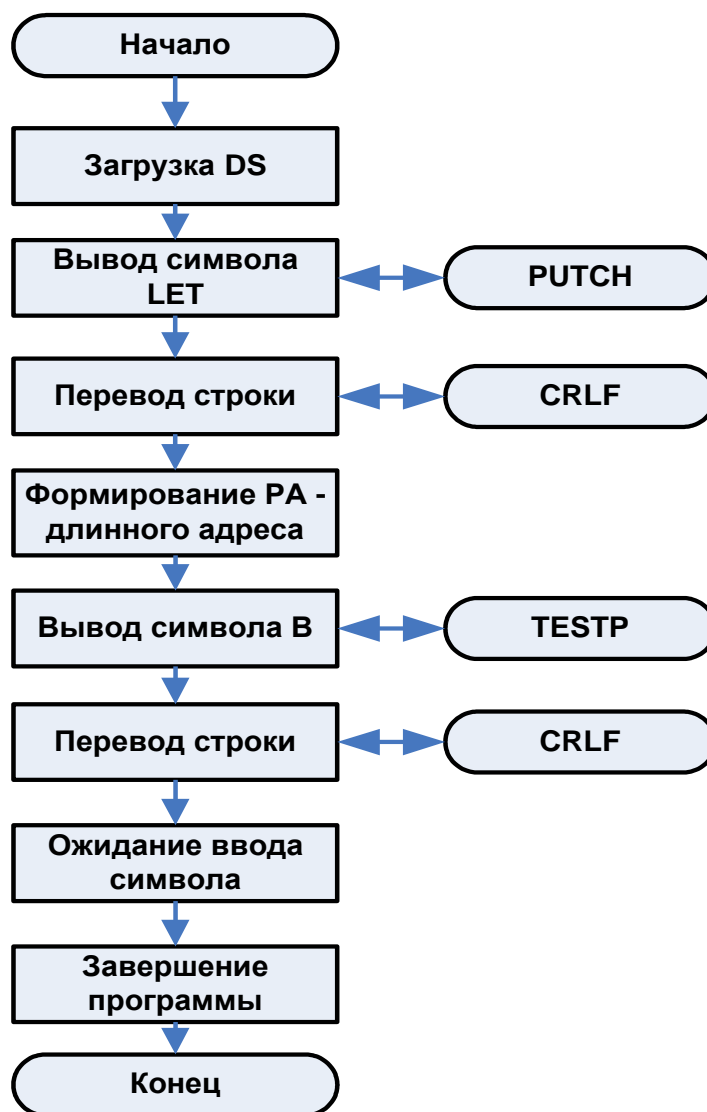
Когда выбран метод реализации алгоритма можно приступить к написанию программы. Если блок-схему для нашего командного файла построить после написания программы, то она может выглядеть так, как показано на следующем рисунке.

Эта блок-схема является более точной, так как может правильно работать и в том случае когда код возврата  $< 1$ .



В этой схеме переключатель построен на основе второй конструкции.

Пример блок-схемы алгоритма программы на языке Ассемблер с процедурами ( см. раздел 10.5. Пример программы с процедурами ). Следующая блок-схема показывает как использовать блоки вызова процедур.

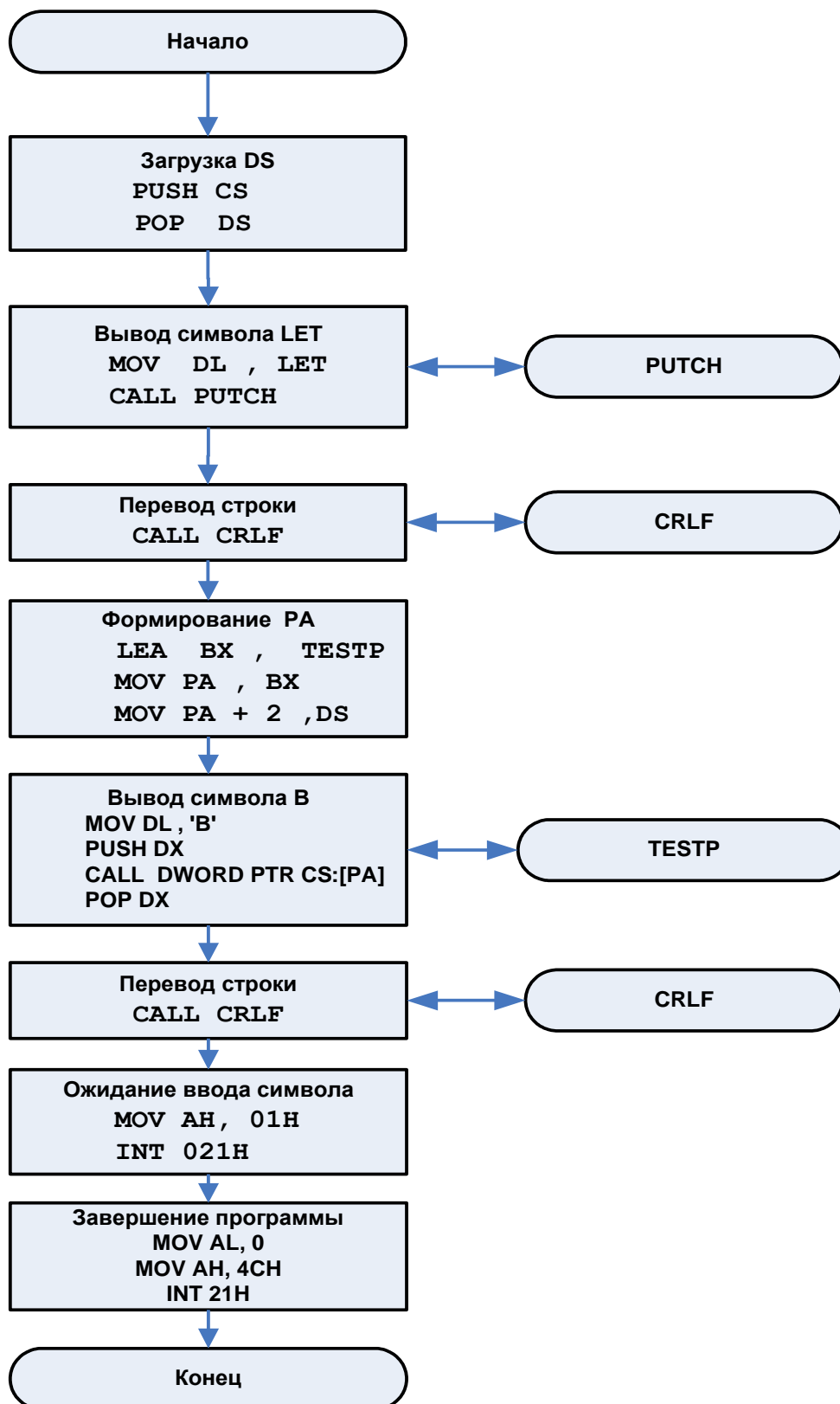


Для отдельных процедур нужно разработать отдельные блок-схемы и представить в отчете. В методических указаниях отмечено, что если Вы используете повторно одни и те же процедуры, то допускается не повторять в новых отчетах блок-схемы этих процедур. Программа для этой блок-схемы очень проста. Однако, блок-схема нужна для детального описания шагов, которые нужно выполнить для решения задачи.

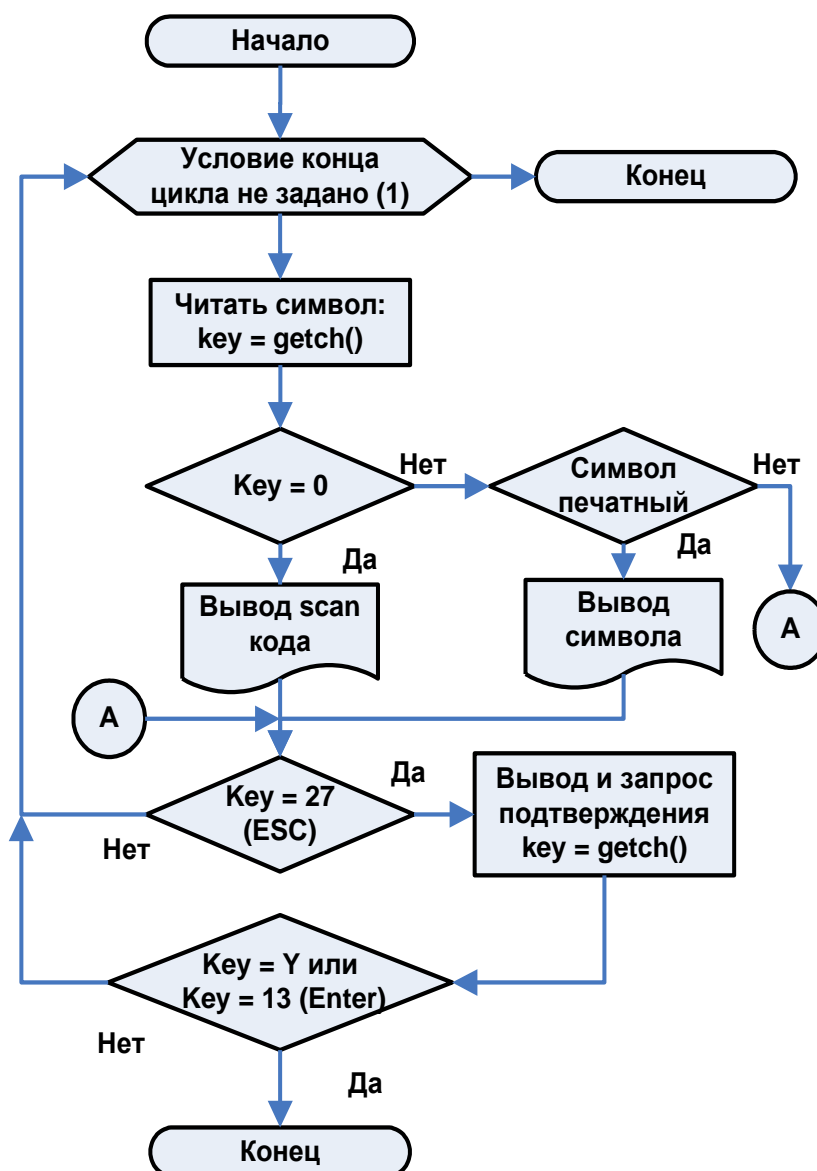
В детальной блок-схеме мы можем расписать алгоритм работы программы вплоть до отдельной команды. Это позволяет процедуру кодирования сделать простой и формализованной. Возможно в группах разработчиков и такое: один программист-аналитик разрабатывает алгоритмы и оформляет детальные блок-схемы, а другие, причем менее квалифицированные выполняют кодирование и отладку программ. Такое разделение возможно, так как процесс создания алгоритмов решения задач более трудоемок, требует большего опыта

и квалификации. Часто бывает так, что начинающие программисты занимаются кодированием программ до тех пор пока не получают достаточного опыта и знаний для самостоятельной разработки сложных алгоритмов программ.

Детальная блок-схема программы на Ассемблере для вывода 2-х символов приведена ниже.



Ниже представлена блок-схема процедуры для программы печати символов и скан кодов клавиш ( см. раздел **Ошибка! Источник ссылки не найден.** ).



В этой программе в интерактивном режиме считываются коды нажатых клавиш, и производится их распечатка. Программа написана на языке СИ. В данном примере проиллюстрировано: использование циклической конструкции. Использование условных элементов и применение элементов для разрыва линий связи в блок-схеме.

В программе организован бесконечный цикл. Выход из цикла выполняется в теле цикла при проверке нажатия клавиши ESC (код 27) и подтверждения клавишами Enter или клавишей “Y”. При считывании кода клавиши проверяется первый байт. Значение первого байта указывает на ввод печатного символа или ввод скан кода ( $key = 0$ ). Подробно о скан-кодах смотрите в разделе пособия 23.5. SCAN – коды.

Проверки в программе выполняются оператором **if** и условным выражением. Условное выражение (**isgraph( key ) ? key : ' '**) вставлено в вызов функции печати – **printf**. Функция **isgraph** проверяет признак печатного символа и отображает символ (**key**) или пробел. Отмечу, что в блок-схеме эта логика отображается условным элементом, в программе вставлена в вызов функции. Кстати, это одна из причин того, почему не всегда программы сгенерированные автоматически являются эффективными и наглядными. После проверки завершения по ESC в программе выполняется запрос символа подтверждения и проверка его ввода. Для этого используются условный элемент блок-схемы.

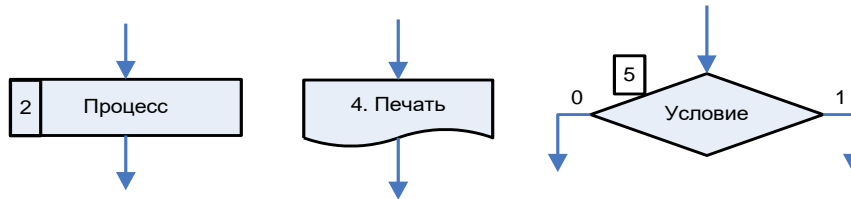
И последнее, с помощью элемента в кружочке (“А”) мы показываем, как сделать разрыв линии связи. Это позволяет не тянуть связующую линию через всю страницу и избежать пересечения линий. После выяснения того, является ли символ печатным или нет, мы должны перейти на проверку завершения цикла. Это делается с помощью добавления элемента с символом “А” в место проверки клавиши завершения.

#### 21.4.24.4 Оформление блок-схемы программы

При оформлении блок-схем нужно соблюдать формальные, визуальные и семантические правила. Формальные правила задаются языком блок-схем и изображением его элементов. Визуальные правила позволяют сделать блок-схему более наглядной. Смысловые правила определяются алгоритмом конкретной задачи разработки программы. Перечислим основные правила:

- Все элементы должны иметь точно такое число входов и выходов, которое определяется изображением элемента.
- Должны существовать пути из начального элемента хотя бы в один конечный элемент.
- Не должно быть элементов типа процесс с одним входом и без выхода.
- Данные в блок-схемах не описываются.
- Из элемента типа процесс не может быть более одного выхода.
- Элементы блок-схемы должны занимать равномерно все пространство страницы или листа, на котором они располагаются.
- Элементы должны иметь приблизительно равные размеры и одинаковое форматирование.
- Расстояние между элементами должно быть по возможности одинаковым.
- Элементы и линии связи не должны располагаться очень близко друг от друга.
- Линии связи не должны пересекаться (только в порядке исключения).
- Если линии связей не имеют стрелок, то направление стрелки подразумевается так: слева направо и снизу вверх. Если возникают неоднозначности, то желательно использовать концевые стрелки.
- Блоки в блок-схеме желательно пронумеровать, в этом случае удобнее создавать описание блок-схем программ.
- Толщина линий связи должна быть одинакова.
- Блок-схема должна оформляться с помощью стандартных программ рисования, например MS VISIO или MS WORD.

Нумерация блоков может быть выполнена различными способами. На рисунке ниже показаны варианты такой нумерации.



Для описания блок-схем очень удобно использовать программный продукт MS VISIO, причем можно воспользоваться любой версией программного продукта. Рисунки сделанные в VISIO можно легко вставить в документы MS, в частности MS WORD.

Примечание. После вставки рисунка желательно выполнить две операции:

- Выполнить форматирование объекта рисунка так, чтобы он раздвигал текст по горизонтали (выделить рисунок, формат объект =>Закладка Положение =>Кнопка дополнительно => Закладка Обтекание => Выбор Снизу и верху => ОК =>ОК ).
- Выполнить привязку рисунка к абзацу. Абзац лучше выбрать над рисунком. ( Включить режим просмотра непечатаемых знаков - п, выделить рисунок, переместить якорь перед абзацем слева, формат объект =>Закладка Положение =>Кнопка дополнительно => Закладка Положение рисунка => Галочка установить привязку => ОК =>ОК). После этого на якорь появиться замочек. Рисунок будет перемещаться вместе с абзацем. Нужно рассчитать так, чтобы и рисунок и абзац помещались на одной странице документа.

Для рисования в VISIO нужно выбрать метрический шаблон объектов с названием “Basic Flowchart Shapes”. В этом шаблоне содержатся все необходимые объекты для рисования блок-схем.

### 21.5.21.5 Блок-схемы и описания данных

В блок-схемах не предусмотрено элементов для описания данных. Это объясняется тем, что блок-схема используется для описания действий (операторов), а описания переменных никаких действий в программе обычно не выполняют. Поэтому описание переменных, так называемая спецификация переменных программы производится отдельно. Лучше всего это сделать в отдельной таблице, в которой отображается: название и тип переменной, ее назначение и использование. Примером такой таблицы для программы вывода символов может быть следующая таблица:

№	Название	Тип	Назначение	Где используется
1.	LET	DB	Для вывода символа на экран	В программе и процедуре PUTCH

Попытка описания данных в лабораторных работах и рейтингах будет расцениваться как ошибка, и свидетельствовать о незнании правил написания блок-схем и логики программ.

## Макросы и их применение

### 22.1. 22.1 Назначение макросов

Макрокоманды – это очень интересный механизм для разработки программ на языке Ассемблер. Макрокоманды или иначе макросы (на жаргоне программистов) обрабатываются на этапе компиляции и позволяют сделать программу более наглядной и обозримой. Макросы могут использоваться для создания собственного языка программирования из команд, записанных пользователем. В программе на языке Ассемблер Вы можете встретить:

- Машинные команды,
- Директивы Ассемблера
- Макрокоманды и
- Комментарии.

Для использования макросов в программах на языке Ассемблер нужно:

- Описать макрокоманды, дать определения макроса.
- Выполнить вызов макрокоманды (макрывызов)
- Проверить правильность работы макрокоманды (проверить правильность генерации макрорасширения).

Рассмотрим отдельно эти действия.

### 22.2. 22.1 Описание макросов

Описание макроса или макроопределение задается в следующем формате:

```
<имя макроса> MACRO <список формальных параметров макросов>
...
<тело макрокоманды>
...
        ENDM
```

Тело макрокоманды состоит из обычных машинных команд, записанных в правилах синтаксиса Ассемблера, директив языка Ассемблер, команд включающих формальные параметры макроса и специальных директив макрокоманд. Список формальных параметров описывается так:

```
<список формальных параметров макросов>:=<формальный параметр>|
    ,<список формальных параметров макросов >
<формальный параметр>:= <имя параметра>
```

Пример простой макрокоманды для печати символа на дисплей показан ниже:

```
PUTCHAR MACRO SYMBOL
    PUSH AX
    PUSH DX
    MOV DL, SYMBOL
    MOV AH , 02H
    INT 021H
    POP DX
    POP AX
    ENDM
```

Данная макрокоманда сохраняет в стеке регистры AX и DX, выводит символ на дисплей и восстанавливает значения регистров. Хотя данная макрокоманда выглядит универсальной, при ее вызове возможны ошибки. Например, при задании в качестве параметра регистра AX (например, макрвызов – **PUTCHAR AX**) компилятор Ассемблера выдаст ошибку при обработке команды в макросе **MOV DL, SYMBOL**, так как размерности регистров не согласованы (DL – 1 байт и AX – два байта соответственно). Описания макрокоманд могут быть

сложными, из одной макрокоманды можно вызывать другие макрокоманды, создавая, таким образом, системы макрокоманд, которые могут в том числе подключаться из библиотек.

### 22.3. 22.3 Параметры макросов и макровыводы

Параметры, задаваемые при описании макрокоманды (определении макрокоманды) называются формальными, а параметры, задаваемые при использовании макрокоманды (макровыводы) называются фактическими. Вызов макрокоманды или макровывода задается так:

```
<имя макрокоманды> <список фактических параметров>
<список формальных параметров макросов>:=<формальный параметр>|
, <список формальных параметров макросов>
<формальный параметр>:= <имя параметра> | <пусто>
```

Как видно из формального описания, при макровыводе параметры могут быть пропущены, при этом запятая должна оставаться на месте. Вызов макрокоманды для предыдущего макроопределения может быть таким:

```
PUTCHAR AL
...
PUTCHAR VAR
...
PUTCHAR 'A'
```

Если при вызове макрокоманды задается несколько параметров, то допускается пропуск параметров:

```
COMPARE VAR1, VAR2, VAR3
...
COMPARE VAR1, , VAR3
...
COMPARE , VAR2, VAR3
...
COMPARE VAR1, VAR2
```

Если параметры при вызове опущены, то в самой макрокоманде нужно предусмотреть проверку наличия конкретного фактического параметра, для чего используются специальные директивы условной компиляции.

### 22.4. 22.4 Параметры по умолчанию

При описании макрокоманды могут быть заданы значения параметров по умолчанию. Это выполняется следующим образом (в дополнение к формальному описанию заголовка макрокоманды):

```
<формальный параметр>:= <имя параметра> |
<имя параметра> = <значение по умолчанию> |
<имя параметра> = REQ
```

Такая возможность существует не во всех макроассемблерах. Приведем пример:

```
PUTCHAR MACRO SYMBOL=REQ , CR=<PER>
```

В этом примере первый параметр является обязательным, а второй параметр имеет значение по умолчанию, которое используется в том случае, когда в макровыводе параметр не задан.



## 22.5. 22.5 Макрорасширения

После компиляции на место макровывоза вставляются команды, полученные на основе макрорасширения. Эти команды должны строго соответствовать синтаксису языка Ассемблер и логике работы программы. Макрорасширение вставляется автоматически и его можно увидеть в листинге программы. Для включения в листинг команд макрорасширения используются специальные директивы:

- **.LALL** - получение в листинге всех макрорасширений
- **.SALL** - подавление в листинге всех макрорасширений
- **.XALL** – восстановление характеристик вывода листинга, заданного по умолчанию.
- **.LIST** – включает печать исходного текста программы.
- **.XLIST** – выключает всю печать исходного текста программы.

Макрорасширения для наших примеров получим в виде:

```
PUTCHAR AL
1  PUSH AX
1  PUSH DX
1  MOV DL, AL
1  MOV AH , 02H
1  INT 021H
1  POP DX
1  POP AX
```

В первой строке макрорасширения показан уровень расширения. В нашем случае это “1”. Если в самой макрокоманде встречается другой макровывоз, то эти строки макрорасширения будут помечены “2”, и так далее.

```
PUTCHAR VAR
1  PUSH AX
1  PUSH DX
1  MOV DL, VAR
1  MOV AH , 02H
1  INT 021H
1  POP DX
1  POP AX
```

В этом случае в регистр DL заноситься переменная

```
PUTCHAR AX
1  PUSH AX
1  PUSH DX
1  MOV DL, AX
1  ***** ERROR *****
1  MOV AH , 02H
1  INT 021H
1  POP DX
1  POP AX
```

Компилятор фиксирует ошибку, так как регистры разной длины.

```
PUTCHAR 'A'
1  PUSH AX
1  PUSH DX
1  MOV DL, 'A'
1  MOV AH , 02H
1  INT 021H
1  POP DX
1  POP AX
```

В последнем макрорасширении мы видим занесение в регистр константы.

## 22.6. 22.6 Директивы написания макрокоманд

Помимо специальных директив описания макрокоманд (MACRO и MEND) в тексте программы на Ассемблере и в самих макрокомандах могут использоваться следующие директивы макрогенерации:

- REPT – директива циклического повторения текста по счетчику.
- IRP - директива циклического повторения текста по списку параметров.
- IRPC – директива циклического повторения по списку символов (строке).
- EXITM – директива завершения работы макрокоманды или цикла
- LOCAL – директива описания локальных переменных и меток макроса.
- IF, IFE, ELSE, ENDIF – директивы условной компиляции (логическое условие).
- IF1, IF2, ELSE, ENDIF – директивы условной компиляции (проходы компилятора).
- IFDEF, IFNDEF, ELSE, ENDIF – директивы условной компиляции (объявление переменных этапа компиляции).
- IFB, IFNB, ELSE, ENDIF – директивы условной компиляции (проверка наличия параметра макрокоманды).
- IFIDN, IFIDIF, ELSE, ENDIF - директивы условной компиляции (сравнение переменных этапа компиляции).
- Служебные символы: “%” , “&” , “!” , “;” – используются для написания макрокоманд.

Рассмотрит ниже назначение и возможности основных директив описания макрокоманд.

## 22.7. 22.7 Циклическая компиляция

Директивы циклической компиляции (REPT, IRP, IRPC) позволяют организовать циклы генерации текста программы.

Директива REPT имеет следующий синтаксис:

```
REPT <выражение этапа компиляции>  
<тело цикла>  
ENDM
```

Параметр <выражение этапа компиляции> определяет число повторений цикла. Тело цикла может содержать любые команды и операторы языка Ассемблер, включая макрокоманды. Цикл генерации будет повторяться заданное число раз. Это число вычисляется до начала цикла и не может изменяться в цикле. Простейший пример цикла:

```
REPT 3  
DB 0  
ENDM  
1 DB 0  
1 DB 0  
1 DB 0
```

Если задана переменная этапа компиляции N, то получим макрорасширение:

```
N = 3  
REPT N  
DB N  
ENDM  
1 DB 3  
1 DB 3  
1 DB 3
```

Если задано выражение, то получим макрорасширение:

```
N = 3  
REPT N - 1
```

```

    DB N
    ENDM

```

```

1   DB 3

```

```

1   DB 3

```

В теле цикла могут быть использованы выражения этапа компиляции:

```

    K = 3
    N = 65 ; код 'A'
    REPT K
    DB N
    N = N + 1
    ENDM
1   DB 65
1   N = N + 1
1   DB 66
1   N = N + 1
1   DB 67

```

Допустимо также использование вложенных циклов:

```

    K = 2
    N = 3
    REPT K
        REPT N
            DB 0
        ENDM
    ENDM
1   REPT N
1   DB 0
1   ENDM
2   DB 0
2   DB 0
2   DB 0
1   REPT N
1   DB 0
1   ENDM
2   DB 0
2   DB 0
2   DB 0

```

Подчеркнутым шрифтом выделены результаты первого прохода компиляции (уровень 1), а окончательные результаты расширения даны на втором уровне.

Директива IRP имеет следующий синтаксис:

```

IRP <переменная цикла>, <<список фактических параметров > >
<тело цикла>
ENDM

```

**<список фактических параметров > := <параметр> | , <список фактических параметров >**

Директива IRP повторяет цикл столько раз, сколько указано фактических параметров в угловых скобках (отмечу, что в данном случае это терминальные символы). На каждой шуге цикла значение переменной цикла, являющейся формальным параметром этого цикла последовательно принимает значения из перечня параметров. Пример использования директивы IRP приведен ниже:

```

IRP p, <AX, BX, CX, DX>
XOR p, p
ENDM

```

```

1      XOR AX,AX
1      XOR BX,BX
1      XOR CX,CX
1      XOR DX,DX

```

Жирным шрифтом текста показан код на макроязыке, а обычным макрорасширения. Другой пример иллюстрирует задание выражений в списке фактических параметров для директивы **IRP**. Здесь используются специальные символы и вспомогательные переменные этапа компиляции (**N** и **W4**), которым предварительно задаются значения ().

```

N = 3
W4 EQU 5
      IRP A, <N+1, %N + 1, W%N+1, %N , N >
      DW A
      ENDM

1      DW N+1
1      DW 4
1      DW W4
      2      DW 5

1      DW 3
1      DW N

```

Служебный символ “%” позволяет взять значение переменной в выражении, об этих символах мы поговорим ниже. На втором уровне выделена подстановка значения переменной этапа компиляции **W4**.

Директива **IRPC** имеет следующий синтаксис:

```

IRPC <переменная цикла>, <Строка символов >
<тело цикла>
ENDM

```

**<Строка символов > := <символ> | <Строка символов >**

Переменной цикла присваивается значение символа из строки и с этой переменной цикл выполняется. Число повторений цикла равно числу символов в строке. Пример использования циклического оператора:

```

IRPC D, 123456789
      V1&D DB D
ENDM

1      V11      DB 1
1      V12      DB 2
1      V13      DB 3
1      V14      DB 4
1      V15      DB 5
1      V16      DB 6
1      V17      DB 7
1      V18      DB 8
1      V19      DB 9

```

Формальный параметр “**D**” является переменной цикла. Для каждого символа из строки генерируется описание в виде байтовой переменной. В этом примере используется служебный символ макрокоманд “&”, который позволяет сливать имена переменных.

## 22.8. 22.8 Локальные метки и переменные макрокоманд

При разработке макрокоманд могут использоваться вспомогательные метки и переменные. Так как макрокоманда может быть вызвана несколько раз, то при совпадении имен будет выдана ошибка и компиляция остановится. Для автоматической индексации таких

имен используется специальный механизм локальных переменных. Для его использования внутри макроопределения задается директива макрокоманд LOCAL, которая имеет следующий синтаксис:

**LOCAL <локальное имя> {,< локальное имя > ... }**

В качестве локального имени могут использоваться локальные переменные или локальные метки. Компилятор заменяет эти имена служебными идентификаторами (??dddd, где d – десятичная цифра) и обеспечивает сквозную нумерацию во всем исходном модуле. Так исключается дублирование описаний. Покажем применение локальных меток и переменных на примере. Пусть имеется макроопределение:

```
print MACRO CH, CR
    LOCAL loc_per , loc_met
    MOV DL, CH
    MOV loc_per , DL
    CALL DISPL
    IFIDN <CR>,<PER>
    CALL crlf
    ENDIF
    JMP loc_met
loc_per db 0
loc_met:
    ENDM
```

В этом макроопределении объявлена локальная метка (**loc\_met**) и переменная (**loc\_per**). При вызове макрокоманды мы получим в листинге следующие макрорасширения:

```
print '5', PER
1      MOV DL, '5'
1      MOV ??0000 , DL
1      CALL DISPL
1      CALL crlf
1      JMP ??0001
1      ??0000 db 0
1      ??0001:
```

В следующем вызове макрокоманды print метки и переменные будут проиндексированы:

```
print '7', PER
1      MOV DL, '7'
1      MOV ??0002 , DL
1      CALL DISPL
1      CALL crlf
1      JMP ??0002
1      ??0002 db 0
1      ??0003:
```

Общее число локальных меток и переменных во всех макрорасширениях одного исходного модуля, очевидно, не может превышать 9999.

## 22.9. 22.8 Служебные символы макрокоманд

Служебные символы применяются для написания макрокоманд для следующих целей:

- “%” - данный символ позволяет получить значение числовой или текстовой переменной этапа компиляции и вставить его в макрорасширение. Он работает в макрокомандах и циклах этапа компиляции.

- “&” - позволяет склеить строку символов и переменную этапа компиляции или параметр макроопределения (или наоборот).
- “!” - используется в аргументе для указания Ассемблеру, что символ, следующий за ним, является литералом, а не именем (параметра, переменной этапа компиляции).
- “<” “>” – используется для задания текста, который передается в макрорасширения целиком, даже если он содержит пробелы и другие знаки операций.
- “;;” - используются для включения комментария в текст макроопределения. Эти комментарии не передаются в макрорасширения.

Рассмотрим примеры. Служебный символ “%”. Если переменная символьная:

```
L      EQU <'&text'>
      IRP D,<%L, L >
      DB D
      ENDM

1      DB '&text'
1      DB L
```

При использовании числовой переменной, получим:

```
L = 2
      IRP D,<%L, L >
      DB D
      ENDM

1      DB 2
1      DB L
```

Служебный символ “&”. Опишем макрокоманду для генерации макрокоманд MOVSB и MOVSW. Проверим ее вызовы двумя разными параметрами.

```
; Описание макрокоманды
      MOVE MACRO TAG
      REP  MOVSB&TAG
      ENDM

;; Вызов макрокоманды и расширение
      MOVE W
1      REP  MOVSW
      MOVE B
1      REP  MOVSB
```

Служебный символ “&” используется для слияния константы и формального параметра макрокоманды (TAG). Следующая макрокоманда иллюстрирует использование почти всех служебных символов (“&”, “%”, “%”).

```
testpr MACRO PAR1 , PAR2 , V
      LOCAL MET, MSG
      MOV AH , 09H
      MOV DX , OFFSET &PAR1
      INT 21H
      JMP MET
MSG    DB    '&PAR2 ---', '!&V ,&V , %V,!%V'
MET:
      ENDM
```

Параметры PAR1 и PAR2 вставляются в макрорасширение по значению. Пример вызова:

```
MSG    DB 'TEST$',10,13
...
TESTPR MSG , <Пример текста !%V> , 1
1      MOV AH , 09H
1      MOV DX , OFFSET MSG
```

```

1          INT 21H
1          JMP ??0000
1      ??0001  DB      'Пример текста %V ---', '!1 ,1 , %V , !%V'
```

В этом примере используются при вызове макрокоманды угловые скобки, для передачи в макрокоманду текста с несколькими пробелами. В примере видно, что параметры **&V** будет заменены (в примере на 1). Параметр в тексте **%V** должен в передаваемом тексте (PAR2) быть помечен знаком “!”, в противном случае будет отмечена ошибка (он будет интерпретирован как литерал). Значение **&PAR1** и просто **PAR1** в этом примере являются эквивалентными (проверьте!).

## 22.10. Вложенные вызовы макрокоманд

Из одних макрокоманд можно вызывать другие макрокоманды. Такие вызовы называются вложенными. Использование вложенных макрокоманд позволяет сделать настройки программы более гибкими и сократить размеры макрокоманд. Рассмотрим пример. Первая макрокоманда заполняет поле значением параметра.

```

; Макрокоманда выделения памяти с заполнением выражениями
FIELD MACRO VAR
DB &VAR
ENDM
```

Во второй макрокоманде организован цикл заполнения массива (NAME). Натуральными числами в порядке возрастания, начиная с базового значения (BASE), с определенным шагом (STEP) и определенного размера (SIZEM).

```

; Натуральные числа в порядке возрастания
MAS MACRO SIZEM , NAME , BASE , STEP
FCount = BASE
&NAME      DB BASE
REPT  SIZEM - 1
FIELD %FCount + STEP
FCount = FCount + 1
ENDM
ENDM
```

При вызове макрокоманды мы получим следующее расширение:

```

MAS 10 , T0 , 3 , 5
1  T0      DB 3
3      DB 8
3      DB 9
3      DB 10
3      DB 11
3      DB 12
3      DB 13
3      DB 14
3      DB 15
3      DB 16
```

Макрокоманда генерирует массив T0 размером в 10 элементов, начиная с 3 с шагом 5.

## 22.11. Условная компиляция

Директивы условной компиляции позволяют включать или не включать текст в исходную программу в зависимости от ряда условий. Перечень групп директив приведен ниже:

- IF1, IF2, ELSE, ENDIF – директивы условной компиляции (проходы компилятора).

- IF, IFE, ELSE, ENDIF – директивы условной компиляции (логическое условие).
- IFDEF, IFNDEF, ELSE, ENDIF – директивы условной компиляции (объявление переменных этапа компиляции).
- IFB, IFNB, ELSE, ENDIF – директивы условной компиляции (проверка наличия параметра макрокоманды).
- IFIDN, IFIDIF, ELSE, ENDIF – директивы условной компиляции (сравнение переменных этапа компиляции).

Рассмотрим кратко их возможности.

**Директивы IF1, IF2** включают текст в исходную программу, если выполняется первый (IF1) или второй (IF2) проход компилятора. Обычно это делается при подключении библиотек макрокоманд, так как повторное включение приводит к ошибкам. Например:

```
IF1
INCLUDE MACRO.LIB
ENDIF
```

Макробибблиотека MACRO.LIB будет включена только на первом проходе.

**Директивы IF, IFE, ELSE, ENDIF** включают текст в исходную программу, на основе логических условий этапа компиляции. Если условие выполнено (равно нулю для IFE), то текст в исходную программу включается. Если условие не выполнено (не равно нулю для IF), то текст в исходную программу включается. Конструкция ELSE позволяет задать альтернативу по этому условию. Синтаксис и семантика директив описаны ниже:

**IF <выражение>** – **IF TRUE** (если истина) Если значение выражения не нулевое, операторы условного блока ассемблируются.

**IFE <выражение>** – **IF FALSE** (если ложь) Если значение выражения равно нулю, операторы условного блока ассемблируются.

Значения истина и ложь в Ассемблере определяются так:

```
FALSE      (0000h)  - ложь
TRUE       (FFFFh)  - истина
```

В общем виде конструкции условных директив можно описать так:

```
IFXX <аргументы>
...
<директивы для истинного значения условия>
...
[ELSE]
...
<директивы для ложного значения условия>
...
ENDIF
```

Рассмотрим простые примеры условных директив:

```
IF N EQ 5
    print 'A', PER
ELSE
    print 'B', PER
ENDIF
```

Если переменная этапа компиляции N равна 5, то выполняется обращение к макрокоманде print с параметром 'A', в противном случае с параметром 'B'.

```
IFIDN <&CR>, <PER>
    CALL LFCR
ENDIF
```

Если параметр макрокоманды (CR) установлен в значение <PER>, то вызывается процедура перевода строки и возврата каретки.



```

IFNDEF COUNT
    COUNT = 0
ENDIF

```

Если переменная этапа компиляции COUNT не определена, то она определяется с начальным значением равным нулю.

```

IFB V
    JMP MET
ENDIF

```

Если параметр макрокоманды V не задан при вызове (BLANK), то генерируется команда безусловной передачи управления на метку.

Надеюсь, что более полную информацию по условной компиляции вы найдете в литературе, документации и справочниках по макроассемблеру. Практическое использование таких директив Вы должны продемонстрировать при выполнении 9-й ЛР.

## 22.12. Директива EXITM

Обычно работа макрокоманды завершается в тот момент, когда встретила последняя директива ENDM макроопределения. Существует и другая возможность завершить работу макрокоманды (как процедуры этапа компиляции) – это использование директивы EXITM. Если встретила такая директива, то дальнейший режим обработки макровызова завершается и Ассемблер переключается на обработку команды, следующей за данным макровыводом. В одном макроопределении может быть несколько выходов, которые выполняются директивой EXITM. Например, если после проверки параметра мы обнаруживаем ошибку, то можем сделать выход из макроопределения. Например:

```

...
IF <PAR> NE <TEST>
...
    JMP MET
ELSE
...
    EXITM
ENDIF
...

```

## 22.13. Отладка макрокоманд

Отладка макрокоманд является трудоемкой задачей, так как нет специальных отладчиков для этого. Результат работы макрокоманд можно увидеть только в листинге программы с включенным режимом макрорасширений (см. выше). По результатам макрорасширений можно судить о правильности их работы. При необходимости можно выдать сообщение на стандартный вывод (дисплей или окно ошибок) собственное пользовательское сообщение. Это выполняется директивой %OUT. Например:

```

%OUT Отладка макрокоманды print

```

Если такая директива встретится в макрокоманде, то будет напечатано это сообщение. Для инициации (форсирования) ошибки используется другая директива - .ERR. Например:

```

.ERR
%OUT Ошибка в макрокоманде print

```

В окно ошибок будет помещено это сообщение.

## 22.14. Сравнение макросов и процедур

По сравнению с процедурами макрокоманды имеют четыре преимущества:

1. Макрокоманды являются более гибкими по сравнению с процедурами. Они позволяют задавать параметры, пропускать их при вызове макрокоманд, выполнять условную компиляцию.
2. Макрокоманды выполняются с более высокой скоростью (более эффективной), так как не тратится время на команды вызова, а все действия по настройке выполняются на этапе компиляции.
3. Макрокоманды легко упаковать в библиотеку и подключать их в программу.
4. Программа, построенная на основе макрокоманд, является более наглядной и понятной.

Недостатками макрокоманд по сравнению с процедурами является:

1. Программа, построенная на основе макрокоманд, имеет больший размер, так как при каждом вызове макрокоманды выполняется макрорасширение. В некоторых случаях это может стать существенным недостатком.
2. Макрокоманды труднее отлаживать, так как трудно создать специальный отладчик, так как все выполняется на этапе компиляции.

В целом, можно сделать вывод, что если рационально использовать комбинацию механизмов макрокоманд и процедур, то можно получить эффективный и наглядный код даже на языке Ассемблера.

## 22.15. Макрокоманды – процедуры этапа компиляции

Для тех, кто хорошо освоил механизм макрокоманд, приходит понимание того, что механизм макрокоманд и условной компиляции программ очень мощный и эффективный механизм для создания программных систем. Главное понять, что макрокоманды, по сути, являются процедурами этапа компиляции. Так как существует возможность вложенного вызова макрокоманд, возможность определения переменных этапа компиляции и использования библиотек, то возникает возможность описания собственного языка специального назначения. Действительно существуют специализированные системы программирования, целиком построенные на основе макроассемблера. Например, система имитационного моделирования GPSS в первых своих версиях была реализована по этой схеме. Важно, что трудозатраты на такие системы значительно ниже, чем при создании системы программирования с нуля.

## 22.16. Примеры программы с макросами

В следующем тексте работающей программы с макросами, которые я привожу без пояснений, вы найдете много полезных фрагментов для выполнения 9-й ЛР.

```
mycode segment 'code'  
    assume cs:mycode, ds:mycode  
    assume ss: stseg  
; Макрокоманда выделения памяти с заполнением выражениями
```

```

FCount = 1
Fparam = 1
FIELD MACRO VAR
    DB &VAR
ENDM

;;
;;; Натуральные числа в порядке возрастания
MAS MACRO SIZEM , NAME , BASE , STEP
    FCount = BASE
    &NAME    DB BASE
    REPT    SIZEM - 1
    FIELD %FCount + STEP
    FCount = FCount + 1
    ENDM
ENDM

;;; Натуральные числа в порядке убывания
MASM MACRO SIZEM , NAME , BASE , STEP
    FCount = BASE
    &NAME    DB BASE
    REPT    SIZEM - 1
    FIELD %FCount - STEP
    FCount = FCount - 1
    ENDM
ENDM

; Арифметическая прогрессия
MASN MACRO SIZEM , NAME , BASE , STEP
    FCount = BASE
    &NAME    DB BASE
    REPT    SIZEM - 1
    FIELD %FCount * STEP
    FCount = FCount + 1
    ENDM
ENDM

; Геометрическая прогрессия
MASP MACRO SIZEM , NAME , BASE , STEP
    FCount = 1
    &NAME    DB STEP + BASE
    REPT    SIZEM - 1
    FParam = 1
    REPT Fcount + 1
    FParam = Fparam * STEP
    ENDM
    FIELD %FParam + BASE
    FCount = FCount + 1
    ENDM
ENDM

;
SUM MACRO NAME , SIZE , VSUM
    LOCAL VM
    LOCAL VERR
    PUSH CX

```

```

        PUSH AX
        PUSH DX
        PUSH SI
        MOV CX , SIZE
        CMP CX , 0
        JB VERR
        XOR AX , AX
        MOV SI,0
        XOR DX , DX
VM:     MOV DL , NAME[SI]
        ADD AX, DX
        INC SI
        LOOP VM
        MOV VSUM , AX
        MOV DX, AX
        CALL PRINTDEC
VERR:   POP SI
        POP DX
        POP AX
        POP CX
        ENDM

;Макрокоманда для произведения
PRSTR MACRO NAME , SIZE , KSTR
        LOCAL MC
        LOCAL MC1
        LOCAL MFIN
        LOCAL MPROD
        PUSH CX
        PUSH SI
;; НУ цикла
;;   Перевод строки
        mov dl, 0Ah
        mov ah, 2
        int 21h
        mov dl, 0Dh
        mov ah, 2
        int 21h
;;
        MOV CX , SIZE
        MOV SI , 0
;; Цикл по массиву
MC:     PUSH CX
        MOV CX , KSTR
;; Цикл по строке
MC1:    MOV DL, NAME[SI]
        CALL PRINTB
        INC SI
        CMP SI , SIZE
        JNE MPROD
;; Последний перевод
        mov dl, 0Ah
        mov ah, 2

```

```

        int 21h
        mov dl, 0Dh
        mov ah, 2
        int 21h
        JMP MFIN
MPROD:
        LOOP MC1
        POP CX
;; Перевод строки
        mov dl, 0Ah
        mov ah, 2
        int 21h
        mov dl, 0Dh
        mov ah, 2
        int 21h
        LOOP MC
;; Конец Цикла
MFIN:
        POP SI
        POP CX
        ENDM

;Макрокоманда печати массива в десятичном виде по столбцам
PRCOL MACRO NAME , SIZE , KCOL
        LOCAL MC
        LOCAL MC1
        LOCAL MFIN
        LOCAL MPROD
        LOCAL TEMP
        LOCAL OBXOD
        LOCAL MTEMP
        PUSH CX
        PUSH SI
;; НУ цикла
;; Перевод строки
        mov dl, 0Ah
        mov ah, 2
        int 21h
        mov dl, 0Dh
        mov ah, 2
        int 21h
;; Определение числа строк
        MOV CX , SIZE/KCOL
        MOV SI , SIZE MOD KCOL
        CMP SI , 0
        JE MTEMP
        ADD CX, 1
MTEMP: MOV TEMP , CX
;;
        MOV SI , 0
;; Цикл по массиву
MC:    PUSH CX
        MOV CX , KCOL

```

```

        PUSH SI
;; Цикл по строке
MC1:    MOV DL, NAME[SI]
        CMP SI , SIZE
        JGE MPROD
        CALL PRINTB
        ADD SI , TEMP
        LOOP MC1

;;
MPROD:  POP SI
        INC SI
        POP CX
;;      Перевод строки
        mov dl, 0Ah
        mov ah, 2
        int 21h
        mov dl, 0Dh
        mov ah, 2
        int 21h
        LOOP MC
;;      Конец Цикла
MFIN:
        POP SI
        POP CX
        JMP OBXOD

TEMP    DW 0
OBXOD:
        ENDM

;
main proc
;Занесение регистра DS
        PUSH CS
        POP DS
;;;;;;;;;;;;;
        JMP MEND1
; Описания массивов
.XALL
        MAS 10 , T0 , 3 , 5
.SALL
        MASM 5 , T1 , 100 , 1
        MASN 5 , T2 , 1 , 5
.XALL
        MASP 5 , T3 , 10 , 3
.SALL
        PRSTR T0 , 5, 3
;   PRCOL T0 , 5, 3
; выход с ожиданием
VAR     DW 0
;   Запрос символа с клавиатуры
; Суммирование

MEND1:

```

```

        SUM T0 , 10 , VAR
        MOV DX , VAR
        CALL printdec
        MOV DL , 131
        CALL printb
        PRSTR T0 , 20, 5
;       PRCOL T0 , 27, 6
        mov ah, 08h
        int 21h
        mov al, 0
;   Выход в ДОС
        mov ah, 4ch
        int 21h
main endp
;-----
;::::::::::::::::::;
;   Печать числа
;   DX - число
printdec proc
;   Перевод числа
;   10000
        PUSH CX
        PUSH SI
        PUSH AX
;   Начальные условия цикла перевода
        MOV SI , 0
        MOV CX , 4
;   DX:AX - делимое
        MOV DDWORK , 10000
;   Цикл перевода
CICLE:
        MOV AX, DX
        MOV DX , 0
        DIV DDWORK
;   AX - остаток , DX - частное
        XCHG DX , AX
;   Коррекция для вывода символа
        ADD dl , 30h
        CMP dl , 30h
        JNE MC1
        MOV dl , 20h
MC1:      MOV CHI[SI] , dl
        INC SI
;   DX - остаток , AX - частное
        XCHG DX , AX
;   Новый делитель
        PUSH DX
        PUSH AX
        MOV AX , DDWORK
        MOV DX , 0
        DIV D10
        MOV DDWORK , AX

```

```

        POP AX
        POP DX
        LOOP CICLE
;       На конец цикла
; Коррекция для вывода символа
        ADD dl , 30h
        MOV CH[SI] , dl
; Вывод строки
        MOV AH , 09h
        MOV DX , offset STR
        int 021h
        POP AX
        POP SI
        POP CX
        ret
printdec endp
; Перевод десятичное и печать байтовой переменной
; без перевода строки    DL - байт
printb proc
        PUSH CX
        PUSH SI
        PUSH AX
; Начальные условия цикла перевода
        MOV SI , 0
        MOV CX , 2
; DX:AX - делимое
        MOV DDWORK , 100
        MOV DH , 0
; Цикл перевода
CICLE1:
        MOV AX, DX
        MOV DX , 0
        DIV DDWORK
; AX - остаток , DX - частное
        XCHG DX , AX
; Коррекция для вывода символа
        ADD dl , 30h
        CMP dl , 30h
        JNE MC11
        MOV dl , 20h
MC11:    MOV CHIB[SI] , dl
        INC SI
; DX - остаток , AX - частное
        XCHG DX , AX
; Новый делитель
        PUSH DX
        PUSH AX
        MOV AX , DDWORK
        MOV DX , 0
        DIV D10
        MOV DDWORK , AX
        POP AX

```



```

        POP DX
        LOOP CICLE1
;       На конец цикла
; Коррекция для вывода символа
        ADD dl , 30h
        MOV CHIB[SI] , dl
; Вывод строки
        MOV AH , 09h
        MOV DX , offset CHIB
        int 021h
        POP AX
        POP SI
        POP CX
        ret
printb endp
;;
; - mas - size - kcol
;
DDWORK DW 0
D10     DW 10
DD10    DW 10000
DD11    DW 1000
DD12    DW 100
DD13    DW 10
STR      DB 'Результат = '
CHI      DB '      ', 10, 13 , '$'
CHIB     DB '      ', ' ' , '$'
;;;;;;;;;;
mycode ends
;-----
stseg segment stack 'stack'
        dw 256 dup(0)
stseg ends
;-----
        end main

```

## 23. Коды их назначение и виды

Кодировка русских символов различается в разных режимах работы: работа в среде WINDOWS и работы в среде эмуляции ДОО. Так исторически сложилось, что эти кодировки не идентичны, а для нормального представления символов на экране приходится выполнять операции перекодирования. Кодировка в среде ДОО носит название ASCII (American Standard Code for Information Interchange – американский стандартный код для обмена информацией) в первоначальном варианте содержала всего 128 кодов (см. первую таблицу ниже). Затем появилась расширенная кодировка ASCII, которая была рассчитана уже на 256 символов (см. вторую таблицу ниже). В этой кодировке уже можно было представить символы других языков и символы псевдографики. В современных русифицированных программах эту кодировку Вы можете встретить также под названием “Кириллица ДОО/ Кириллица ДОО 2”. Кодировка в среде WINDOWS, в первую очередь вторая часть таблицы (128 - 255), имеет другое содержание и название - ANSI. Другое название ANSI – кодировка WINDOWS. В этой системе кодирования не используются символы псевдографики, так как вывод информации на экран выполняется в графическом режиме. Позиции русских символов находятся во второй половине таблицы (см. таблицы ниже) и имеют отличную кодировку от кода ASCII. Это создает много проблем при совместной работе в WINDOWS и режиме командной строки.

Состояние кодировки влияет на следующие операции:

- Вывод информации на экран дисплея, если информация не печатается по-русски, это означает, что кодировка вводимого символа или файла не совпадает с кодировкой вывода информации.
- Ввод информации с клавиатуры, вводимые символы отображаются не в том виде, который ожидается программистом или пользователем.
- Взаимный перевод информации из одной кодировки в другую (чаще всего это файлы). Исходная и результирующая кодировка символов должны быть заданы правильно.

**Примечание.** Рекомендую Вам детально разобраться с кодировкой русских символов при разработке отчетов по лабораторным работам. Если Вы пойдете по “простому пути” и ограничитесь только латинскими буквами, то это будет неверно, и такой отчет по ЛР не будет считаться правильным.

Таким образом, для корректной работы в среде двух кодировок необходимо обеспечить корректный ввод и отображение информации. Это обеспечивается специальными программами – русификаторами, которые должны быть предварительно запущены перед основной работой. Русификатор представляет собой драйвер (резидентную программу), который будет работать при каждом нажатии клавиши на клавиатуре и при выводе символов на экран дисплея. Чаще всего такие драйверы совмещают две основные функции: обслуживание клавиатуры и обеспечение корректного вывода на экран дисплея. При выполнении ввода русских символов нужно переключиться в режим русской раскладки (обычно с помощью “горячей” клавиши). Никаких дополнительных действий после запуска драйвера, для корректного вывода на экран, предпринимать не нужно, он сработает автоматически. На сайте представлен драйвер RKM, который работает устойчиво и, по умолчанию, обеспечивает переключение раскладок клавиатуры с помощью клавиши “правый Shift”. Могут быть заданы и другие настройки.

Ниже мы кратко рассмотрим кодировки символов для разных режимов.

## 23.1. ASCII

Таблица кодировки ASCII, первая ее половина показана на рисунке расположенном ниже. Эта табличка получена в среде QC25 в ее справочной системе. Эта первая часть таблицы (0-127) является общей для всех кодировок и для разных кодовых страниц.

ASCII коды в диапазоне 0 - 127															
0	(nul)	16	▸ (dle)	32	(sp)	48	0	64	@	80	P	96	`	112	p
1	☉ (soh)	17	◀ (dc1)	33	! (sp)	49	1	65	A	81	Q	97	a	113	q
2	☐ (stx)	18	⬆ (dc2)	34	"	50	2	66	B	82	R	98	b	114	r
3	♥ (etx)	19	!! (dc3)	35	#	51	3	67	C	83	S	99	c	115	s
4	♦ (eot)	20	¶ (dc4)	36	\$	52	4	68	D	84	T	100	d	116	t
5	♠ (eng)	21	⌘ (nak)	37	%	53	5	69	E	85	U	101	e	117	u
6	♣ (ack)	22	— (syn)	38	&	54	6	70	F	86	V	102	f	118	v
7	• (bel)	23	‡ (eth)	39	'	55	7	71	G	87	W	103	g	119	w
8	▣ (bs)	24	↑ (can)	40	(	56	8	72	H	88	X	104	h	120	x
9	(tab)	25	↓ (em)	41	)	57	9	73	I	89	Y	105	i	121	y
10	(lf)	26	→ (eof)	42	*	58	:	74	J	90	Z	106	j	122	z
11	♂ (vt)	27	← (esc)	43	+	59	;	75	K	91	[	107	k	123	{
12	♀ (np)	28	⌞ (fs)	44	,	60	<	76	L	92	\	108	l	124	
13	(cr)	29	↗ (gs)	45	-	61	=	77	M	93	]	109	m	125	}
14	☾ (so)	30	▲ (rs)	46	.	62	>	78	N	94	^	110	n	126	~
15	✱ (si)	31	▼ (us)	47	/	63	?	79	O	95	_	111	o	127	Δ

Первая группа символов в этой таблице (0-32 или 0h – 020h) является группой служебных символов, которые используются для управления. Они обычно не отображаются на экране (например, символ ESC – код 27). Несмотря на это, в программе они могут быть использованы. Например, символ с кодом 7 (bel) может быть использован для выдачи звукового сигнала при его передачи на дисплей в текстовом режиме – при его выводе срабатывает стандартный динамик компьютера (если ООП не отключен!). Символ с кодом 13 (cr) является символом возврата каретки или конца строки, а символ с кодом 26 (eof) стандартным символом конца файла. В литературе и справочниках Вы найдете детальную расшифровку служебных символов и способов их использования.

Вторая часть таблицы ASCII показана ниже. Обратите внимание на кодировку русских букв и символов псевдографики.

ASCII коды в диапазоне 128 - 255							
128 А	144 Р	160 а	176 ▯	192 L	208 Ш	224 р	240 Ё
129 Б	145 С	161 б	177 ▯	193 1	209 Т	225 с	241 ё
130 В	146 Т	162 в	178 ▯	194 T	210 П	226 т	242 С
131 Г	147 У	163 г	179 ▯	195 T	211 П	227 у	243 с
132 Д	148 Ф	164 д	180 ▯	196 -	212 Ё	228 ф	244 Ъ
133 Е	149 Х	165 е	181 ▯	197 +	213 F	229 х	245 ъ
134 Ж	150 Ц	166 ж	182 ▯	198 F	214 П	230 ц	246 ъ
135 З	151 Ч	167 з	183 п	199 ▯	215 П	231 ч	247 ъ
136 И	152 Ш	168 и	184 э	200 ▯	216 T	232 ш	248 ъ
137 Й	153 Щ	169 й	185 ▯	201 ▯	217 T	233 щ	249 ъ
138 К	154 Ъ	170 к	186 ▯	202 ▯	218 Г	234 ъ	250 ъ
139 Л	155 Ы	171 л	187 ▯	203 ▯	219 ▯	235 ы	251 ъ
140 М	156 Ь	172 м	188 ▯	204 ▯	220 ▯	236 ь	252 ъ
141 Н	157 Э	173 н	189 ▯	205 =	221 ▯	237 э	253 ъ
142 О	158 Ю	174 о	190 ▯	206 ▯	222 ▯	238 ю	254 ъ
143 П	159 Я	175 п	191 1	207 ▯	223 ▯	239 я	255 ъ

Символы псевдографики могут быть использованы в программах лабораторных работ для выполнения основных и дополнительных требований. Так в 4-й ЛР в качестве дополнительных требований предлагается поместить таблицу символов в рамку. Эту рамку можно сделать с помощью символов с кодами: 179, 191, 192, 217 и 218. Эти символы нужно будет вывести на экран в определенной последовательности.

## 23.2. Кодировка ANSI

Первая часть таблицы кодировки WINDOWS (ANSI) показана ниже (0-127). Вы можете убедиться в том, что она полностью совпадает с таблицей кодировки ASCII. Можете сами в этом убедиться. В этой таблице три колонки: символ (SYM), десятичный (DEC) и шестнадцатеричный (HEX) коды. В каждой строке приводятся значения 4-х символов.

SYM	DEC	HEX	SYM	DEC	HEX	SYM	DEC	HEX	SYM	DEC	HEX
	32	20	!	33	21	"	34	22	#	35	23
\$	36	24	%	37	25	&	38	26	'	39	27
(	40	28	)	41	29	*	42	2A	+	43	2B
,	44	2C	-	45	2D	.	46	2E	/	47	2F
0	48	30	1	49	31	2	50	32	3	51	33
4	52	34	5	53	35	6	54	36	7	55	37
8	56	38	9	57	39	:	58	3A	;	59	3B
<	60	3C	=	61	3D	>	62	3E	?	63	3F
@	64	40	A	65	41	B	66	42	C	67	43
D	68	44	E	69	45	F	70	46	G	71	47
H	72	48	I	73	49	J	74	4A	K	75	4B
L	76	4C	M	77	4D	N	78	4E	O	79	4F
P	80	50	Q	81	51	R	82	52	S	83	53
T	84	54	U	85	55	V	86	56	W	87	57
X	88	58	Y	89	59	Z	90	5A	[	91	5B
\	92	5C	]	93	5D	^	94	5E	_	95	5F
`	96	60	a	97	61	b	98	62	c	99	63
d	100	64	e	101	65	f	102	66	g	103	67
h	104	68	i	105	69	j	106	6A	k	107	6B
l	108	6C	m	109	6D	n	110	6E	o	111	6F
p	112	70	q	113	71	r	114	72	s	115	73
t	116	74	u	117	75	v	118	76	w	119	77
x	120	78	y	121	79	z	122	7A	{	123	7B
	124	7C	}	125	7D	~	126	7E	•	127	7F

Вторая часть таблицы кодировки WINDOWS (ANSI) показана ниже (128-255). Структура таблицы аналогична. Обратите внимание на кодировку русских букв, она существенно отличается от кодировки ASCII.

Ъ	128	80	Ѓ	129	81	/	130	82	ѓ	131	83
„	132	84	…	133	85	†	134	86	‡	135	87
€	136	88	%	137	89	љ	138	8A	<	139	8B
њ	140	8C	ќ	141	8D	ћ	142	8E	џ	143	8F
ђ	144	90	`	145	91	'	146	92	“	147	93
”	148	94	•	149	95	—	150	96	—	151	97
□	152	98	™	153	99	љ	154	9A	>	155	9B
њ	156	9C	ќ	157	9D	ћ	158	9E	џ	159	9F
	160	A0	Ў	161	A1	ў	162	A2	J	163	A3
џ	164	A4	Ѓ	165	A5	ђ	166	A6	Ѓ	167	A7
Ѓ	168	A8	©	169	A9	€	170	AA	«	171	AB
¬	172	AC	–	173	AD	®	174	AE	і	175	AF
°	176	B0	±	177	B1	І	178	B2	і	179	B3
ѓ	180	B4	µ	181	B5	¶	182	B6	·	183	B7
ё	184	B8	№	185	B9	е	186	BA	»	187	BB
ј	188	BC	S	189	BD	s	190	BE	і	191	BF

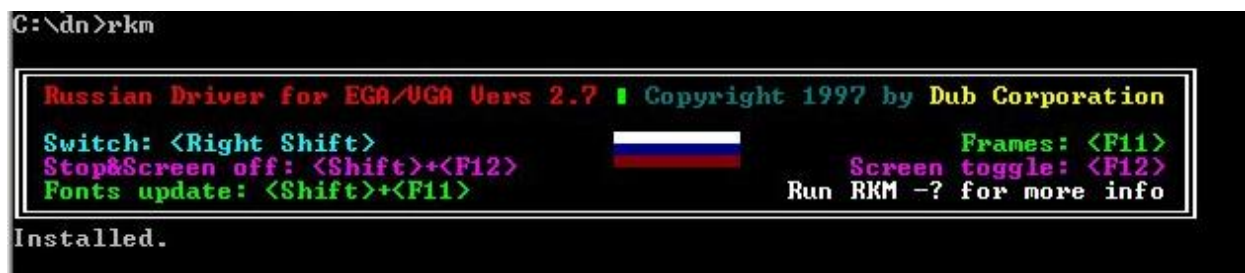
А	192	С0	Б	193	С1	В	194	С2	Г	195	С3
Д	196	С4	Е	197	С5	Ж	198	С6	З	199	С7
И	200	С8	Й	201	С9	К	202	СA	Л	203	СB
М	204	СC	Н	205	СD	О	206	СE	П	207	СF
Р	208	Д0	С	209	Д1	Т	210	Д2	У	211	Д3
Ф	212	Д4	Х	213	Д5	Ц	214	Д6	Ч	215	Д7
Ш	216	Д8	Щ	217	Д9	Ъ	218	ДA	Ы	219	ДB
Ь	220	DC	Э	221	DD	Ю	222	DE	Я	223	DF
а	224	Е0	б	225	Е1	в	226	Е2	г	227	Е3
д	228	Е4	е	229	Е5	ж	230	Е6	з	231	Е7
и	232	Е8	й	233	Е9	к	234	ЕA	л	235	ЕB
м	236	ЕC	н	237	ЕD	о	238	ЕE	п	239	ЕF
р	240	F0	с	241	F1	т	242	F2	у	243	F3
ф	244	F4	х	245	F5	ц	246	F6	ч	247	F7
ш	248	F8	щ	249	F9	ъ	250	FA	ы	251	FB
ь	252	FC	э	253	FD	ю	254	FE	я	255	FF

### 23.3. Русификаторы

После запуска командной строки необходимо запустить русификатор, в нашем случае это RKM. Запуск выполняется так:

```
>RKM.COM ↵
```

Окно запуска русификатора RKM и сообщение об его успешном запуске показано



ниже:

Для получения более подробной информации и настройки его можно вызвать в режиме справки:

```
>RKM.COM -? ↵
```

Снятие русификатора выполняется вместе с завершением работы в режиме командной строки. Кроме этого выгрузка может быть выполнена так:

```
>RKM.COM -U ↵
```

Для русификации Вы можете использовать и другие русификаторы, однако перед их использованием желательно детально разобраться в возможностях, в параметрах программы и ограничениях применения.

### 23.4. Перекодировка символов

Лабораторные работы по курсу выполняется в режиме командной строки, в файловом менеджере и в операционной системе MS DOS. В этом случае используется кодировка ASCII. Текстовые константы разрабатываемых программ и комментарии в них должны вводиться в этой кодировке. При оформлении отчетов использовать среду WINDOWS и текстовые редакторы в ней (MS WORD). Эти текстовые редакторы работают в кодировке ANSI. Поэтому в процессе работ необходимо выполнять перекодировки типа:

- ANSI => ASCII и
- ASCII => ANSI.

Такие операции могут быть выполнены следующим образом:



– С использованием специального текстового редактора – ASN\_ED.EXE, в котором предусмотрены эти операции преобразования (он есть на сайте). Для выполнения перекодировки нужно: загрузить файл и с помощью меню выполнить операции: “Edit” => “Convert ANSI =>ASCII” или “Edit” => “Convert ASCII =>ANSI”.

– Воспользоваться специальной простой программой перекодировки – TRANS.EXE (есть на сайте).

– Разработать самостоятельно для себя программу перекодировки и использовать ее для работы. Этот вариант предлагаю сделать самостоятельно для практики программирования. Кодировка символов для этого была уже рассмотрена.

При работе с командной TRANS.EXE нужно руководствоваться следующим синтаксисом командной строки:

<Запуск TRANS >:= TRANS.EXE\_</?> | TRANS.EXE\_<Режим>\_<Исходный файл>\_<Результирующий файл>

Где:

TRANS.EXE – имя программы перекодировки.

“\_” - символ пробела,

</?> - параметр выдачи справки,

<Режим>:= DW | WD – способ перекодировки: DW - ASCII =>ANSI, а WD - ANSI =>ASCII.

<Исходный файл> и <Результирующий файл> - стандартные текстовые файлы операционной системы, имена файлов могут совпадать.

Пример запуска перекодировки файла test.txt из DOC в WINDOWS (ASCII =>ANSI):

```
>TRANS.EXE DW test.txt test.win ↵
```

В результате получим перекодированный файл test.win. для обратной перекодировки нужно вызвать программу так:

```
>TRANS.EXE WD firstw.asm firstd.asm ↵
```

Для вызова справки о работе программы нужно задать:

```
>TRANS.EXE /? ↵
```

## 23.5. SCAN – коды

Кодировка вводимых символов, в первую очередь отображаемых на экране, представлена во множествах ASCII и ANSI. Однако этого недостаточно для обработки в программе сигналов от клавиатуры. Например, необходимо знать была ли нажата клавиша CTRL совместно с другой клавишей, или проверить была ли нажата клавиша NUMLOCK или любая клавиша на дополнительной части клавиатуры. Фактически нужно знать номер (!!!) нажатой клавиши. Термин скан-код (Scan Code) соответствует номеру нажатой клавиши, причем этот номер может соответствовать нескольким разным символам. Так, например, скан-код 08 (см. колонку Code в таблице, приведенной ниже) соответствует символам “\*” и “8” (если учесть русификацию то символов может быть больше). С помощью клавиатуры в 101 символ приходится кодировать 256 символов и служебных управляющих сигналов, определяющих управление вводом с клавиатуры. Скан-коды могут быть прочитаны в программу и использованы для работы.

Таблица основных скан-кодов клавиатуры представлена ниже:

Key	Code	Key	Code	Key	Code	Key	Code
ESC	01	U	16	or \	2B	F6	40
! or 1	02	I	17	Z	2C	F7	41
@ or 2	03	O	18	X	2D	F8	42
# or 3	04	P	19	C	2E	F9	43
\$ or 4	05	{ or [	1A	V	2F	F10	44

% or 5	06	} or ]	1B	B	30	NUMLOCK	45
^ or 6	07	ENTER	1C	N	31	SCROLL LOCK	46
& or 7	08	CTRL	1D	M	32	HOME or 7	47
* or 8	09	A	1E	< or ,	33	UP or 8	48
( or 9	0A	S	1F	> or .	34	PGUP or 9	49
) or 0	0B	D	20	? or /	35	-	4A
_ or -	0C	F	21	RIGHT SHIFT	36	LEFT or 4	4B
+ or =	0D	G	22	PRTSC or *	37	5	4C
LEFT	0E	H	23	ALT	38	RIGHT or 6	4D
TAB	0F	J	24	SPACEBAR	39	+	4E
Q	10	K	25	CAPSLOCK	3A	END or 1	4F
W	11	L	26	F1	3B	DOWN or 2	50
E	12	: or ;	27	F2	3C	PGDN or 3	51
R	13	" or '	28	F3	3D	INS or 0	52
T	14	or `	29	F4	3E	DEL or .	53
Y	15	LEFT SHIFT	2A	F5	3F		

Хотя для кодировки вводимых символов достаточно одного байта, при вводе с клавиатуры в программу может быть прочитано 2 байта информации. Логика ввода информации с клавиатуры такова:

- При выполнении специальных операций ввода (например, функции `getch()` в СИ) первый байт либо содержит код вводимого с клавиатуры (например, “А” – код 65), либо специальное значение, обозначающее использование режима ввода скан-кода.
- Специальное значение может быть либо 0, либо 0xE0h. Величина 0 означает, что получен скан-код и его значение находится во втором байте, который должен быть считан дополнительно. Скан-код позволяет однозначно определить комбинацию клавиш нажатых одновременно (Например, для ALT+V – скан-код = 47). Значение 0xE0h также сигнализирует о режиме ввода скан-кодов, и дополнительно указывает, что используется клавиши 101 символьной клавиатуры дублирующие основные клавиши.

Список распространенных расширенных скан-кодов приведен в фрагменте программы на СИ, описывающей перечисление (`enum EXTENDED`) с константами скан – кодов.

```
enum EXTENDED
{
    extINTRO = 0,          // Для всех клавиатур
    extINTRO2 = 0xE0,      // For keypad and other keys unique to
                          // 101-key keyboard

    /* Второй байт содержит расширенные скан коды : */

    /* Цифровая клавиатура */
    extUP = 72,  extDOWN = 80,  extLEFT = 75,  extRIGHT = 77,
    extPGUP = 73,  extPGDN = 81,  extHOME = 71,  extEND = 79,
    extINS = 82,  extDEL = 83,

    extCTRL_PRTSC = 114,
    extCTRL_LEFT = 115,  extCTRL_RIGHT = 116,
    extCTRL_PGUP = 132,  extCTRL_PGDN = 118,
    extCTRL_HOME = 119,  extCTRL_END = 117,

    NullKey = 3,  extSH_Tab = 15,

    /* ALT+буква */
    extALT_A = 30,  extALT_B = 48,  extALT_C = 46,  extALT_D = 32,
```

```

extALT_E = 18, extALT_F = 33, extALT_G = 34, extALT_H = 35,
extALT_I = 23, extALT_J = 36, extALT_K = 37, extALT_L = 38,
extALT_M = 50, extALT_N = 49, extALT_O = 24, extALT_P = 25,
extALT_Q = 16, extALT_R = 19, extALT_S = 31, extALT_T = 20,
extALT_U = 22, extALT_V = 47, extALT_W = 17, extALT_X = 45,
extALT_Y = 21, extALT_Z = 44,

/* extALT+цифровая клавиатура */
extALT_1 = 120, extALT_2, extALT_3, extALT_4, extALT_5,
extALT_6, extALT_7, extALT_8, extALT_9, extALT_0,

extALT_minus = 130, extALT_equals,

/* функциональная клавиша */
extF1 = 59, extF2, extF3, extF4, extF5,
extF6, extF7, extF8, extF9, extF10,
extF11 = 133, extF12,

/* SHIFT+функциональная клавиша */
extSH_F1 = 84, extSH_F2, extSH_F3, extSH_F4, extSH_F5,
extSH_F6, extSH_F7, extSH_F8, extSH_F9, extSH_F10,
extSH_F11 = 137, extSH_F12,

/* CTRL+ функциональная клавиша */
extCTRL_F1 = 94, extCTRL_F2, extCTRL_F3, extCTRL_F4,
extCTRL_F5, extCTRL_F6, extCTRL_F7, extCTRL_F8,
extCTRL_F9, extCTRL_F10, extCTRL_F11 = 137,
extCTRL_F12,

/* ALT+ функциональная клавиша */
extALT_F1 = 104, extALT_F2, extALT_F3, extALT_F4, extALT_F5,
extALT_F6, extALT_F7, extALT_F8, extALT_F9, extALT_F10,
extALT_F11 = 139, extALT_F12,
};

```

В справочниках и в литературе вы найдете полный перечень расширенных скан – кодов. Кроме того, вы можете воспользоваться программкой, которая приведена в конце данного раздела для получения таблиц кодов самостоятельно.

## 23.6. 23.6 Кодировка UNICODE

Большинство современных программ позволяет работать в специальной кодировке UNICODE. В этой кодировке символы кодируются 2-мя байтами, поэтому возможно закодировать 65536 символов. Такой способ кодирования позволяет таблицы кодов для разных стран и для разных иностранных языков. Это в свою очередь позволяет хранить информацию в файлах и БД на разных языках, что делает программные продукты более универсальными.

Здесь мы не будем приводить особенности работы с кодировкой UNICODE, так как для выполнения ЛР и курсовой работы в этом нет необходимости. Таблицы UNICODE очень большие, поэтому для получения информации о кодах символов удобнее воспользоваться специальной программой **charmap.exe**, которую Вы легко найдете в операционной системе. Эту программу достаточно вызвать из главного меню WINDOWS: “Пуск/Start” => “Выполнить/Run” => “**charmap.exe**”.



