Министерство образования Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра компьютерных систем в управлении и проектировании (КСУП)

В.В. Одиноков

ЭВМ и вычислительные системы

Однопрограммные системы

Учебное пособие

2002

Одиноков В.В.
ЭВМ и вычислительные системы. Однопрограммные системы: Учебное пособие. – Томск: Томский межвузовский центр дистанционного образования, 2002. – 133 с.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данное пособие предназначено для обучения студентов специальности «Информатика и управление в технических системах» по односеместровой дисциплине «Электронно-вычислительные машины и вычислительные системы». Основной целью данного курса является получение студентами знаний и навыков по использованию и построению системных управляющих программ для однопрограммных вычислительных систем.

Вычислительной системой (**ВС**) называется система, состоящая из аппаратных и программных средств, предназначенная для выполнения некоторого множества задач по переработке информации. При отсутствии таких задач ни сама ВС, ни её подсистемы не нужны.

Каждая задача, решаемая ВС, имеет алгоритм решения. ***Алгоритм*** – правило, определяющее последовательность действий над исходными данными, приводящую к получению искомых результатов. Форма представления алгоритма решения задачи, ориентированная на машинную реализацию, называется ***прикладной программой***.

Обязательной подсистемой любой ВС является аппаратное обеспечение, называемое обычно ***аппаратурой***. Сюда входят центральный процессор (ЦП), выполняющий машинные команды, из которых состоит любая машинная программа (в том числе и прикладная), а также оперативная память (ОП), предназначенная для хранения программ и обрабатываемых ими данных. Кроме того, любая ВС имеет периферийные аппаратные устройства: устройства ввода-вывода (например, дисплей и клавиатуру) и устройства внешней памяти (например, винчестер).

Несмотря на то, что ВС в составе двух названных подсистем (аппаратура и прикладные программы) в принципе пригодна для решения задач по переработке информации, практическое применение такой системы ограничивается лишь простейшими задачами. Это обусловлено тем, что разработка даже простой (в смысле решаемой задачи) прикладной программы на “голой” аппаратуре представляет собой очень трудоёмкий процесс. Уменьшение трудоёмкости прикладного программирования возможно по следующим направлениям:

1) изменение среды выполнения прикладной программы, приводящее к её упрощению;

2) предоставление разработчику прикладной программы возможности разрабатывать не реальную, а виртуальную (кажущуюся) программу.

Реализация перечисленных направлений обеспечивается за счет включения в ВС системных программ. Кроме того, применение системных программ позволяет создавать мультипрограммные ВС. В отличие от ***однопрограммной*** **ВС**, позволяющей выполнять прикладные программы только по одной (каждая следующая программа ждет завершения предыдущей), ***мультипрограммная*** **ВС** выполняет одновременно несколько

прикладных программ. Реализация такой системы требует значительных усилий на обеспечение защиты данных одной прикладной программы от воздействия других прикладных программ. Кроме того, требуется выполнять распределение ресурсов ВС между прикладными программами.

В настоящем пособии рассматриваются вопросы организации однопрограммных ВС. Особое внимание в ходе этого рассмотрения уделяется интерфейсам между всеми тремя подсистемами ВС – прикладными программами, системными программами, аппаратурой. Значительное место в пособии занимают также принципы организации аппаратуры и системного программного обеспечения.

В качестве примера реализации аппаратуры и операционной системы для однопрограммной ВС в данном пособии рассматриваются центральный процессор Intel 8086 (сокращенно – i8086) и операционная система MS-DOS. Данный выбор обусловлен, во-первых, тем, что MS-DOS является самой распространенной однопрограммной операционной системой. Во-вторых, эта система разрабатывалась для совместного использования с процессором i8086, который аппаратно имитируется всеми последующими процессорами фирмы Intel. Выбор примера реализации однопрограммной ВС позволяет, во-первых, вести изложение на достаточно конкретном уровне. Во-вторых, многие результаты изложения могут быть перенесены на другие однопрограммные ВС, а в некоторой степени и на мультипрограммные системы.

Для изучения данного пособия требуется наличие первоначальных знаний и навыков по программированию на языке ассемблера для процессора i8086. Для их получения рекомендуются пособия автора по курсу “Информатика” [1, 2], но можно пользоваться и другими доступными учебными пособиями по этому языку программирования.

Так как пособие ориентировано, прежде всего, на получение студентами знаний и навыков по разработке управляющих программ, в нем имеется достаточно большое (9) число законченных программ, а также фрагменты программ. Почти все эти программы получены автором пособия и являются рабочими (не считая драйверов для гипотетических устройств).

В завершении данного пособия приведены методические указания по выполнению лабораторных и контрольных работ. Целью выполнения лабораторных работ является получение практических навыков по программированию операций ввода-вывода на уровне аппаратных интерфейсов (на уровне портов). Программирование ведется на языке ассемблера для процессора i8086 в среде операционной системы MS-DOS. В качестве устройств ввода-вывода используются экран и клавиатура.

Целью выполнения контрольной работы №1 является закрепление основных определений и теоретических положений данного курса. Данная контрольная работа выполняется в диалоге с компьютерной контролирующей программой.

Целью выполнения контрольной работы №2 является развитие навыков программирования на ассемблере задач, описание которых приведено в настоящем пособии.

Изучение курса “ЭВМ и ВС” заканчивается получением зачета и сдачей компьютерного экзамена. Для получения зачета требуется успешно выполнить две лабораторные и две контрольные работы, подтвердив это соответствующими отчетами. Отчеты по всем лабораторным и контрольным работам предоставляются в ТМЦДО в виде файлов на дискете.

При выполнении второй контрольной работы, а также при выполнении первой лабораторной работы используется номер варианта (от 1 до 20). Этот номер рассчитывается по формуле:

$$V = (20 \times K) \text{ div } 100,$$

где V – искомый номер варианта (при V = 0 выбирается номер варианта 20);

K – значение двух последних цифр пароля (число от 00 до 99);

div – целочисленное деление (после деления отбрасывается дробная часть).

1. ФУНКЦИИ ОДНОПРОГРАММНОЙ СИСТЕМЫ

1.1. Виртуальная машина пользователя прикладной программы

Как сказано во введении, основной функцией любой ВС является выполнение прикладных программ. На рис.1 приведена укрупненная структура однопрограммной системы. На этом рисунке показаны интерфейсы между пользователем и ВС, а также между аппаратурой и программами. При этом под ***интерфейсом*** понимается граница между двумя взаимодействующими подсистемами. Более строгое определение: ***интерфейс*** – алгоритм взаимодействия соседних подсистем. Этот алгоритм может быть задан своей блок-схемой или языком общения взаимодействующих подсистем. Например, интерфейс между аппаратурой и программами определяется правилами машинного языка.

Как видно из рис.1, однопрограммная ВС предоставляет в распоряжение своего пользователя две виртуальные (кажущиеся) ЭВМ:

1) виртуальную машину пользователя прикладной программы;
2) виртуальную машину пользователя для запуска программ.

Рассмотрим сначала первую из этих виртуальных машин. Вспомним, что однопрограммная ВС может выполнять одновременно не более одной прикладной программы. В ходе своего выполнения такая программа предоставляет пользователю возможность работать на виртуальной ЭВМ, называемой далее ***виртуальной машиной пользователя прикладной программы*** (**ВМ_П_ПП**). Общение пользователя с данной виртуальной машиной производится на языке диалога этой прикладной программы. На этом языке пользователь выполняет постановку задачи по переработке информации, а затем получает результаты ее решения.

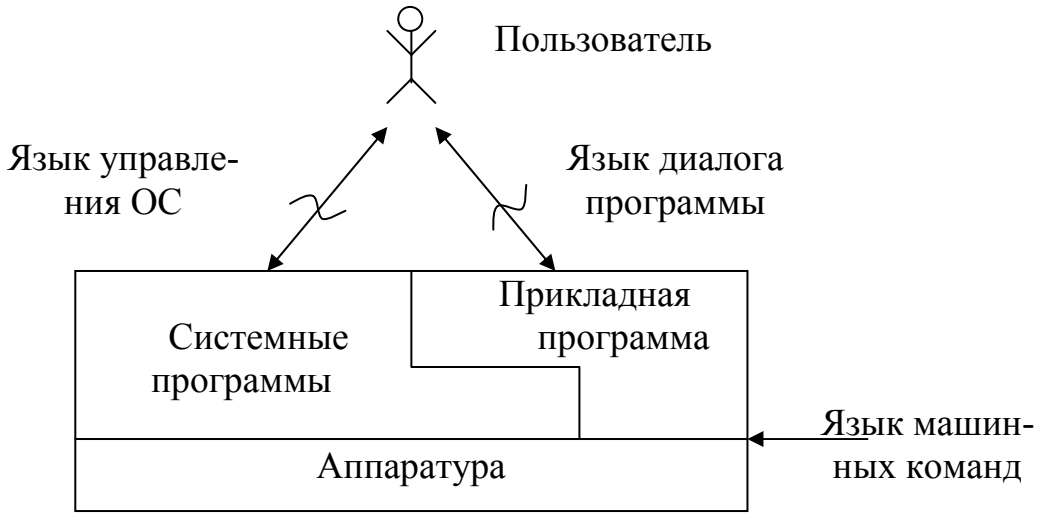


Рис.1. Подсистемы ВС

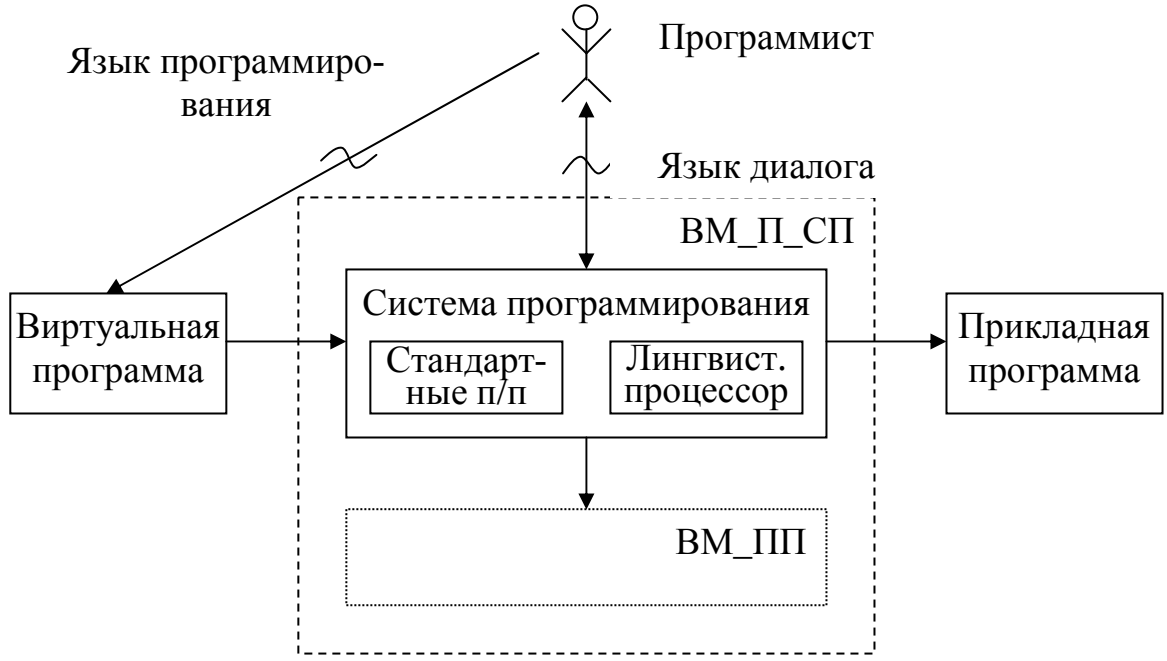
Многообразие прикладных программ чрезвычайно велико. Некоторые из этих программ, имеющие наибольшее распространение, включены в состав системных программ. Несмотря на это, для остальных системных программ и для аппаратуры эти программы выглядят точно также, как и обычные прикладные программы. Подобные программы будем называть далее ***системными обрабатывающими программами***. К ним относятся системы программирования и утилиты.

Пользователями систем программирования являются ***программисты*** – не самая многочисленная, но очень важная часть пользователей ВС. Конечной задачей любого программирования является получение реальной программы, записанной на машинном языке. Только такая программа может быть понята и выполнена центральным процессором. К сожалению, трудоемкость программирования на таком языке очень велика и не позволяет записывать на нем сколько-нибудь сложные (по решаемым задачам) программы. Решением данной проблемы является предоставление программисту возможности разрабатывать не реальную, а виртуальную прикладную программу.

Виртуальная прикладная программа записывается на языке программирования, отличном от языка машинных команд. Преобразование этой программы в реальную программу выполняет совокупность системных программ, называемая ***системой программирования***. В её состав входят стандартные подпрограммы и лингвистический процессор. Благодаря наличию системы программирования программист работает на ***виртуальной машине пользователя системы программирования*** (рис.2). Эта ВМ “понимает” операторы используемого языка программирования, а также команды управления работой лингвистического процессора.

Лингвистический процессор – системная программа, выполняющая перевод описания алгоритма с одного языка на другой. Сущность алгоритма при этом сохраняется, но форма его представления, ориентированная на программиста, преобразуется в форму, ориентированную на ЦП. Лингвистические процессоры делятся на трансляторы и интерпретаторы. В результате работы ***транслятора*** алгоритм, записанный на языке программирования (исходная виртуальная программа), преобразуется в алгоритм, записанный на машинном языке. (На самом деле, как будет показано позже, машинная программа является результатом совместной работы нескольких лингвистических процессоров.)

Трансляторы делятся на компиляторы и ассемблеры. Исходная программа для ***транслятора–ассемблера*** записывается на ***языке-ассемблере***. Один оператор данного языка транслируется в одну машинную команду. Исходная программа для ***компилятора*** записывается на языке программирования высокого уровня. Каждый оператор такого языка транслируется в несколько машинных команд. Примерами языков высокого уровня являются Паскаль и Си.



ВМ_П_СП – виртуальная машина пользователя системы программирования;

ВМ_ПП – виртуальная машина прикладной программы.

Рис.2. Виртуальная машина пользователя системы программирования

Интерпретатор в отличии от транслятора не выдаёт машинную программу целиком. Выполнив перевод очередного оператора исходной программы в соответствующую совокупность машинных команд, интерпретатор обеспечивает их выполнение. Затем преобразуется тот исходный оператор, который должен выполняться следующим по логике алгоритма и т.д.

Стандартная подпрограмма – подпрограмма, предназначенная для использования во многих прикладных программах. Примером являются подпрограммы вычисления математических функций. Вместо того, чтобы каждый раз заново программировать, например, вычисление функции sin(x), достаточно записать в свою виртуальную программу оператор вызова системной подпрограммы, выполняющей вычисление синуса. При получении машинной программы тексты стандартных подпрограмм «встраиваются» в нее транслятором.

Предоставляя программисту возможность работать с виртуальной машиной, сама система программирования “выполняется” на ВМ, аналогичной той, на которой выполняется прикладная программа (ВМ_ПП). Это обусловлено тем, что и прикладная программа, и система программирования являются машинными программами. Более того, с точки зрения самой ВС, между ними нет принципиальной разницы, так как и та и другая программы относятся к классу обрабатывающих программ. Рассмотрение

ВМ_ПП отложим до следующего раздела, а пока рассмотрим еще одну разновидность системных обрабатывающих программ – утилиты.

Утилиты – обслуживающие программы, которые выполняют:

1) перенос данных с одного периферийного устройства на другое или перенос данных в пределах одного устройства. Примеры: программа копирования данных на магнитном диске; программа ввода данных с клавиатуры терминала на диск; программа распечатки информации на диске;

2) программы изменения расположения данных. Это различные программы сортировки;

3) программы для изменения представления данных. Сюда относятся редакторы, которые осуществляют редактирование виртуальных программ и других текстов, а также программы перекодировки данных для согласования программ, использующих разные кодировки, или для обеспечения секретности.

Примером сложной утилиты является Norton Commander. Эта утилита переносит с диска на экран информацию, содержащуюся в любом каталоге любого логического диска. По запросу пользователя она выводит на экран файловую структуру любого логического диска. Кроме того, она предоставляет пользователю удобный язык управления операционной системой MS-DOS за счет того, что она переносит имя исполняемого файла программы из позиции экрана, отмеченной пользователем с помощью псевдокурсора (псевдокурсор – светящийся прямоугольник) в то место памяти, откуда это имя может взять интерпретатор команд ОС.

В отличии от систем программирования, утилиты используются не только программистами, но и ***пользователями–непрограммистами***. Эта наиболее многочисленная категория пользователей ВС работает на виртуальных машинах, предоставляемых готовыми прикладными программами, а также утилитами.

1.2. Виртуальные машины для запуска программ

Для того чтобы пользователь мог работать на виртуальной машине, предоставляемой прикладной программой (утилитой или системой программирования), эту программу необходимо запустить. Для этого следует выполнить действия:

1) создать исполняемую ВМ_ПП. То есть для прикладной программы должны быть выделены аппаратные ресурсы (время ЦП, пространство ОП, ПУ) в объёме не менее минимально-необходимого для выполнения этой программы. Иными словами, ВМ_ПП должна опираться (отображаться) на реальную аппаратуру, достаточную для выполнения программы;

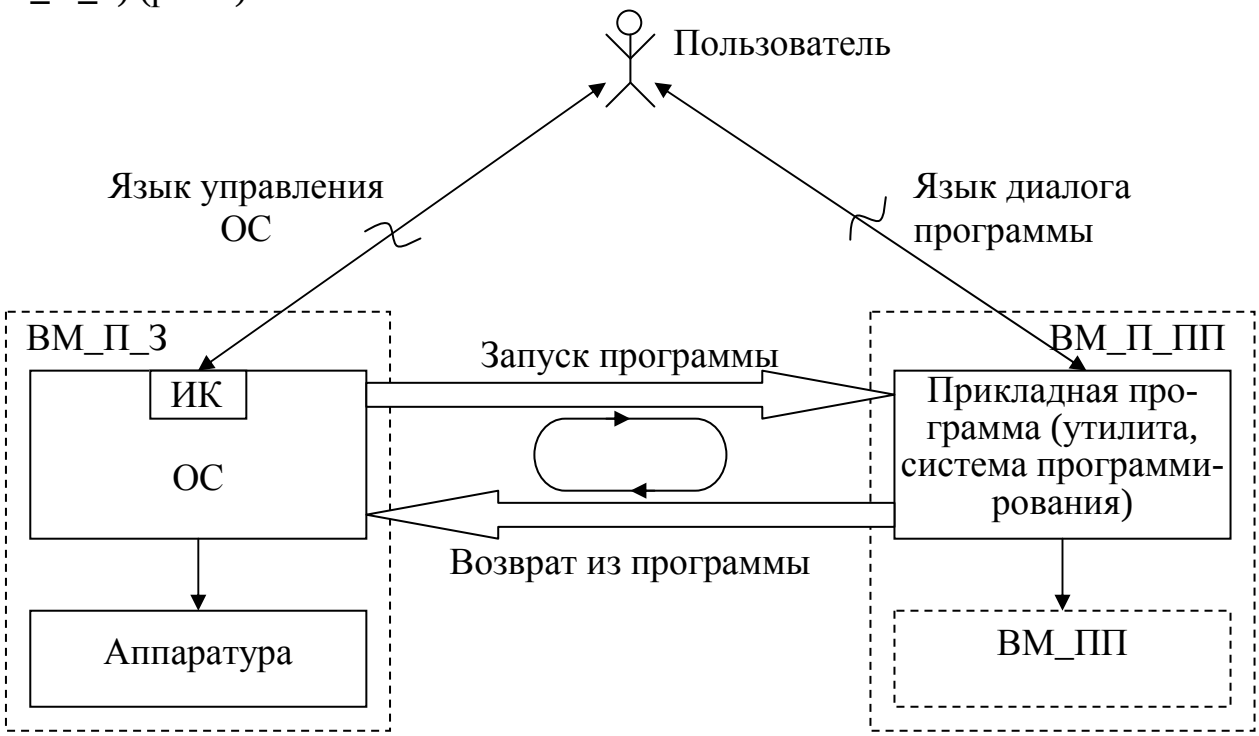
2) загрузить в эту ВМ требуемую прикладную программу, утилиту или систему программирования;

3) инициировать программу. ***Инициирование*** – запуск, приведение в действие. (Не путать с термином ***инициализация*** – начальная подготовка к работе.)

Для задания требуемой прикладной программы пользователь ВС пользуется языком ***управления операционной системой***, называемым также ***командным языком*** или ***языком директив***. Одна команда этого языка задаёт выполнение одной прикладной программы (утилиты или системы программирования). Поэтому язык управления ОС может рассматриваться как язык программирования очень высокого уровня.

Вспомним, что операторы языка программирования могут обрабатываться одним из двух видов лингвистических процессоров – транслятором или интерпретатором. Операторы языка управления ОС обрабатываются модулем ОС, называемым ***интерпретатором команд*** (**ИК**). Другие названия этого модуля: ***командный процессор***, ***командная оболочка***. ИК представляет собой лишь “видимую”, сравнительно небольшую часть ОС. На самом деле, чтобы интерпретировать, т.е. выполнить (преобразовав в совокупность машинных команд), очередную команду пользователя, ИК инициирует многие другие модули ОС. Одни из них ищут текст требуемой прикладной программы на диске, другие выделяют необходимые аппаратные ресурсы (обеспечивают исполняемость ВМ_ПП), третьи загружают программу в ОП и инициируют её.

Имея в своём распоряжении язык управления ОС, пользователь работает на ***виртуальной машине пользователя для запуска программ*** (**ВМ_П_З**) (рис.3).



ВМ_П_З – ВМ пользователя для запуска программ
ВМ_П_ПП – ВМ пользователя прикладной программы
ВМ_ПП – ВМ прикладной программы

Рис.3. Виртуальные машины пользователя в однопрограммной ВС

После того, как пользователь запустил прикладную программу (утилиту или систему программирования), он работает на ВМ, предоставляемой данной программой. После завершения прикладной программы пользователь опять оказывается на ВМ_П_З. Подобная смена виртуальных машин происходит до завершения текущего сеанса работы пользователя с ВС.

Как правило, однопрограммные ВС позволяют своим пользователям выдавать команды управления как по одиночке, так и группами, называемыми ***командными файлами***. В этом случае одному сеансу работы на ВМ_П_З соответствуют несколько сеансов работы на различных ВМ_П_ПП. В остальном описанная выше схема остаётся справедливой.

1.3. Виртуальная машина прикладной программы

Предоставление разработчику прикладной программы возможности разрабатывать не реальную, а виртуальную программу многократно уменьшает трудоемкость разработки. Другим важным направлением снижения трудоемкости прикладного программирования является изменение среды выполнения прикладной программы, приводящее к её упрощению. (Понятно, что чем проще машинная программа, тем проще её разработка.)

Изменение среды выполнения прикладной программы достигается путём предоставления ей возможности пользоваться услугами системных подпрограмм управления аппаратурой. Примером такой подпрограммы является подпрограмма, выполняющая вывод символьной информации на устройство печати (принтер). Вместо того, чтобы реализовывать в своей прикладной программе функцию печати (для этого нужны сотни машинных команд), наша программа использует единственную машинную команду для вызова соответствующей системной подпрограммы. С учётом того, что для каждого аппаратного устройства в ВС имеется одна или несколько управляющих подпрограмм, прикладная программа имеет в своём распоряжении не реальную машину с “голой” аппаратурой, а ***виртуальную машину прикладной программы*** (**ВМ_ПП**) (рис.4).

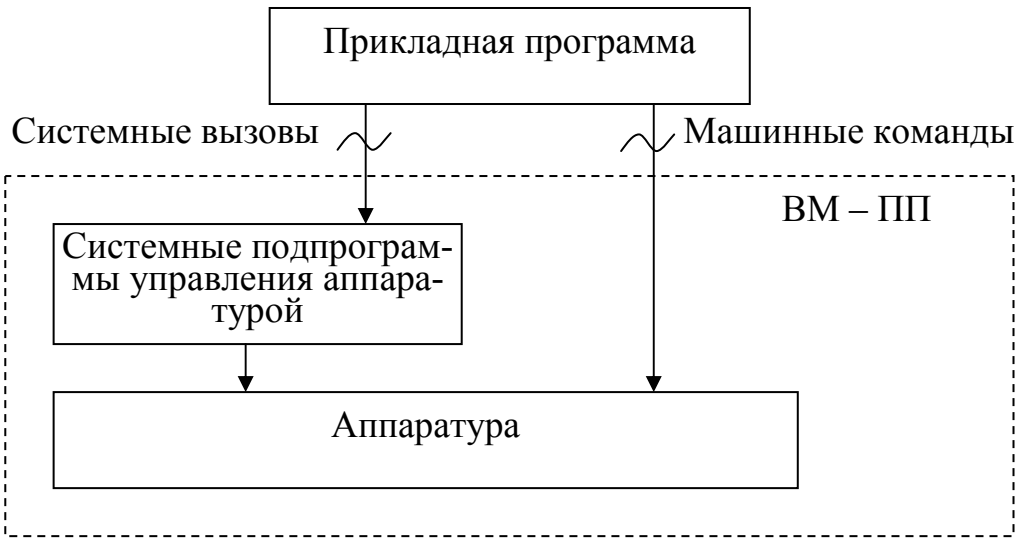


Рис.4. Виртуальная машина прикладной программы

ВМ_ПП предоставляет в распоряжение прикладной программы виртуальные устройства:

1) виртуальный ЦП. Несмотря на то, что в прикладной программе присутствуют только машинные команды реального ЦП, эффект от выполнения некоторых из них многократно усиливается за счёт вызова ими (при их выполнении на ЦП) системных подпрограмм. Другим отличием виртуального ЦП от реального является то, что он обслуживает нашу прикладную программу непрерывно (реальный ЦП может отвлекаться от нашей программы, выполняя другие программы);

2) виртуальная ОП. Объём этой памяти может многократно превосходить объём реальной ОП, предоставляемой в распоряжение программы. Кроме того, способ указания (способ адресации) ячеек виртуальной ОП обычно отличается от соответствующего способа для реальной ОП;

3) виртуальная ВП. Такая память имеет структуру, существенно отличную от структуры реальной ВП. Благодаря этому пользоваться виртуальной ВП намного удобнее;

4) виртуальные ПУ. Если наша программа работает с периферийным устройством путём вызова соответствующей системной подпрограммы, то эта подпрограмма и представляет для программы виртуальное ПУ.

Все перечисленные виртуальные устройства (за исключением ОП) реализуются путем вызова из прикладной программы системных управляющих подпрограмм. (Виртуальная ОП реализуется в общем случае совместными усилиями аппаратуры ЦП и управляющих программ.) Такие вызовы будем называть далее ***системными вызовами***. Они разделяются на две группы (рис.5):

1) вызовы BIOS;

2) вызовы операционной системы (ОС).

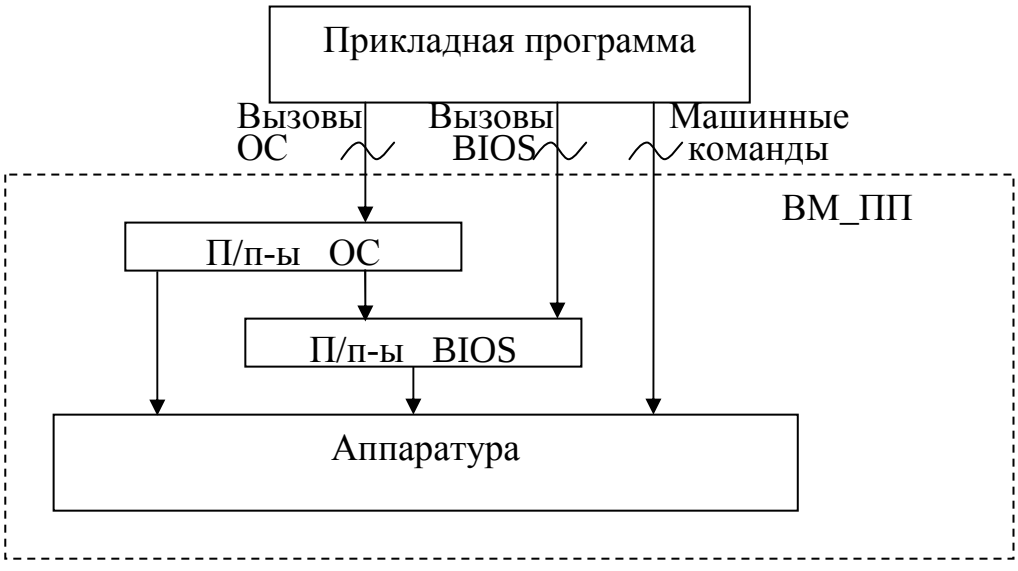


Рис.5. Уточнение виртуальной машины прикладной программы

Подпрограммы BIOS (базовая система ввода-вывода) являются своеобразным “продолжением аппаратуры”. Они ”зашиты” в постоянное запоминающее устройство (ПЗУ) и могут выполняться сразу же после включения питания машины. Это свойство широко используется: одной из подпрограмм BIOS является начальный загрузчик, инициирующий программное обеспечение ВС. Другие подпрограммы BIOS могут использоваться в процессе своего выполнения прикладными и системными программами для работы с ПУ.

1.4. Структура аппаратных средств

Обязательной подсистемой любой ВС является аппаратура. В однопрограммной системе возможно применение одной из следующих структур организации аппаратных средств:

1) с общим ЦП;
2) с общей ОП;
3) с общей шиной.

Из этих структур наиболее распространена структура с общей шиной (рис.6). В данной структуре центральным связывающим звеном между основными блоками является ***общая шина*** (**ОШ**) – группа проводов. ОШ в общем случае есть объединение трех шин: 1) шины управления; 2) шины адреса; 3) шины данных. Рассмотрим кратко назначение других аппаратных блоков.

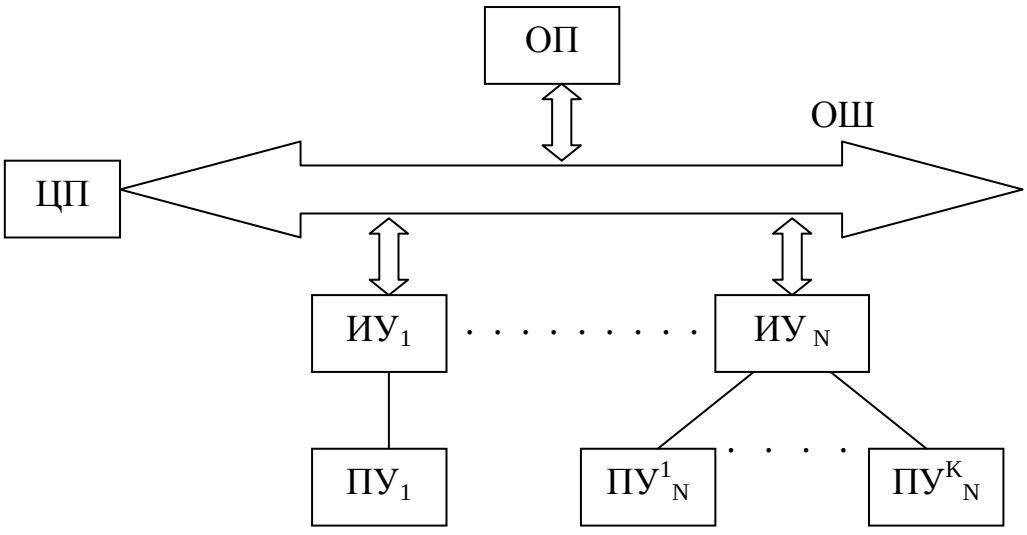


Рис. 6. Структура аппаратуры ВС с общей шиной

Центральный процессор (**ЦП***)* – “мозг” ЭВМ. Он обеспечивает выполнение прикладных и системных программ. Любая программа представляет собой последовательность машинных команд (инструкций), каждая из которых требует для своего размещения один, два или большее число бай-

тов. На рис.7 приведена структура наиболее типичной машинной команды. Здесь **КОП** – код операции. Это комбинация битов, кодирующая тип операции, которую следует выполнить над операндами (например, суммирование). Операнд 1, операнд 2 – это или сами данные, над которыми выполняется машинная команда, или адреса в памяти (ОП или регистры), где эти данные находятся.

КОП	Операнд 1	Операнд 2

Рис.7. Структура машинной команды

Оперативная память (**ОП**) предназначена для кратковременного хранения программ и обрабатываемых ими данных. Название обусловлено тем, что операции чтения содержимого ячеек памяти и записи в них нового содержимого производятся достаточно быстро. Иногда используют другое название – ***операционная память***. Это название обусловлено тем, что ЦП может достаточно просто считывать машинные команды из ОП и исполнять их. Структура любой ОП представляет собой линейную последовательность пронумерованных ячеек (рис.8). В зависимости от ЭВМ ячейкой является байт или машинное слово. Номер ячейки называется ***физическим*** или ***реальным адресом*** этой ячейки. В простейших ЭВМ поле операнда в машинной команде содержит этот номер.

Периферийные устройства (ПУ) – устройства ввода-вывода и устройства внешней памяти. Посредством ***устройств ввода-вывода*** ЭВМ «разговаривает» с человеком-пользователем. Сюда относятся: клавиатура, экран (дисплей), телетайп, печатающее устройство и т.д.

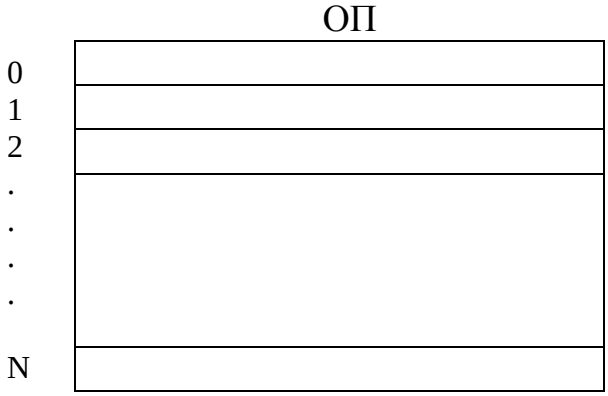


Рис. 8. Структура ОП

Устройство внешней памяти предназначено для работы с ***носителем внешней памяти***. Примером такого устройства является дисковод. Он работает с носителем внешней памяти – магнитным диском. ***Внешняя память*** (**ВП**) имеет следующие отличия от ОП:

1) обмен информацией между ЦП и ВП выполняется во много раз медленнее, чем между ЦП и ОП;

2) ЦП не может выполнять команды, записанные в ВП. Для выполнения этих команд их необходимо предварительно переписать в ОП;

3) информация на носителе ВП сохраняется и после выключения питания.

В недавнем прошлом емкость носителей ВП многократно превосходила емкость ОП. В настоящее время это выполняется не для всех типов носителей.

Интерфейсное устройство (**ИУ**) предназначено для того, чтобы согласовать стандартную для данной ЭВМ структуру ОШ с конкретным типом ПУ, которых существует очень много.

2. ЦЕНТРАЛЬНЫЙ ПРОЦЕССОР

2.1. Архитектура процессора i8086

В данном разделе рассматривается применение в качестве ЦП микропроцессора i8086. Данный процессор обеспечивает выполнение программ в однопрограммной ВС, обладая всеми необходимыми для этого ресурсами. Знакомство с этими ресурсами начнем с рассмотрения архитектуры (структуры) процессора (рис.9).

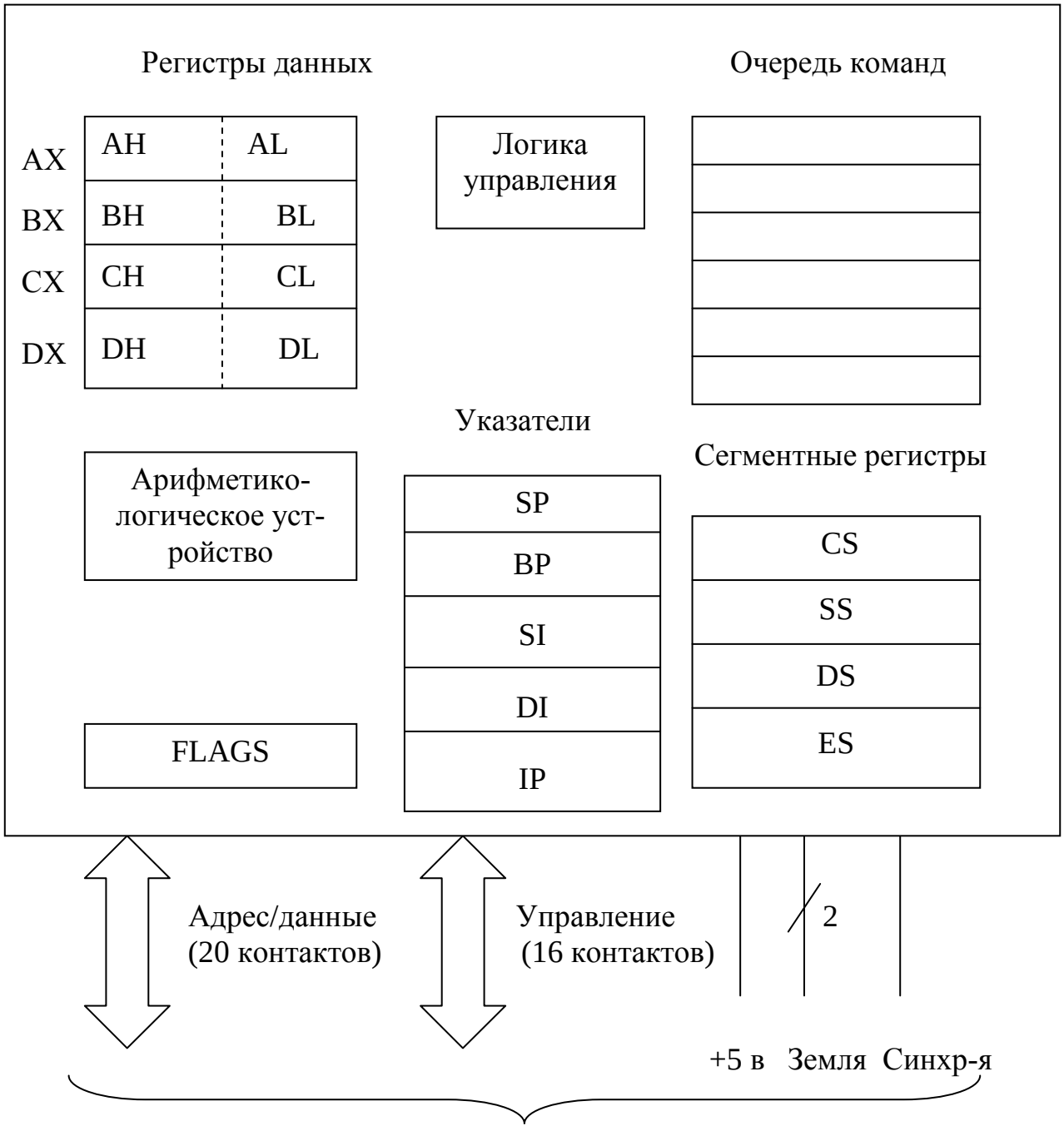


Рис. 9. Структура микропроцессора i8086

Блок ***логика управления*** выполняет дешифрирование и исполнение машинных команд. Если очередная команда является арифметической или логической, то она направляется логикой управления для выполнения в ***арифметико-логическое устройство***.

Регистры данных предназначены для хранения операндов и результатов операций. ***Регистр*** – ячейка памяти, скорость обмена с которой намного выше, чем с другими видами памяти: с ОП и, тем более, с ВП. К регистрам данных относятся регистры **AX**, **BX**, **CX** и **DX**. Они допускают адресацию не только целых регистров, но и их младшей и старшей половин. Например, допускается использовать два байта в регистре AX вместе, а также указывать отдельные байты - **AL** (младший) и **AH** (старший). Регистры BX, CX и DX, кроме обычных функций, имеют и специальные назначения.

Очередь команд содержит текущую команду на время ее дешифрации (распознавания) и выполнения. Кроме того, для повышения быстродействия в данную очередь заранее считываются из ОП следующие по порядку в программе команды.

В группу регистров ***указатели*** входят указатель команды IP и указатель стека SP, а также регистры BP**,** SI и DI**.** ***Указатель команды*** **IP –** очень важный регистр, который содержит относительный адрес (смещение) следующей команды, подлежащей выполнению на ЦП. Указатель стека SP также очень важен: он содержит относительный адрес вершины стека. Подробнее регистры IP и SP будут рассмотрены позже.

Остальные регистры-указатели используются для адресации ячеек какого-то массива относительно его начала. При этом ***базовый регистр*** **BP** часто используется для адресации ячеек программного стека относительно его вершины**.** А ***индексные регистры*** **SI** и **DI** обычно используются при циклической обработке элементов массива. При этом для того, чтобы обрабатывать одной и той же машинной командой различные ячейки массива, достаточно перед выполнением тела цикла осуществлять изменение содержимого того индексного регистра, который используется для адресации ячеек массива.

Регистр флагов **FLAGS** отражает текущее состояние ЦП. Структура этого регистра приведена на рис.10.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Рис.10. Регистр флагов микропроцессора i8086

Семь битов в FLAGS не используются. Остальные биты (флажки) делятся на условные и управляющие. ***Условные флажки*** отражают результат предыдущей арифметической или логической операции. Это:

1) **SF** – ***флажок знака***. Равен старшему биту результата. Т.к. в дополнительном коде старший бит отрицательных чисел содержит 1, а у положительных он равен 0, то SF показывает знак предыдущего результата;

2) **ZF** – ***флаг нуля***. Устанавливается в 1 при получении нулевого результата и сбрасывается в 0, если результат не равен 0;

3) **PF** – ***флажок паритета***. Устанавливается в 1, если младшие 8 битов результата содержат четное число единиц, в противном случае он сбрасывается в 0;

4) **CF** – ***флажок переноса***. При сложении (вычитании) устанавливается в 1, если возникает перенос (заем) в старший бит (из старшего бита). Обычно данный флаг используется не по прямому назначению, а как признак возврата из подпрограммы: если подпрограмма (процедура или обработчик прерываний) завершилась успешно, то она возвращает CF=0, а если с ошибкой, то CF=1;

5) **AF** – ***флажок вспомогательного переноса***. Устанавливается в 1, если при сложении (вычитании) возникает перенос (заем) из бита 3. Флаг предназначен только для двоично-десятичной арифметики;

6) **OF** – ***флажок переполнения***. Устанавливается в 1, если знаковый бит изменился в той ситуации, когда этого не должно было произойти.

Пусть, например, машинная команда ADD (здесь и везде далее используются ассемблерные мнемоники машинных команд) выполнила следующее сложение:

$$\begin{array}{r} 0010\ \ 0011\ \ 0100\ \ 0101 \\ +\ \ 0011\ \ 0010\ \ 0001\ \ 1001 \\ \hline 0101\ \ 0101\ \ 0101\ \ 1110 \end{array},$$

тогда после ее выполнения получаются состояния флажков:

SF = 0, ZF = 0, PF = 0, CF = 0, AF = 0, OF = 0.

Если ADD выполнила сложение:

$$\begin{array}{r} 0101\ \ 0100\ \ 0011\ \ 1001 \\ +\ \ 0100\ \ 0101\ \ 0110\ \ 1010 \\ \hline 1001\ \ 1001\ \ 1010\ \ 0011 \end{array},$$

то флажки принимают состояния:

SF = 1, ZF = 0, PF = 1, CF = 0, AF = 1, OF = 1.

Флажки управления влияют на выполнение специальных функций. Эти флажки устанавливаются лишь несколькими специальными машинными командами. Это флажки:

1) **DF** – ***флажок направления***. Он используется при выполнении команд, обрабатывающих цепочки – последовательности ячеек памяти. Если флаг сброшен, цепочка обрабатывается с первого элемента, имеющего наименьший адрес. Иначе цепочка обрабатывается от наибольшего адреса к наименьшему;

2) **IF** – ***флажок разрешения прерываний***. Когда установлен этот флажок, ЦП выполняет маскируемые прерывания. Иначе эти прерывания игнорируются;

2) **TF** – ***флажок трассировки***. Если этот флажок установлен, то после выполнения каждой машинной команды, ЦП генерирует внутреннее аппаратное прерывание (прерывание номер 1).

Рассмотренный регистр FLAGS чрезвычайно важен не только для понимания логики работы ЦП, но и всей ВС в целом. Это обусловлено тем, что данный регистр фактически является дескриптором ЦП. ***Дескриптор*** (или ***блок управления***) – структура данных, используемая для управления модулем ВС. Каждый сколько-нибудь сложный модуль ВС имеет свой блок управления. На протяжении данного пособия нами будет рассмотрено много различных дескрипторов.

Последний блок ЦП образуют сегментные регистры CS, SS, DS и ES. Данные регистры используются для адресации ячеек ОП.

2.2. Адресация памяти

Применение сегментных регистров вызвано стремлением расширить объем программно адресуемого пространства ОП. Дело в том, что 20-и проводная шина адреса (входит в состав ОШ) позволяет адресовать до 1 млн ячеек (байтов) ОП, так как $2^{20} \approx 1$ млн. Но все регистры в ЦП 16-битные. Ни одного 20-битного нет. Максимальный объем памяти, который можно адресовать с помощью 16 бит, составляет 64К (1К = 1024). Рассмотрим как получаются 20-битные адреса.

Мысленно разобьем ОП на участки по 16 байт, называемые ***параграфами*** (рис.11). Регистр сегмента кода **CS** содержит номер параграфа, с которого начинается выделенный нашей программе сегмент. Например, это может быть число 0002h (буква h означает шестнадцатеричную систему счисления). Для получения реального адреса начальной ячейки параграфа необходимо его номер (0002h) умножить на число 16: 0002h x 10h = =00020h.

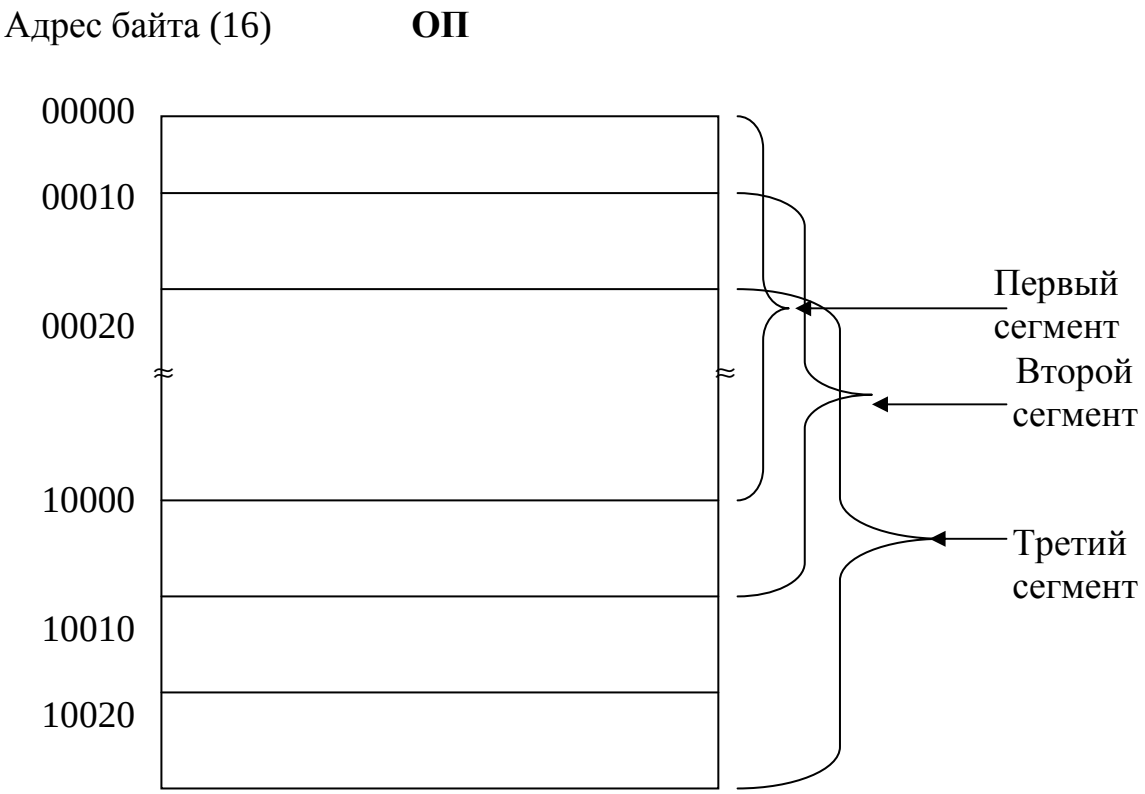


Рис. 11. Разбиение ОП на параграфы

Указатель команды IP содержит относительный адрес (смещение) адресуемой ячейки относительно начала сегмента. Допустим, что это число 100h. Физический адрес R адресуемой ячейки ЦП получает непосредственно перед обращением к ОП путем суммирования содержимого регистра CS, умноженного на 16 (10h), с содержимым регистра IP:

R = (CS) x 10h + (IP).

Например, пусть (CS) = 0002h, а (IP) = 0100h, тогда R = 2h x 10h + +100h = 120h (рис.12).

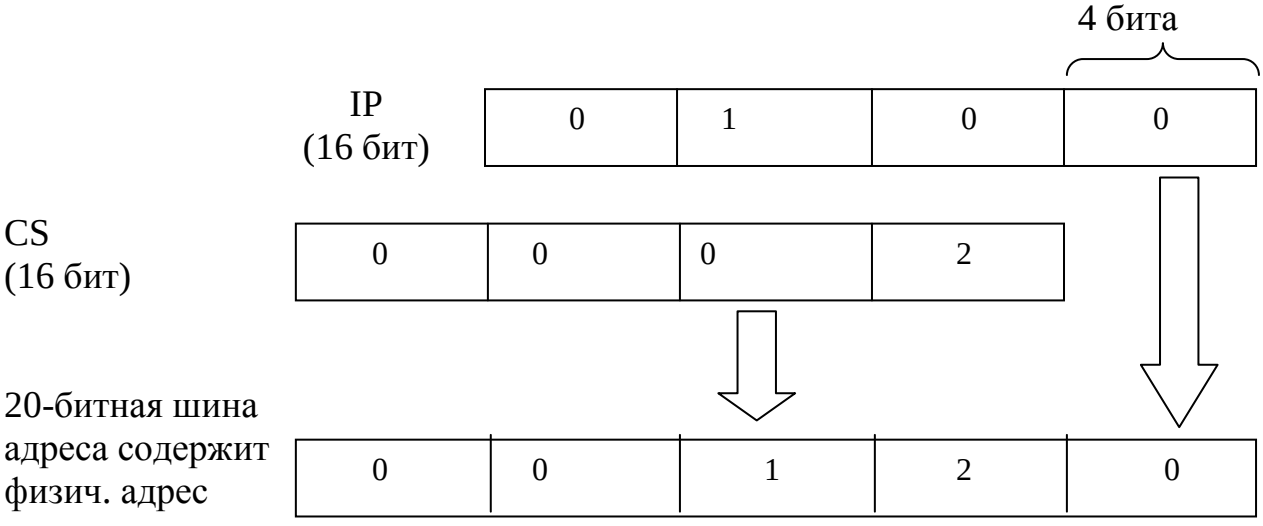


Рис.12. Получение физического адреса

В пределах текущего сегмента IP может обращаться к любой ячейке ОП, имеющей смещение относительно начала сегмента $0 \div 2^{16}-1$, т.е. $0 \div 65535$. Число $2^{16} = 65536 = 10000h = 64K$ называется длиной сегмента.

Имея в своем распоряжении четыре сегментных регистра, программа (а точнее – выполняющий ее ЦП) использует их по разному: CS – для адресации машинных команд, SS – для адресации ячеек стека, DS – для основной адресации данных, ES – для дополнительной адресации данных. Поэтому в любой момент времени любая машинная программа имеет в своем распоряжении четыре логических сегмента ОП размером 64K. Каждый из этих логических сегментов играет строго определенную роль при выполнении команд машинной программы:

1) ***сегмент кода*** – сегмент ОП, номер начального параграфа которого находится в регистре CS. Следующая команда программы выбирается только из сегмента кода;

2) ***сегмент данных*** – сегмент ОП, номер начального параграфа которого – в регистре DS. Команды, у которых ячейки памяти задаются только одним операндом, имеют дело с этим сегментом данных;

3) ***дополнительный сегмент данных*** – сегмент ОП, номер начального параграфа которого – в регистре ES. Команды, у которых ячейки памяти задаются обоими операндами, используют и сегмент данных и дополнительный сегмент данных.

4) ***сегмент стека*** – сегмент ОП, номер начального параграфа которого в регистре SS. Команды, выполняющие операции со стеком, имеют дело с тем стеком, на который “указывает” SS.

Перечисленные логические сегменты как бы «высвечиваются» ЦП из одно-мегабайтового адресного пространства ОП. Причем они могут пересекаться друг с другом, или вообще совпадать. Выполняя запись в сегментные регистры (например, с помощью команды MOV) нового содержимого, программа выполняет замену «высвечиваемых» сегментов.

Для того, чтобы можно было обращаться к любой ячейке логического сегмента, в распоряжении программы имеется регистр, содержащий смещение искомой ячейки относительно начала сегмента. Это:

1) для сегмента кода – IP;
2) для сегмента данных – BX, SI, DI;
3) для дополнительного сегмента данных – BX, DI;
4) для сегмента стека – SP.

Таким образом, логический адрес любой ячейки ОП представляет собой пару:

(S, L) = ((регистр сегмента) , (регистр смещения)),

где S – начальный адрес сегмента (номер параграфа);

L – смещение ячейки относительно начала сегмента;

(регистр) – содержимое регистра.

Далее будем называть S ***адресом-сегментом***, а L – ***адресом-смещением***.

Преобразование логического адреса в физический происходит при попадании соответствующей машинной команды на ЦП так, как это показано на рис.12. Так как это преобразование происходит во время выполнения программы, то оно называется ***динамическим преобразованием адреса***. (***Статическая*** операция производится до начала выполнения программы.)

2.3. Алгоритм работы процессора

На рис.13 приведен алгоритм выполнения машинных команд в ЦП. Он представляет собой упрощенный вариант работы реального процессора i8086, в котором некоторые этапы выполняются во времени не последовательно, а параллельно.

Данный алгоритм представляет собой бесконечный цикл, который инициируется сразу же после включения питания. На одной итерации алгоритма ЦП выполняется одна машинная команда. Это выполнение начинается с определения реального адреса команды в ОП. Этот адрес выдается из ЦП на шину адреса. Получив его, ОП помещает следующую команду на шину данных, и ЦП вводит команду в очередь команд. Пока дешифрируется эта команда – определяется ее длина в байтах и производится увеличение содержимого IP на эту длину, в результате чего IP адресует следующую машинную команду. После этого цикл повторяется.

Последовательная выборка команд из памяти и их выполнение продолжаются до тех пор, пока очередной командой, поступившей из ОП на ЦП, не окажется команда перехода. Команды перехода позволяют изменить естественный порядок следования машинных команд. Они делятся на команды безусловного и команды условного перехода.

Команды безусловного перехода обязательно изменяют естественный порядок выполнения команд. Существуют пять машинных команд безусловных переходов. Все они имеют одну и ту же ассемблерную мнемонику **JMP** и один операнд. Эти команды можно разбить на две группы: команды близких и команды дальних переходов.

Команда близкого перехода выполняет безусловный переход в пределах текущего сегмента кода. Это делается путем замещения содержимого IP (т.е. внутрисегментного адреса следующей по порядку команды) адресом, задаваемым самой командой перехода.

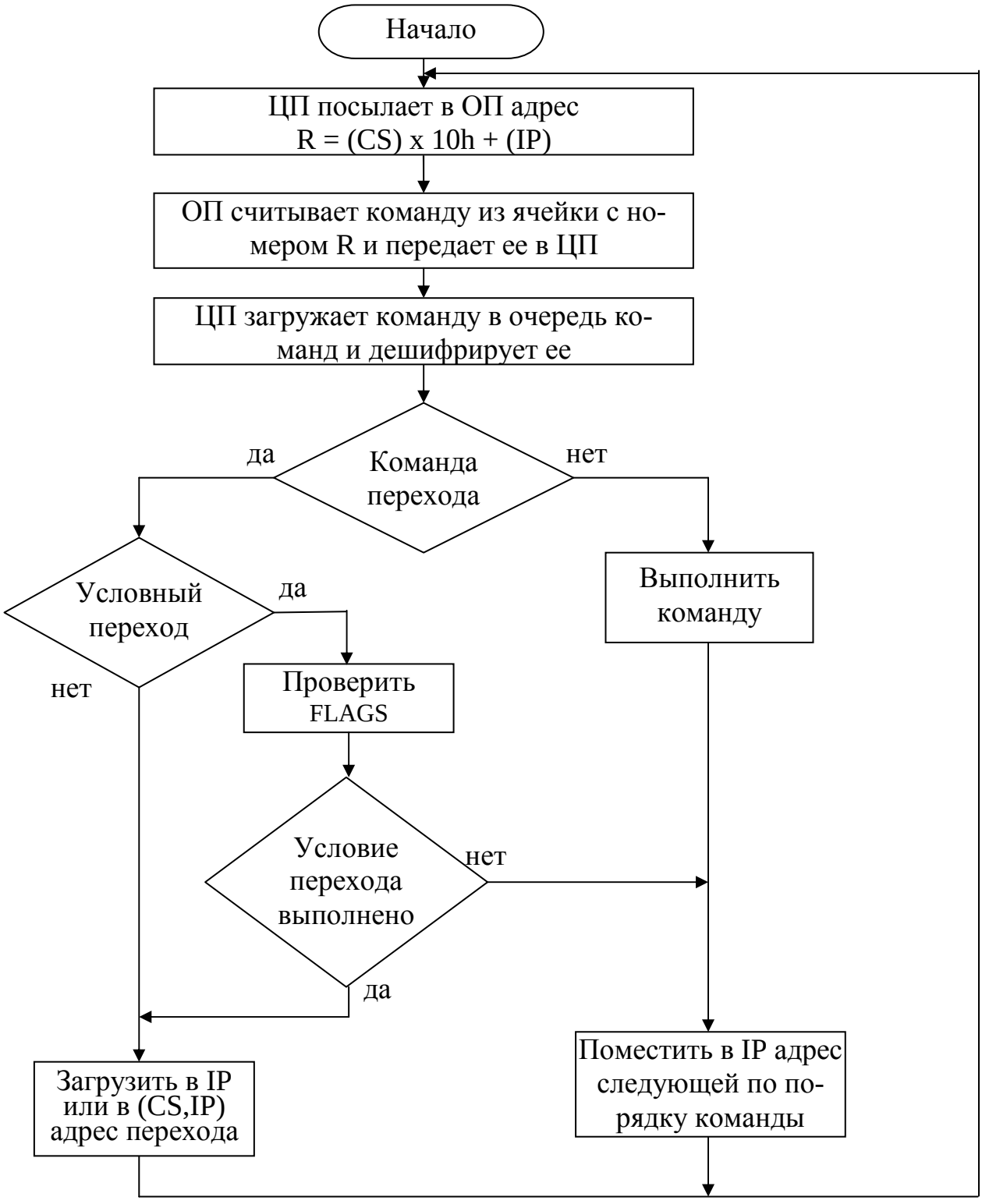


Рис.13. Алгоритм работы ЦП

Команда дальнего перехода выполняет безусловный переход на ячейку ОП, расположенную за пределами текущего сегмента кода. Это делается путем замещения содержимого не только IP, но и CS значениями,

содержащимися в самой команде перехода. Следствием этого является замена логического сегмента кода, и поэтому до тех пор, пока на ЦП не поступит следующая команда дальнего перехода, все последующие команды будут выбираться из нового сегмента.

Кроме команд JMP безусловный переход выполняют команды CALL, RET, INT, IRET. Эти команды мы рассмотрим чуть позже, а пока лишь отметим, что мнемоникам CALL и RET соответствуют по две машинные команды, одна из которых выполняет близкий, а другая – дальний безусловный переход. Команды INT и IRET выполняют только дальние переходы.

Команды условных переходов замещают или не замещают содержимое IP в зависимости от результатов предыдущих команд, отраженных в регистре FLAGS. Например, если после команды SUB (вычитание) в программе находится команда JZ (переход по нулю), то переход осуществляется в том случае, если результат вычитания нулевой, и поэтому ZF (флаг нуля в FLAGS) установлен. Если же ZF=0 (ненулевой результат вычитания) переход не производится. Так как команды условных переходов могут выполнять только очень близкие переходы (не далее 128 байтов), то для реализации больших переходов (в том числе и за пределы сегмента кода) каждый такой оператор дополняется оператором безусловного перехода.

2.4. Работа со стеком

В принципе, любая прикладная программа может обрабатывать любые списки (связанные и несвязанные), но несвязанный стек – единственный вид списков, для работы с которым ЦП имеет специальные регистры и команды. Такое внимание к несвязанному стеку, называемому далее просто ***стеком***, обусловлено тем, что он используется для организации управляющих взаимосвязей между подпрограммами (это применение стека будет рассмотрено в следующих параграфах), и поэтому любая прикладная программа обязательно имеет свой стек.

Для небольших программ, получаемых в результате загрузки com-файлов стек программы создается автоматически, без какого-либо участия программиста. В этом случае данные стека располагаются в том же самом сегменте памяти объемом 64К, что и код программы, а также ее данные.

Допустим, что для нашей программы ОС выделила область (сегмент) ОП начиная с параграфа N 20h (рис.14). Регистр CS содержит число 20h и представляет собой указатель на начало этой области. Одновременно с назначением области ОП для размещения программы ОС назначает сегмент для размещения стека, который будет обслуживать нашу программу. Специальный регистр **SS**, называемый ***регистром сегмента стека***, является указателем на начало этой области. Так как для размещения стека ОС выде-

ляет ту же область, что и для команд программы, то первоначальное содержимое SS совпадает с содержимым CS.

Рис.14. Пример размещения программы в ОП

Так как запись команд программы в сегмент памяти происходит с его начала, то данные стека должны находиться в конце сегмента. Для лучшего использования памяти стек “растет” в сторону меньших адресов. Смещение вершины относительно начала сегмента хранится в регистре **SP**, который называется ***указателем стека***. Первоначально операционная система помещает в SP максимально возможное число без знака, т.е. FFFF или чуть меньшее число. В результате SP указывает на "дно" стека. При добавлении слова данных в стек содержимое SP уменьшается на 2, а при исключении слова из стека – увеличивается на 2.

Если программа будет получена в результате загрузки exe-файла, то программист обязан сам задать в виртуальной программе на ассемблере предельный размер стека программы. Область этого объема будет зарезервирована транслятором исключительно для размещения стека программы. «Дно» стека будет находиться на старшей границе этой области, а «расти» стек будет в сторону младшей границы.

Коль скоро программа обязательно имеет свой индивидуальный стек, то она может использовать его в качестве особой области памяти для своих внутренних потребностей. Особенностью стека, присущей ему по определению, является то, что над ним допустимы только две операции: 1) добавление слова в вершину стека; 2) выборка слова из вершины стека. Несмотря на то, что никто не мешает выполнять программе операции с внутренними ячейками стека (используя базовый регистр BP), этим следует пользоваться осторожно и только при необходимости.

Для выполнения операций со стеком в программе используются две специальные команды. Включение слова данных в стек выполняет машинная команда **PUSH** α, а исключение слова из стека выполняет команда **POP** α, где α - адрес ячейки памяти (ОП или регистровой памяти), содержимое которой следует включить в стек, или в которую следует считать слово данных из стека.

Многоцелевое использование одного и того же стека аппаратурой ЦП, операционной системой и прикладной программой делает необходимым повышенное внимание при работе с ним: каждое записанное в стек слово должно быть вовремя извлечено оттуда.

2.5. Процедуры

За исключением очень небольших программ, проектирование программы предполагает ее представление в виде совокупности относительно независимых частей (модулей), называемых ***подпрограммами***. Применение подпрограмм предоставляет следующие выгоды:

1) существенное уменьшение трудоемкости программирования за счет возможности выполнять разработку программы не целиком, а по частям;

2) существенное сокращение памяти для размещения программы, так как выделив в подпрограмму многократно повторяющийся участок кода программы, достаточно выделить память лишь для одной подпрограммы, инициируя ее из различных мест программы;

3) уменьшение трудоемкости программирования за счет того, что одну и ту же подпрограмму можно использовать не в одной, а в нескольких программах. (Данное преимущество обсуждалось ранее при рассмотрении стандартных подпрограмм.)

Подпрограммы бывают двух типов: процедуры и обработчики программных прерываний. Процедуры рассматриваются в данном параграфе, а обработчики прерываний – в следующем.

Процедура – это список машинных команд, который можно вызывать из различных мест программы. Переход к процедуре называется ***вызовом***, а соответствующий переход назад называется ***возвратом***. Вызов процедуры выполняет команда **CALL** α , где α - адрес, по которому находится первая команда процедуры. Возврат осуществляет команда **RET**

(рис.15). Возврат после каждого вызова осуществляется к команде, которая находится в памяти сразу за командой CALL.

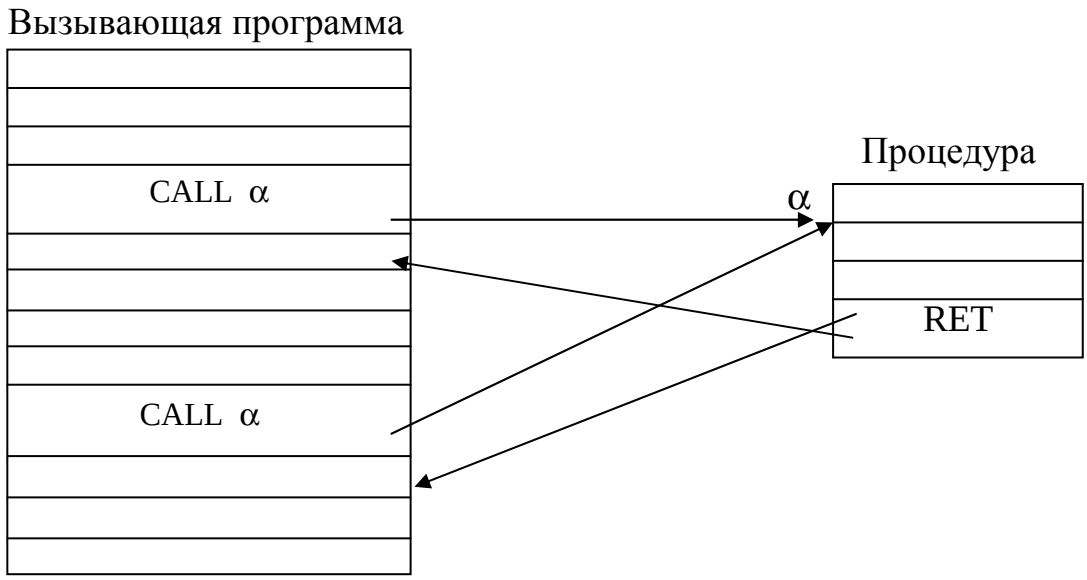


Рис.15. Вызов процедуры и возврат из нее

Подобно команде JMP, команды CALL и RET выполняют или близкие переходы, когда вызывающая программа и процедура находятся в одном сегменте кода, или дальние переходы - программа и процедура находятся в разных сегментах. В первом случае адрес перехода α представляет собой единственное число – смещение первой команды процедуры относительно начала сегмента (новое значение регистра IP). Во втором случае это пара чисел: (CS, IP).

При вызове процедуры и возврате из нее необходимо выполнить следующие три требования:

1) при вызове процедуры необходимо запомнить адрес следующей команды, чтобы можно было впоследствии осуществить возврат в нужное место вызывающей программы;

2) используемые процедурой регистры необходимо запомнить до изменения их содержимого, а перед самым выходом из процедуры – восстановить;

3) процедура должна иметь возможность выполнять обмен данными с вызывающей ее программой.

Первое требование реализуют команды CALL и RET. При близком вызове команда CALL помещает адрес следующей за ней команды (находится в IP) в программный стек, а RET извлекает этот адрес из стека и помещает его в указатель команды IP. При дальнем вызове CALL помещает в

стек, а RET извлекает оттуда не одно, а два слова. – содержимое регистров CS и IP.

Свойство стека «пришел первым, ушел последним» идеально подходит для реализации вложенных вызовов процедур (рис.16), так как оно обеспечивает размещение в стеке адреса возврата вызывающей процедуры раньше, а его выборку позже, чем соответствующие операции с адресом возврата вызываемой процедуры.

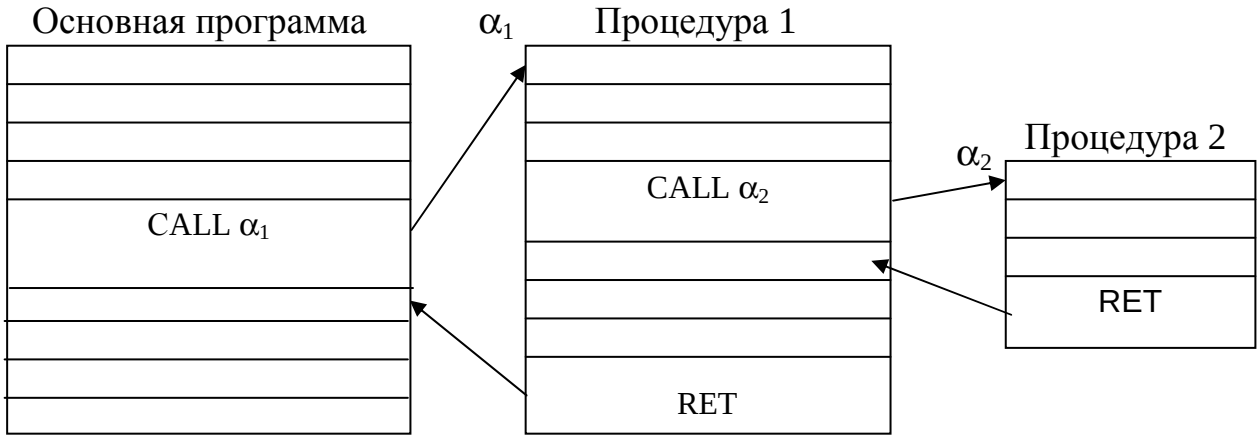


Рис.16. Вложенный вызов процедур

Требование восстановления содержимого регистров, используемых в вызывающей программе, также выполняется с помощью стека. В начале процедуры ее команды PUSH помещают в стек содержимое запоминаемых регистров, а в конце ее команды POP извлекают из стека запомненные слова и помещают их в регистры. Порядок регистров при восстановлении обратный их порядку при запоминании.

Данные, передаваемые процедуре при ее вызове, называются ее ***входными параметрами***. А данные, которые передаются процедурой в вызывающую ее программу, называются ***выходными параметрами*** процедуры. Информационный обмен между программой и процедурой обеспечивается путем использования областей памяти одного или нескольких следующих видов:

1) регистры данных;
2) программный стек;
3) другие области ОП.

Если передаваемых данных немного, то достаточно регистров данных, большее количество данных может быть передано с помощью стека. Если же этих данных слишком много, то они помещаются в область ОП, а начальный адрес этой области записывается или в регистр данных, или в стек.

В завершение сравним программную процедуру с другими видами процедур. Во-первых, любой язык программирования позволяет записывать ***виртуальные программные процедуры***. В зависимости от языка программирования они называются процедурами, функциями или подпрограммами. В любом случае текст любого из перечисленных модулей преобразуется соответствующим транслятором в код программной процедуры.

Во-вторых, программная процедура может рассматриваться как программная реализация ***логической процедуры***, которая является одним из модулей ***графического языка описания логических структур*** . Этот язык будет использован нами в дальнейшем для представления параллельных алгоритмов. Его описание приведено в приложении 1.

2.6. Прерывания

2.6.1. Внешние аппаратные прерывания

Прерыванием называется навязанное принудительно центральному процессору прекращение выполнения текущей программы и переход им на выполнение подпрограммы, которая называется ***обработчиком прерываний***. В работе любой ВС (за исключением простейших) прерывания играют исключительно важную роль. Приводимые ниже примеры иллюстрируют это.

Допустим, что на ЦП выполняется прикладная программа, вошедшая в бесконечный цикл вследствие ошибки программирования. Тогда для прекращения выполнения этой программы достаточно нажать ту комбинацию клавиш, которая специально предназначена для этой цели в используемой ВС. Например, в MS-DOS это одновременное нажатие клавиш <Ctrl> и <C>. Следствием этого является выдача интерфейсным устройством клавиатуры аппаратного сигнала прерывания, который поступает в ЦП по ОШ. Процессор прерывает выполнение зациклившейся программы и начинает выполнять обработчик прерываний клавиатуры. Эта подпрограмма распознает код нажатой комбинации клавиш (<Ctrl>&<C>) и обеспечивает завершение программы. (Обработчик прерываний клавиатуры прекращает выполнение текущей программы не сам, а использует для этого программное прерывание INT 23h.)

В качестве второго примера рассмотрим обслуживание таймера. ***Таймер*** – аппаратное устройство, выдающее сигналы прерываний в ЦП через фиксированные промежутки времени. Эти сигналы называются ***тиками***. Их можно использовать в ВС, например, для определения времени суток. Для этого по приходу каждого тика ЦП прерывает выполнение текущей прикладной программы и начинает выполнять обработчик прерываний таймера, который, в свою очередь, увеличивает на единицу счетчик

тиков и, возможно, пересчитывает текущее время, увеличивая значения счетчиков секунд, минут и часов.

Прерывания от клавиатуры и от таймера являются примерами ***внешних аппаратных прерываний***. Сигналы этих прерываний выдаются в ЦП различными ПУ (а точнее – их ИУ), которым требуется внимание со стороны программ ЦП. Эти сигналы передаются в ЦП по шине управления, которая входит в состав ОШ. Характерной особенностью этих прерываний является то, что их первичной причиной являются процессы, асинхронные (независимые) по отношению к текущей работе ЦП. Примером такого асинхронного процесса является деятельность пользователя по нажатию клавиш, приводящая к возникновению прерываний от клавиатуры и от мыши.

Аппаратные процессы, протекающие в ПУ, также могут являться первичными источниками внешних аппаратных прерываний. Характерной особенностью соответствующих сигналов прерываний является то, что они должны обрабатываться в реальном времени. Термин ***реальное время*** (**РВ**) означает, что интервал времени между моментом появления сигнала прерывания и завершением его обработки не должен превышать предельно допустимой величины, которая зависит от типа сигнала прерывания. Например, для таймера этот интервал не должен превышать один тик. Задержка завершения обработки на большую величину приведет к потере тика и, следовательно, к нарушению правильности системного времени.

Внешние аппаратные прерывания разделяются на маскируемые и немаскируемые прерывания. ***Немаскируемые прерывания*** – наиболее важные прерывания, обработка которых не может быть отложена ни на какое время. Сюда относятся прерывания, возникающие при сбоях питания. Соответствующие сигналы прерываний выдаются в ЦП аппаратными схемами контроля питания. ***Маскируемые прерывания*** – прерывания, обработка которых может быть отложена на время, требуемое ЦП для выполнения какой-то другой, более важной операции. Запрет всех маскируемых прерываний выполняет машинная команда **CLI**, а разрешение - команда **STI**. Первая из этих команд сбрасывает, а вторая устанавливает флажок разрешения прерываний IF в регистре FLAGS.

2.6.2. Исключения

Кроме внешних аппаратных прерываний существуют также ***внутренние аппаратные прерывания***, называемые обычно ***исключениями***. Источником аппаратного сигнала исключения является одна из аппаратных схем самого ЦП. Он выдается в том случае, если при выполнении на ЦП очередной машинной команды возникла ситуация, требующая помощи со стороны системных программ. Например, если при выполнении на ЦП команды DIV (деление), получилось частное, величина которого превыша-

ет предельно допустимую величину для 16-битного регистра, то возникает исключение «деление на нуль». Стандартный обработчик данного исключения завершает выполнение программы, а также выводит соответствующее сообщение на экран.

В качестве второго примера рассмотрим исключение «трассировка». Данное исключение будет происходить при выполнении на ЦП каждой машинной команды, если установлен флаг TF в регистре FLAGS. В результате этого наша прикладная программа будет «спотыкаться» - по завершению каждой команды программы будет инициироваться обработчик исключения. Результат работы этого обработчика зависит от целей трассировки. Так как обычно она выполняется с целью обнаружения ошибок в программе, то обработчик данного исключения выполняет выдачу на экран содержимого регистров ЦП. Подобный обработчик исключений входит в состав системной программы, называемой ***отладчиком***. Пример отладчика – Debug.

В завершение примера отметим, что в i8086 отсутствуют специальные команды для установки и сброса флага TF. Поэтому для выполнения требуемого действия отладчик должен: 1) с помощью команды PUSHF записать (скопировать) содержимое FLAGS в стек; 2) скорректировать слово в вершине стека, установив или сбросив бит 8; 3) с помощью команды POPF выбрать слово из вершины стека и поместить его в регистр FLAGS.

2.6.3. Программные прерывания

Третий тип прерываний – ***программные прерывания*** (термин неудачный, но общепринятый). Причиной такого прерывания является сама программа (отсюда и название), а именно: попадание на ЦП машинной команды **INT** n, где n – номер прерывания. Подобно исключениям, программные прерывания являются синхронными к текущей работе ЦП, так как обслуживают выполняемую на нем программу. Но в отличие от исключений, моменты возникновения которых автору прикладной программы заранее не известны, команды INT помещаются программистом в текст программы сознательно с целью выполнения системных вызовов.

Более того, следует четко представлять себе, что в однопрограммной ВС не существует иного способа вызова из прикладной программы системных управляющих подпрограмм (ОС или BIOS), кроме применения команды программного прерывания INT.

Примером системного вызова MS-DOS является команда INT 21h. Особенностью этого вызова является то, что под номером 21h «скрывается» программное прерывание, инициирующее ***диспетчер функций*** **MS-DOS**. Эта подпрограмма обрабатывает огромное большинство системных вызовов, адресованных MS-DOS из прикладных программ. Для того, чтобы диспетчер функций понял, какую внутреннюю процедуру MS-DOS ему

следует вызвать, в прикладной программе следует задать номер требуемой функции (в регистре AH), а если этого недостаточно, то и номер подфункции (в регистре AL).

Как и процедура, обработчик программного прерывания имеет входные и (или) выходные параметры. Для их передачи используются регистры данных и, возможно, стек программы. Использование стека менее предпочтительно, так как входные параметры будут в нем «пригружены» адресом возврата и содержимым регистра FLAGS. А выходные параметры должны быть «пригружены» этими же данными самим обработчиком, так как иначе адрес возврата не будет найден. Несмотря на то, что перечисленные трудности устраняются программой с помощью простых команд PUSH и POP, на их выполнение требуются дополнительные затраты времени ЦП.

Команды программного прерывания INT могут применяться не только для реализации системных вызовов, но и для реализации вызова прикладных подпрограмм. Реализация прикладной подпрограммы не в виде процедуры, а в виде обработчика прерываний целесообразна только тогда, когда эта подпрограмма вызывается не из одной, а из нескольких прикладных программ. Только в этом случае удается сократить размер прикладных программ за счет того, что в их код не будет включен код подпрограммы. (При использовании процедуры, наоборот, ее код должен быть включен в код каждой программы, использующей ее.) Реализация подобного вызова подпрограмм требует, чтобы они были резидентны, то есть постоянно находились бы в ОП. В противном случае выполнение каждой прикладной программы, использующей подпрограмму, должно было бы предваряться инициализацией соответствующего обработчика прерываний, что очень неудобно. Вопросы разработки резидентных программ будут рассмотрены в п.3. Там же будет приведен пример обработчика программных прерываний.

Наряду с процедурами обработчики программных прерываний относятся к подпрограммам. Поэтому команда вызова такой подпрограммы **INT** очень похожа на команду дальнего вызова процедуры CALL, а команда завершения обработчика прерываний **IRET** – на команду возврата из дальней процедуры – RET. Работа данных команд будет рассмотрена далее совместно с другими этапами общего алгоритма обработки прерываний.

2.6.4. Алгоритм выполнения прерывания

Прежде чем приводить данный алгоритм, рассмотрим два понятия, играющих в нем важную роль. Во-первых, это ***номер прерывания***. Подобно другим однотипным объектам в ВС, все типы прерываний пронумерованы. Номер – ***численный идентификатор*** объекта, пользуясь которым система находит дескриптор объекта в своих таблицах. В данном пособии нам встретятся и другие численные идентификаторы. Отметим их общее свойство: однотипные идентификаторы уникальны в пределах всей одно-

программной ВС. Например, в системе, построенной на основе i8086 и MS-DOS номера упомянутых выше прерываний следующие:

00h - Деление на нуль
01h - Трассировка
02h - Немаскируемое прерывание
. . . .
08h - Таймер
09h - Клавиатура
. . . .
10h-1Fh Прерывания BIOS
20h-3Fh Прерывания MS-DOS
. . . .

Вторым понятием, необходимым для изложения алгоритма обработки прерываний, является ***таблица векторов прерываний***. При использовании в качестве ЦП i8086 эта таблица занимает первые 1024 байта ОП (рис.17) и их никогда нельзя использовать для других целей. Один элемент этой таблицы занимает два слова (4 байта) ОП и называется ***вектором прерываний***. Он соответствует своему типу прерывания и содержит адрес (сегмент и смещение) первой ячейки обработчика прерываний в ОП. При этом первое слово содержит смещение, а второе – сегмент. Общее число векторов прерываний 256 = 100h. Следовательно, таково максимальное число типов прерываний, которые могут существовать в системе.

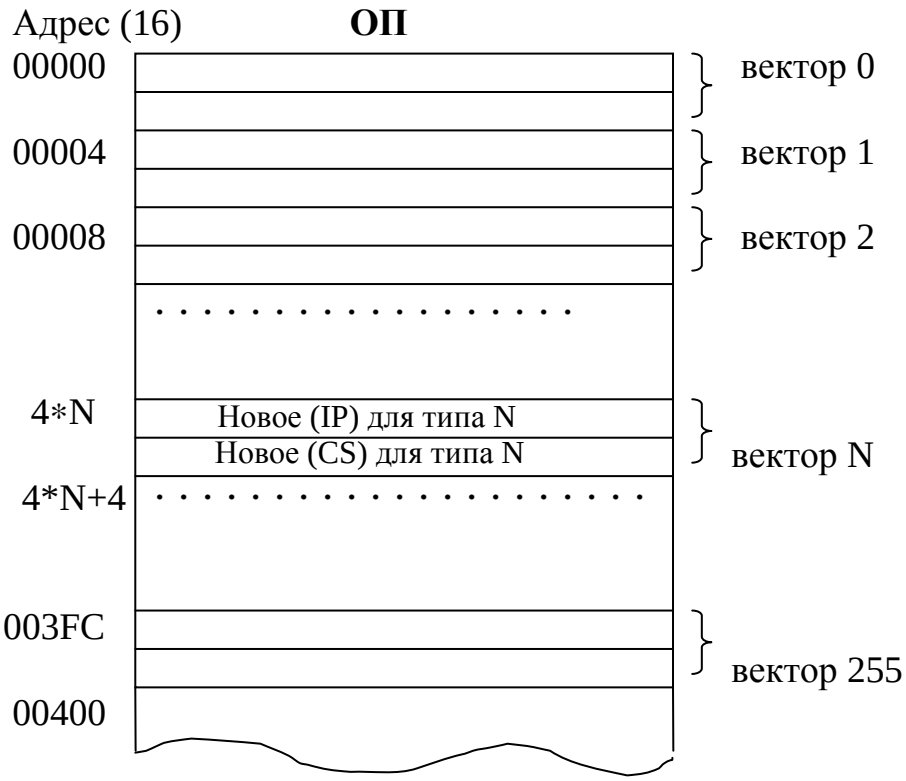


Рис.17. Таблица векторов прерываний

Обработка прерывания производится совместными усилиями аппаратуры ЦП и программы обработчика прерываний. Общий алгоритм обработки любого типа прерываний:

1) поступление сигнала прерывания в ЦП. При этом сигнал маскируемого внешнего аппаратного прерывания поступает на вход INTR процессора, сигнал немаскируемого внешнего аппаратного прерывания - на вход NMI, а сигнал исключения или программного прерывания поступает на внутреннюю линию процессора;

2) собственно прерывание. Оно выполняется ЦП по окончанию выполнения текущей машинной команды и включает действия:

– текущее содержимое FLAGS, CS и IP помещается в стек (рис.18). Последние два слова представляют собой адрес возврата в прерванную программу;

– считывание с ОШ или с внутренней шины процессора 8-битного числа – номера прерывания N;

– копирование вектора прерываний с номером N в регистры IP и CS. В результате эти регистры содержат стартовый адрес обработчика прерываний;

– сбрасываются в нуль флажки IF и TF в регистре FLAGS. В результате запрещаются все маскируемые внешние прерывания (кроме того, которое уже обрабатывается), а также запрещаются исключения «трассировка»;

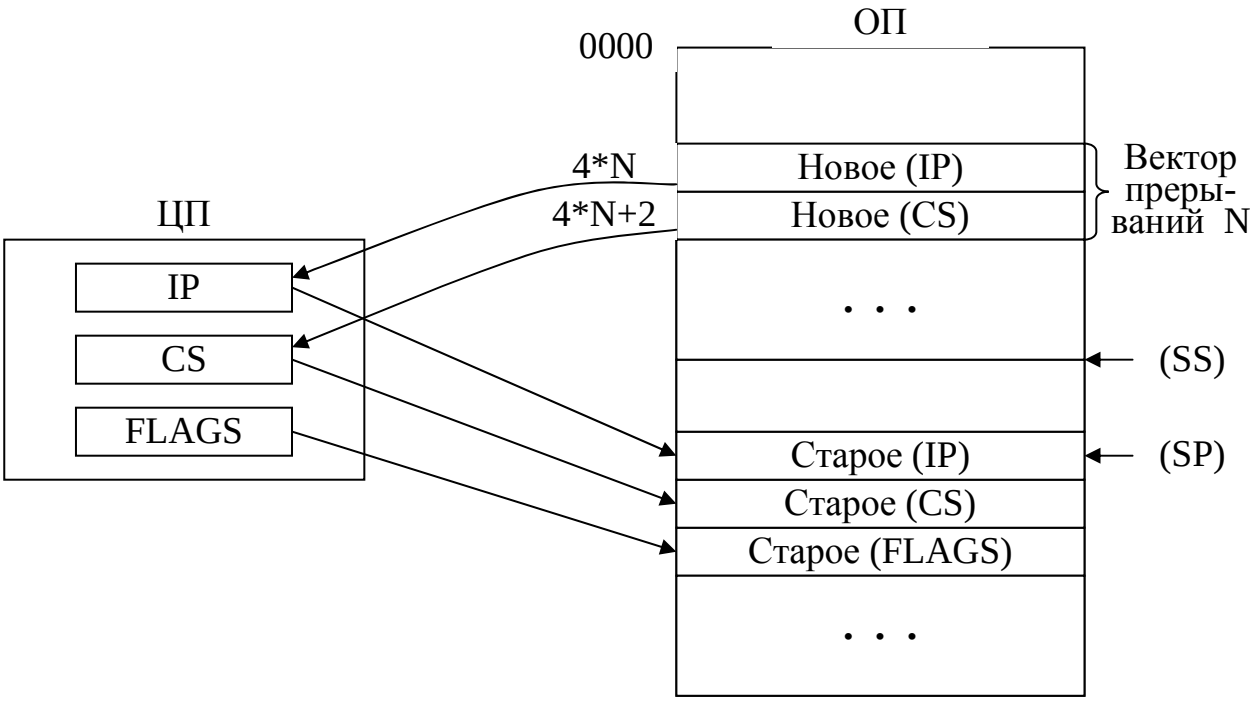


Рис.18. Перемещение содержимого регистров при прерывании с номером N

3) начальный этап программной обработки прерывания. Он выполняется обработчиком прерываний и включает действия:

– разрешение маскируемых прерываний с помощью команды STI. В результате важные маскируемые прерывания, например от таймера, не будут более откладываться и вызовут прерывание текущего обработчика прерываний точно так, как прерывается обыкновенная программа;

– сохранение в программном стеке содержимого тех регистров, с которыми будет работать программа обработчика;

– запись в сегментный регистр данных DS значения, которое соответствует адресу-сегменту данных обработчика прерываний. Это выполняется потому, что в результате прерывания из всех сегментных регистров только CS «переключается» на обработчик прерываний, а остальные сегментные регистры по-прежнему адресуют данные прерванной программы;

4) действия, определяемые типом прерывания. Эти действия обработчик прерываний выполняет сам, или он обращается за помощью к другим подпрограммам ОС, вызывая их как обычные процедуры командами CALL или используя для их вызова команды программного прерывания INT;

5) завершение обработки прерывания. Сюда относятся действия:

– восстановление содержимого регистров, запомненного ранее в стеке;

– возврат из прерывания. Его выполняет команда IRET, завершающая программу обработчика прерываний. При попадании этой команды на ЦП его аппаратура выталкивает из стека в регистры IP, CS и FLAGS прежнее их содержимое. В результате следующей выполняемой на ЦП командой будет очередная команда прерванной программы.

В дальнейшем мы не раз вернемся к данному алгоритму, используя его для написания программ обработчиков прерываний, а также для получения алгоритма выполнения маскируемых внешних прерываний (применение дополнительной аппаратуры потребует уточнения некоторых этапов рассмотренного алгоритма).

В заключение заметим, что среди трех типов прерываний (программные, исключения, внешние аппаратные) программные прерывания - наименее, а внешние аппаратные – наиболее приоритетные прерывания. Это означает, что обработчики программных прерываний могут прерываться сигналами исключений и сигналами внешних прерываний так, как прерываются обыкновенные прикладные программы. Обработчики исключений, в свою очередь, могут прерываться сигналами внешних аппаратных прерываний. Что касается обработчиков внешних прерываний, то, как будет показано в п.5.3, они могут прерываться сигналами других, более приоритетных внешних прерываний. (Чтобы такие прерывания были действительно возможны, прерываемые обработчики прерываний должны предварительно выполнить команду STI.)

3. ВЫПОЛНЕНИЕ ПРИКЛАДНЫХ ПРОГРАММ В СРЕДЕ MS-DOS

3.1. Получение прикладной программы

В п.1.1 отмечалось, что преобразование виртуальной прикладной программы, записанной на языке программирования, в реальную машинную программу выполняет системная обрабатывающая программа, называемая лингвистическим процессором. На самом деле для такого преобразования требуются три лингвистических процессора: транслятор, редактор связей и загрузчик. В принципе, их можно объединить в единую программу, которая, естественно, также будет лингвистическим процессором. Именно так устроены современные ***среды программирования***, позволяющие «выполнять» виртуальные программы. Кроме перечисленных программ, они могут включать также текстовый редактор для набора программ, а также отладчик. Вне зависимости от того, используются ли перечисленные лингвистические процессоры и утилиты по отдельности, или в виде модулей среды программирования, общая схема получения и выполнения прикладной программы остается прежней (рис.19).

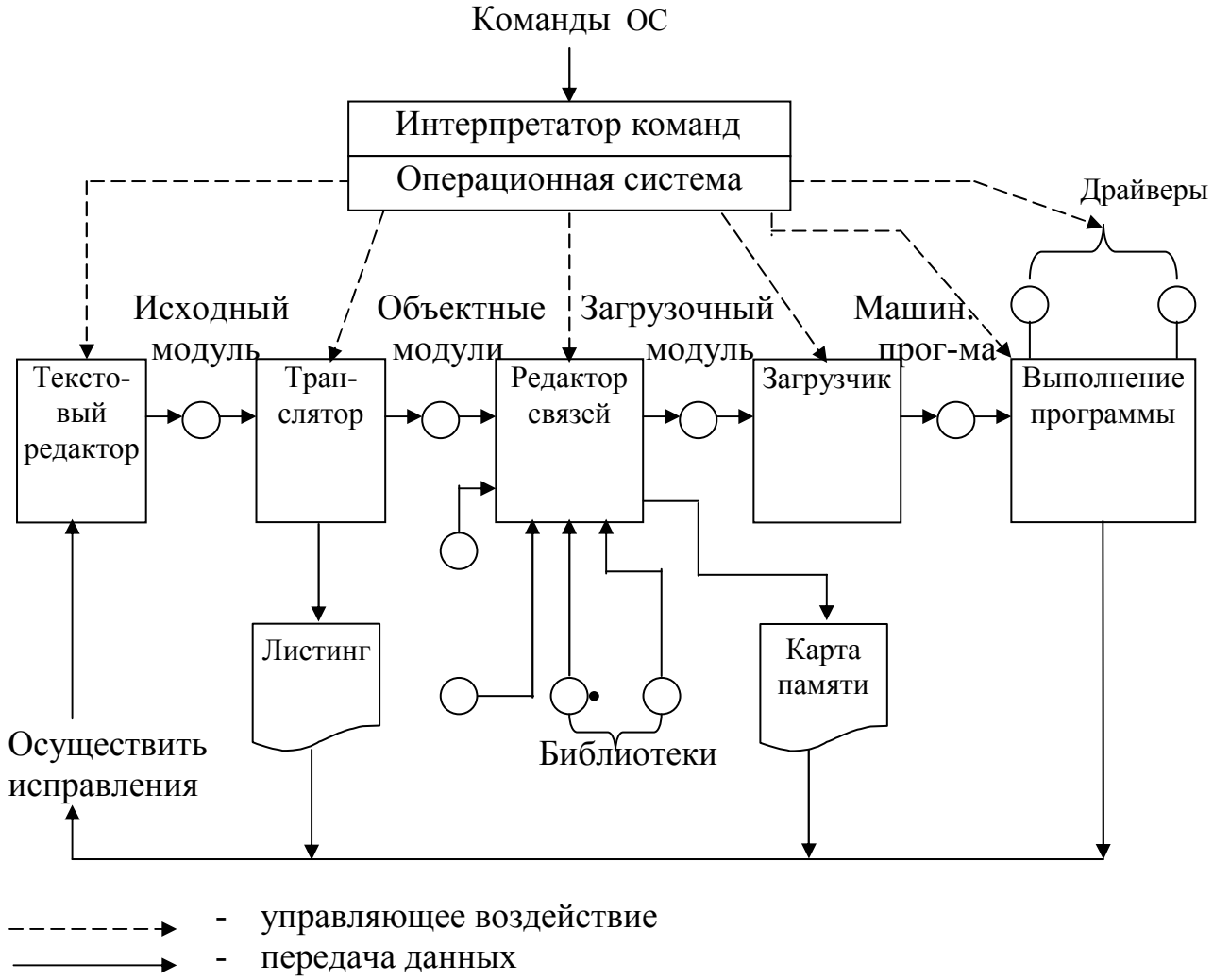


Рис.19. Общая схема создания и выполнения программы

На первом этапе в диалоге с текстовым редактором программист вводит в память ВС текст виртуальной программы, называемой также ***исходной программой***. Для удобства программиста исходная программа может быть записана в виде не одного, а нескольких текстовых файлов, называемых ***исходными модулями***. Программист сам определяет, как разбить свою программу на модули. Более того, он может программировать свои исходные модули на разных языках.

Исходный модуль представляет собой входные данные для транслятора того языка, на котором написана исходная программа. В результате одного выполнения транслятора исходный программный модуль преобразуется в ***объектный программный модуль***. При этом частично решается задача получения последовательности машинных команд, отображающих алгоритм модуля. Транслятор окончательно записывает коды операций машинных команд, а также проставляет в эти команды номера используемых регистров. Что касается символьных имен, то они обрабатываются транслятором по-разному.

Рассмотрим преобразование адресов транслятором. ***Адрес*** – место в ОП, которое займет соответствующий программный объект: команда или данное. Любой язык программирования позволяет программисту использовать в программе не адреса, а их заменители – ***символьные имена*** (***метки***). Основные типы меток: 1) метки операторов; 2) имена переменных; 3) имена процедур (имя процедуры заменяет для программиста адрес, по которому первая команда процедуры находится в ОП).

Все символьные имена в исходном модуле делятся на ***внешние*** и ***внутренние***. Символьное имя называется внутренним, если выполняются два условия: 1) соответствующий программный объект (оператор, данное или процедура) находится в этом исходном модуле; 2) данный программный объект используется (вызывается) только внутри данного исходного модуля.

Все внешние метки делятся на входные и выходные. ***Внешние выходные метки*** определены внутри данного исходного модуля, а используются вне его. Это: 1) имена процедур, которые входят в состав данного исходного модуля, но могут вызываться в других исходных модулях; 2) имена переменных, которые определены в данном исходном модуле, а используются вне его. ***Внешние входные метки*** определены вне данного исходного модуля, а используются в нем. Это: 1) имена процедур, не входящих в данный исходный модуль, но используемых в нем; 2) имена переменных, которые используются в исходном модуле, но определены вне его.

При получении объектного модуля вместо внутренних меток и внешних выходных меток транслятор проставляет в машинные команды, использующие соответствующие адреса, или смещение относительно текущего содержимого указателя команд IP, или смещение относительно на-

чала сегмента ОП, в котором находится соответствующий программный объект. В первом из этих вариантов расчет смещения относительно начала сегмента памяти производится в момент выполнения данной машинной команды на ЦП (см. п.2.2). Что касается внешних входных меток, то обработать их транслятор не может. Он ничего не знает о размещении соответствующих программных объектов в памяти, так как имеет в своем распоряжении единственный исходный модуль, в котором этих объектов нет. Дальнейшее преобразование программы выполняет системная программа, называемая редактором связей.

Редактор связей (компоновщик) связывает ("сшивает") все объектные модули программы в единый ***загрузочный модуль***. При получении загрузочного модуля редактор связей записывает объектные модули один за другим в ОП. Поэтому он "знает", где расположен в памяти каждый программный объект. Следовательно, он может заменить все внешние метки, оставленные транслятором, на соответствующие численные адреса (адреса-сегменты и адреса-смещения). В конце своей работы редактор связей записывает загрузочный модуль в файл на магнитном диске и выдает программисту сообщение, называемое ***картой памяти***. Эта карта описывает распределение памяти машинной программе.

В операционной системе MS-DOS допускается существование двух типов загрузочных модулей – типа com и типа exe (тип модуля совпадает с расширением имени файла, содержащего загрузочный модуль). Загрузочный модуль типа **com** применяется для создания небольших машинных программ (объемом не более 64K), а типа **exe** – для создания любых программ. Основное различие между этими загрузочными модулями заключается в том, что **com–файл** содержит готовую машинную программу, не требующую "доводки" при загрузке в память, а **exe–файл** содержит "заготовку" машинной программы, а также служебную информацию, используемую загрузчиком для настройки программы.

Как говорилось в п.1.2, пользователь осуществляет запуск прикладных и системных обрабатывающих программ с помощью интерпретатора команд ОС. (ИК для MS-DOS существует в виде отдельного загрузочного модуля Command.com.) В диалоге с ним пользователь сообщает имя загрузочного модуля требуемой программы. В ответ интерпретатор команд вызывает системную программу - ***загрузчик***, передав ему имя загрузочного модуля.

Загрузчик переписывает загрузочный модуль из ВП в ОП и, возможно, настраивает некоторые адреса в полях машинных команд. ***Настройка*** – изменение адреса в зависимости от фактического расположения машинной программы в ОП. Нетрудно предположить, что при загрузке exe-файла настройка производится, а для com-файла – не производится.

3.2. Структура прикладной программы

3.2.1. Префикс программного сегмента

Что касается машинной программы, записанной загрузчиком в ОП, то, кроме собственно команд программы и обрабатываемых ею данных, она содержит вспомогательную информацию, называемую ***префиксом программного сегмента*** (**PSP**)**.** Длина PSP 256 (100h) байтов, и эта структура данных находится в начале любой машинной программы, в независимости от того, получена ли программа загрузкой com- или exe-файла. Информация в PSP используется операционной системой при выполнении запросов со стороны программы, а также может быть использована самой программой. Структура PSP приведена на рис.20.

Относит. адрес	
00h	Команда INT 20h
02h	Верхняя граница памяти программы
04h	Зарезервировано
05h	Вызов диспетчера функций
0Ah	Векторы прерываний 22h – 24h
16h	Адрес-сегмент PSP родительской программы
18h	Таблица логич. номеров файлов данных
2Ch	Адрес-сегмент блока окружения
2Eh	Зарезервировано
5Ch	2 блока управления файлами
80h	Хвост команды

Рис.20. Структура PSP

Для наглядного знакомства со структурой PSP удобно использовать отладчик Debug. Допустим, что мы ранее получили загрузочный модуль программы с именем Prob.com. Тогда для анализа PSP вызовем Debug следующей командой MS-DOS:

Debug Prob.com parametr,

где вместо слова parametr может быть записана любая символьная строка (в коде ASCII), содержащая входные параметры нашей прикладной программы. Эти параметры совершенно не интересуют ни Debug, ни тем более MS-DOS, а используются лишь для передачи в программу какой-то исходной информации. (Кстати, если рассматривать в качестве программы сам Debug, то в качестве его параметра выступает имя исследуемой программы, то есть Prob.com.)

После того как мы запустили Debug описанным выше способом, с помощью команды Debug – **d 0** получим на экране дамп начальной части PSP. В результате на экран будут выведены первые 128 байтов PSP. Для вывода на экран каждых следующих 128 байтов достаточно просто ввести команду d.

В первых двух байтах PSP находится код машинной команды INT 20h. Чтобы убедиться в этом, введем команду Debug – **u 0**. Команда программного прерывания выполняет возврат в ОС при завершении программы. Сама же эта команда получает управление следующим образом: во-первых, сразу после загрузки программы в память загрузчик помещает в ее стек 16-битное нулевое слово (0000h). Во-вторых, если в конце главной подпрограммы прикладной программы стоит команда RET, а тип главной подпрограммы – NEAR, то при попадании RET на ЦП в качестве адреса возврата из стека будет выбран 0 и, следовательно, следующей исполняемой командой будет INT 20h. Следует отметить, что это не единственный способ возвращения из программы в ОС. Например, вместо RET в главной подпрограмме можно записать оператор INT 20h. В этом случае управление из программы возвращается в ОС непосредственно, минуя PSP. Другой способ возврата – запись в конце главной подпрограммы команды INT 21h (функция 4Ch). Этот способ является наиболее предпочтительным, так как он позволяет возвращать из программы в ОС код возврата (в регистре AL), информирующий операционную систему о причине завершения программы.

В следующих двух байтах PSP находится верхняя граница (адрес-сегмент) ОП. Обычно она равна величине A000h. Мы еще вернемся к этой величине, когда в дальнейшем будем рассматривать распределение ОП.

В пяти байтах PSP, начиная с 05h, находится вызов диспетчера функций MS-DOS. Данный вызов представляет собой команду дальнего вызова процедуры. С помощью команды Debug – **u 5** мы можем получить на экране, например, следующее:

CALL F01D:FEF0 ,

где F01D – адрес-сегмент, а FEF0 – адрес-смещение первой команды диспетчера функций MS-DOS. (Соответствующий реальный адрес интересен тем, что он находится за пределами первого мегабайта. Мы вернемся к нему в вопросе о распределении памяти.)

Для того чтобы обратиться из программы к MS-DOS (с целью получения помощи), достаточно поместить в нее команду CALL 5. В результате следующей командой, исполняемой на ЦП, будет команда вызова диспетчера функций. (Данный способ обращения к MS-DOS менее предпочтителен по сравнению с командой программного прерывания INT 21h.)

В двенадцати байтах PSP, начиная с 0Ah, содержатся копии векторов прерываний с номерами 22h, 23h, 24h. Эти прерывания MS-DOS использует для управления выполнением программы. При этом прерывание 22h используется для вызова той подпрограммы, которая должна получить управление при завершении прикладной программы. Прерывание 23h происходит при нажатии клавиш <Ctrl>&<C>. (MS-DOS использует для обработки этой комбинации специальный обработчик.) Причинами прерывания 24h являются аппаратные ошибки при работе с ПУ. Некоторые из этих ошибок: 1) попытка записи на защищенный диск; 2) нет бумаги на принтере; 3) неизвестное устройство. Хранение копий векторов перечисленных прерываний обусловлено тем, что в случае, если прикладная программа изменяет содержимое векторов (с целью выполнить свою обработку прерываний), то после ее завершения MS-DOS восстанавливает прежнее содержимое векторов, используя копии из PSP.

По адресу 16h находится адрес-сегмент PSP родительской программы. Назначение этого поля будет понятно из п.3.4. Следующее поле PSP начинается по адресу 18h и имеет длину 20 байтов. В нем размещена таблица логических номеров файлов, которая будет рассмотрена в следующей главе.

Начиная с ячейки 2Ch два байта PSP содержат адрес-сегмент блока окружения программы. ***Блок окружения*** – совокупность переменных окружения. ***Переменная окружения*** – символьная строка в коде ASCII, имеющая структуру:

<имя переменной> = <первое значение>; … ;<последнее значение>00h

После последней переменной окружения записывается не один, а два нулевых байта (0000h). Пример переменной окружения:

PATH = C:\WINDOWS; C:\WINDOWS\COMMAND00h

Данная переменная задает те каталоги (директории), в которых могут находиться используемые программой файлы. В том случае, если на вход программы поступает не имя-путь файла, а лишь завершающая часть этого имени, то для получения имени-пути программа должна последовательно просмотреть заданные значения переменной окружения PATH. В случае обнаружения в очередном каталоге искомого файла имя-путь каталога соединяется с именем файла в единое имя-путь файла.

Допустим, что с помощью команды Debug – **d 0** мы прочитали значение адреса-сегмента блока окружения, равное 2065. Тогда для получения на экране содержимого блока окружения следует воспользоваться командой **d 2065:0** . Первоначальное содержимое своего блока окружения программа получает «в наследство» от той программы, которая создает дан-

ную программу, используя системный вызов **EXEC** (INT 21h, функция 4Bh). Данный вызов будет рассматриваться позже. А сейчас лишь заметим, что программа может не только читать свои переменные окружения, но и вносить в них изменения.

По адресу 5Ch находятся два блока управления файлами, используемые при работе с линейными файловыми структурами. Так как эти структуры сейчас полностью вытеснены иерархическими файловыми структурами, соответствующее поле PSP может использоваться для хранения любой другой информации. Например, учитывая, что, подобно блоку окружения данные блоки управления передаются «по наследству», родительская программа может помещать в них исходные данные для дочерней программы. Другой способ передачи исходных данных в программу заключается в использовании «хвоста» команды.

Хвост команды – остаток команды ОС (после имени программы), запустившей прикладную программу. Иными словами, хвост содержит параметры команды, а также пробелы, которые были набраны в командной строке. В приведенном выше примере хвостом является слово parametr, предваряемое одним пробелом. Для размещения хвоста используется поле в PSP, начиная с ячейки 81h. Длина хвоста содержится в байте 80h. (Убедитесь в этом, используя Debug.)

Пример. Следующая простая com-программа выводит на экран свои параметры (хвост команды).

```
  ;
  ;          Программа выводит на экран свои параметры
  ;          ----------------------------------------------------------
  ;
                    ASSUME      CS:_Text
_Text  SEGMENT      PUBLIC      'CODE'
                    ORG   80h                   ; Следующий байт имеет адрес-смещение 80h
Lparam              DB    ?                     ; Длина хвоста команды
Param               DB    ?                     ; Первый байт хвоста
                    ORG   100h
Start:              XOR   CX, CX                ; Обнуление CX
                    MOV   CL, Lparam            ;  CL ← Длина хвоста
                    CMP   CX, 0                 ; Хвост отсутствует ?
                    JZ    Exit                  ; Если да
                    MOV   AH, 0Eh               ;  0Eh – функция вывода символа
                    MOV   BH, 0                 ; Нулевая страница экрана
                    XOR    SI, SI               ;  SI ← 0
 Next:              MOV   AL, Param[SI]         ;  Вывод следующего
                    INT   10h                   ;    символа
                    INC    SI                   ;
                    LOOP Next                   ;  Повторить для след. символа
  Exit:             MOV   AX, 4C00h             ;  Возврат в DOS с
                    INT    21h                  ;    кодом завершения 0
_Text      ENDS
END                 Start
```

В данной программе для вывода символа на экран используется системный вызов BIOS – **INT 10h** (функция **0Eh**). Предварительно программа помещает код выводимого символа в регистр AL.

3.2.2. Программа типа com

Такая программа получается в результате загрузки com-файла, которая сводится к простому переписыванию содержимого этого файла с диска в ОП.

На рис.21 приведено содержимое сегментных регистров и регистров-указателей IP и SP после загрузки com-программы в момент передачи ей управления. Собственно передача управления заключается в записи в CS и IP тех значений, которые соответствуют первой исполняемой команде программы. В этот момент все сегментные регистры содержат одно и тоже – начальный адрес (а точнее – номер начального параграфа) области ОП, в которую загружена программа. Так как программа начинается с PSP, то следовательно, номер начального параграфа PSP и содержится в сегментных регистрах CS, DS, ES и SS.

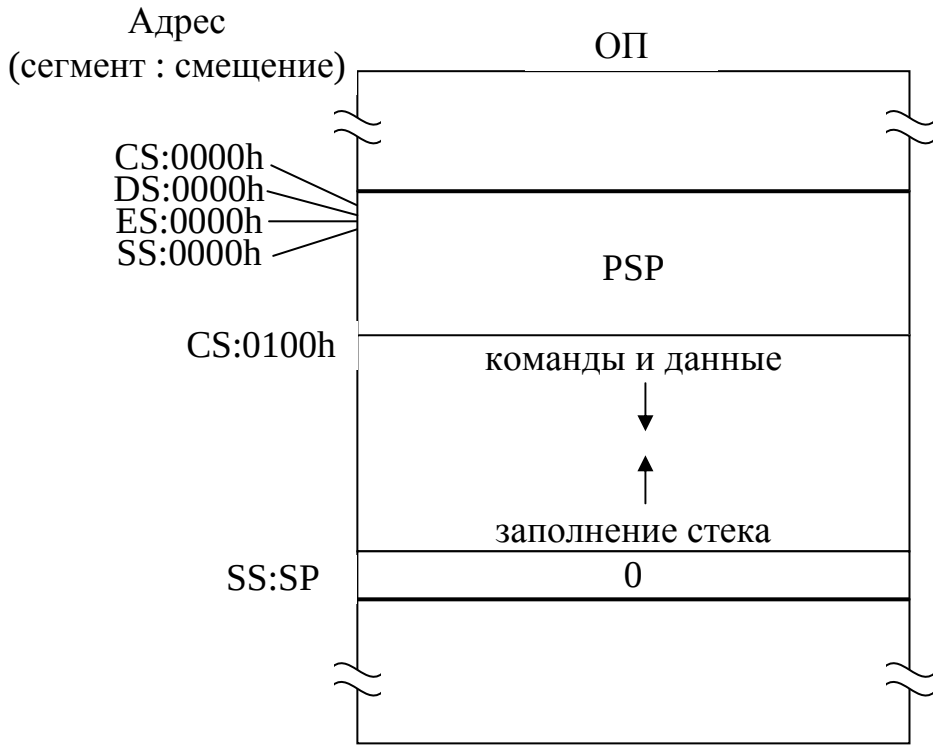


Рис.21. Результат загрузки com-программы

Первоначальное содержимое указателя команд IP задает адрес-смещение первой исполняемой команды программы. Для com-программ это всегда число 100h, так как первая исполняемая команда начинается сразу же после завершения PSP. Что касается указателя стека SP, то он со-

держит адрес-смещение вершины стека, в который при загрузке программы уже было помещено одно нулевое слово. По мере того, как в стек будут записываться новые слова данных, его вершина будет приближаться к области памяти, занимаемой командами и данными программы. В случае наложения стека на другие данные программы дальнейшее ее продолжение будет или ошибочным, или вообще невозможным.

Исходная программа на ассемблере, ориентированная на получение com-программы, имеет следующие особенности:

1) первому исполнительному оператору программы предшествует псевдооператор **ORG 100h**, который обеспечивает требуемое смещение соответствующей машинной команды относительно начала программы;

2) операнд псевдооператора END в конце исходной программы в качестве точки входа задает первый исполнительный оператор программы;

3) исходная программа не имеет виртуального сегмента стека. Отсутствие такого сегмента не означает, что в машинной программе не будет стека (любая машинная программа обязательно имеет стек), а лишь говорит о том, что стек будет создан автоматически, то есть без участия программиста;

4) исходная программа не должна содержать исполнительных операторов, выполняющих действия с сегментными адресами. (Такие адреса должны настраиваться загрузчиком в зависимости от размещения машинной программы в ОП, а настройка com-файла не производится.) Например, недопустимы следующие операторы:

```
MOV     AX, _Data              ; _Data – сегмент данных
MOV     SI, SEG  Buff          ; SI ← адрес-сегмент переменной Buff
```

Пример исходной программы, ориентированной на получение загрузочного модуля типа com:

```
;       Программа выводит сообщение “Здравствуйте” на экран
;       ---------------------------------------------------------------------
;
cr      EQU   0Dh                       ; Код ASCII возврата каретки
lf      EQU   0Ah                       ; Код ASCII перевода строки
        ASSUME  CS:_Text
_Text   SEGMENT  PUBLIC  ‘CODE’
        ORG   100h
Start:  JMP   Begin                     ; Переход через данные
Msg     DB    ‘Здравствуйте’, cr, lf, ‘$’   ; Выводимое сообщение
Begin:  MOV   DX, OFFSET Msg            ; DX ← адрес сообщения
        MOV   AH, 9                     ; Вывод строки
        INT   21h                       ;   на экран
        MOV   AX, 4C00h                 ; Возврат в DOS с
        INT   21h                       ;   кодом завершения 0
_Text   ENDS
END   Start
```

Эта исходная программа включает единственный виртуальный сегмент – сегмент кода _Text. Данные, обрабатываемые программой, размещены в сегменте _Text так, чтобы они не могли попасть на ЦП в качестве исполняемых команд, что неизбежно привело бы к сбою в работе процессора. Считается хорошим тоном при написании com-программы на ассемблере располагать ее данные в начале программы, так как это упрощает трансляцию. Первый исполнительный оператор JMP «перепрыгивает» через эти данные на исполнительную часть программы.

Для вывода строки на экран программа использует системный вызов **INT 21h** (функция **9**). Перед применением вызова программа помещает в регистры DS и DX соответственно адрес-сегмент и адрес-смещение для младшего байта выводимой строки. (В нашей программе DS уже содержит требуемый адрес и поэтому не загружается.) Выводимая строка обязательно должна заканчиваться символом “$”. Сама строка может содержать любые другие символы, в том числе и управляющие. В программе используются управляющие символы 0Dh (возврат каретки) и 0Ah (перевод строки).

3.2.3. Программа типа exe

В отличие от com-программы, загружаемой всегда в один сегмент ОП (объемом 64K), программа типа exe может занимать несколько таких сегментов. Следствием этого является наличие в программе команд, выполняющих запись в регистры адресов-сегментов, требующих настройки во время загрузки.

В результате получение exe-программы требует не простого копирования соответствующего файла в ОП, а предполагает преобразование этого файла загрузчиком. В результате такого преобразования, во-первых, записывается PSP программы, а во-вторых, настраиваются некоторые адреса в полях машинных команд. Наличие первой из этих функций обусловлено тем, что преобразуемый exe-файл вообще не содержит PSP. Вместо него в начале этого файла находится ***заголовок***, размер которого кратен 512 байтам, содержащий управляющую информацию, используемую загрузчиком для получения exe-программы.

Свою работу по созданию exe-файла редактор связей начинает с того, что последовательно записывает в свою память код (команды), данные и стек будущей программы. Порядок расположения этих частей программы, в отличие от com-файла, может быть любым. Их содержимое редактор связей считывает из доступных ему объектных модулей программы. Выполняя размещение программы в памяти, редактор связей обращается с ее адресами так, как будто программа загружена в ОП начиная с условного адреса 00000h. Поэтому считается, что часть программы (код, данные или стек), размещаемая первой, имеет начальный логический адрес

0000h:0000h. Если, например, вторая часть программы начинается с условного реального адреса 000E7h, то ему соответствует начальный логический адрес 000Eh:0007h. Применение таких условных адресов позволяет редактору связей выполнить запись численных значений для всех внешних адресов, оставшихся не проставленными после трансляции.

Исключение составляют команды, выполняющие запись в регистры значений адресов-сегментов. К таким командам относятся, во-первых, команды дальнего перехода JMP и CALL (они выполняют запись адреса-сегмента в регистр CS), а во-вторых, команды MOV, выполняющие запись адресов-сегментов в регистры данных (из регистров данных другие команды MOV переписывают значения в регистры сегментов DS, ES или SS). Например, следующие два оператора исходной программы выполняют запись в регистр DS того адреса-сегмента, который соответствует точке исходной программы с меткой Msg:

```
MOV    DX, SEG  Msg
MOV    DS, DX
```

Первый оператор MOV транслируется в трех-, а второй – в двухбайтовую машинную команду. При этом второй и третий байты первой команды используются для размещения адреса-сегмента, загружаемого в регистр DX.

Так как адреса-сегменты зависят от фактического размещения будущей программы в ОП, которое редактор связей «не знает», то он не может проставить численные значения адресов-сегментов в поля машинных команд. Тем не менее он выполняет значительную подготовку для будущей простановки этих значений загрузчиком. Для этого, во-первых, редактор связей помещает в эти поля условные адреса-сегменты, полученные в предположении, что программа загружается в память начиная с нулевого адреса.

Во-вторых, редактор связей создает ***таблицу настройки***, занимающую почти весь объем заголовка exe-файла. Эта таблица состоит из 4-х байтовых полей, каждое из которых соответствует одному адресу-сегменту, используемому какой-то командой программы и требующему настройки в зависимости от фактического размещения программы в памяти. В качестве содержимого этого поля редактор связей записывает условный логический адрес (сегмент и смещение) той ячейки (двух байтов) программы, которая содержит адрес-сегмент, требующий настройки.

Сама настройка адресов-сегментов программы производится загрузчиком сразу же после размещения программы (в том числе и PSP) в ОП. Для этого загрузчик последовательно просматривает таблицу настройки, считывая из нее указатель на очередную последовательность из двух байтов программы, требующую настройки. А затем он прибавляет к их содер-

жимому (то есть к условному адресу-сегменту) номер того параграфа ОП, начиная с которого программа загружена фактически.

После того как программа загружена, загрузчик передает ей управление. На рис.22 приведено содержимое сегментных регистров и указателя стека SP в момент передачи управления exe-программе. В этот момент сегментные регистры данных содержат номер начального параграфа PSP. Благодаря этому программа может использовать полезную информацию, содержащуюся в PSP. (Запомнив где-то первоначальное значение этих регистров, программа обычно записывает в них новое значение, указывающее на начало области данных программы.) Что касается CS, то в него загрузчик помещает начальный номер параграфа не PSP, а области кода программы. Аналогично содержимое SS указывает на границу области стека программы.

Исходная программа на ассемблере, ориентированная на получение exe-программы, обязательно имеет, кроме виртуальных сегментов кода и данных, виртуальный сегмент стека. Порядок записи этих сегментов в программе может быть различным. В качестве точки входа в программу (операнд оператора END) может быть задан любой исполнительный оператор программы.

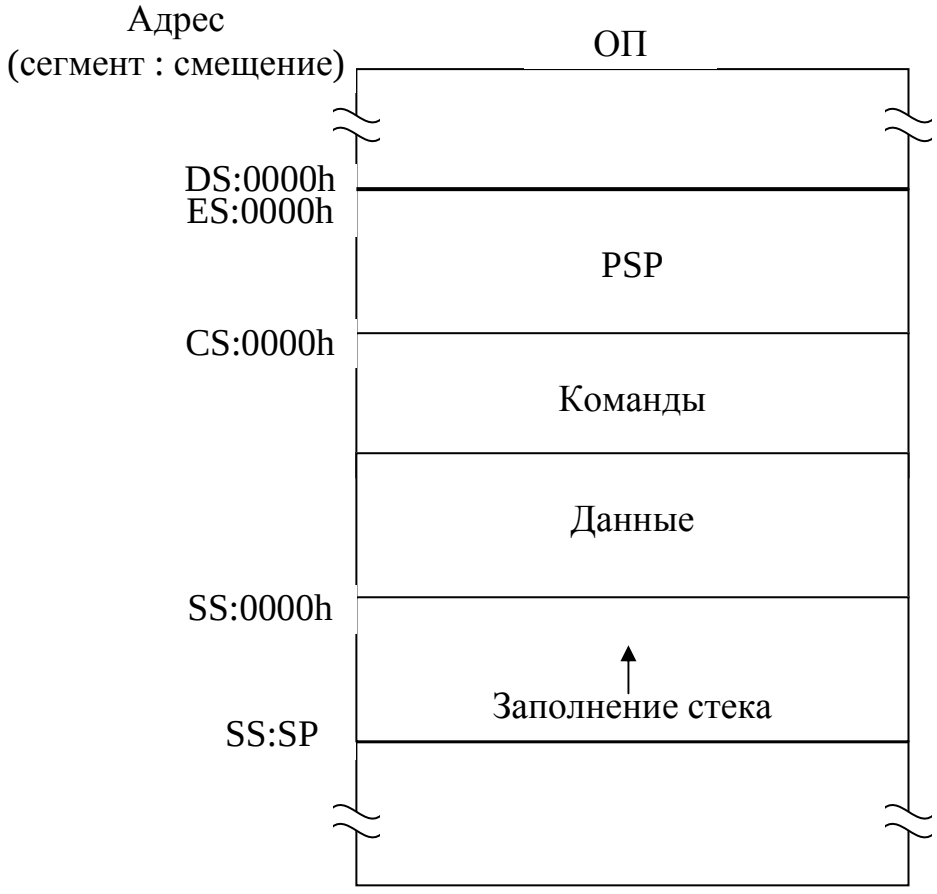


Рис.22. Результат загрузки exe-программы

Пример. Следующая exe-программа выполняет то же самое, что и предыдущая com-программа.

```
  ;        Программа выводит сообщение “Здравствуйте” на экран
  ;        ---------------------------------------------------------------------
  ;
cr      EQU   0Dh                                ; Код ASCII возврата каретки
lf      EQU   0Ah                                ; Код ASCII перевода строки
        ASSUME   CS:_Text, DS:_Data, SS:_Stack

_Text    SEGMENT  PUBLIC  ‘CODE’                 ; Сегмент кода
Start:  MOV  AX, _Data                           ; Сделаем сегмент
        MOV  DS, AX                              ;   данных адресуемым
        MOV  DX, OFFSET Msg                      ; DX ← адрес сообщения
        MOV  AH, 9                               ; Вывод строки
        INT    21h                               ;   на экран
        MOV  AX, 4C00h                           ; Возврат в DOS с
        INT     21h                              ;   кодом завершения 0
_Text     ENDS

 _Data    SEGMENT  PUBLIC  ‘DATA’                ; Сегмент данных
        Msg   DB  ‘Здравствуйте’, cr, lf, ‘$’    ; Выводимая строка
_Data    ENDS

_Stack   SEGMENT  STACK  ‘STACK’                 ; Сегмент стека
        DB    20  DUP (?)
_Stack   ENDS
END    Start
```

В данной исходной программе три виртуальных сегмента – сегмент кода _Text, сегмент данных _Data и сегмент стека _Stack. Обратите внимание, что в начале программы в регистр DS загружается адрес-сегмент, соответствующий началу данных программы. Это делается для того, чтобы фактическое размещение данных в памяти никак не зависило от размещения в ней PSP. (Для com-программы адрес-смещение данных вычисляется относительно начала PSP.)

3.3. Распределение памяти

Оперативная память – ресурс, требуемый для выполнения любой программы. В рассматриваемой ВС (MS-DOS + i8086) объем адресуемого пространства ОП составляет 1 Мбайт. Только часть этого объема (640К) соответствует реальной ОП (рис.23). Остальная часть адресов используется для работы с другими видами памяти (ПЗУ и видеопамять). Благодаря общему адресному пространству работа с этими видами памяти выполняется командами ЦП аналогично ОП (за исключением того, что в ПЗУ нельзя записывать).

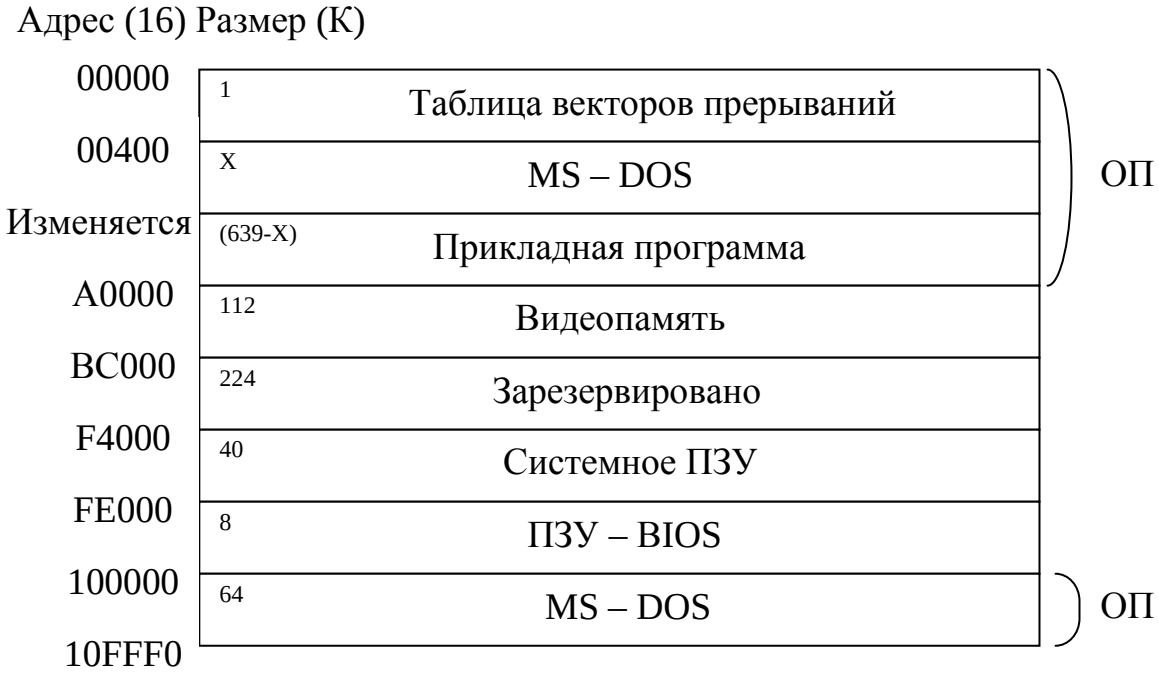


Рис.23. Распределение адресного пространства ОП

Как видно из рис.23, общий объем адресуемой памяти составляет не ровно 1Мбайт, а немного больше (почти на 64К). Дело в том, что если поместить в регистр сегмента предельно большое значение (FFFFh), то за счет содержимого регистра-смещения можно превысить число FFFFFh, которое является номером старшего байта в первом мегабайте памяти. Предельная величина адреса получается суммированием чисел FFFF0h и FFFFh и равна 10FFEFh. Адреса за пределами 1 Мбайта соответствуют реальной ОП, которую MS-DOS обычно использует для своих дополнительных нужд. Основная часть ОП занимаемой ОС находится в начале адресного пространства.

Что касается прикладной программы, то ей может быть распределена вся память, оставшаяся от 640К после размещения MS-DOS. Распределение памяти программе, сделанное до начала ее выполнения, называется ***статическим распределением памяти***, а во время ее выполнения – ***динамическим распределением***.

Статическое распределение памяти для exe- и com-программ выполняется MS-DOS различно. Несмотря на то, что для размещения com-программы требуется всего один блок памяти в 64К, ОС распределяет ей всю память, свободную на момент создания программы. Для exe-программы MS-DOS статически распределяет лишь первоначально необходимый объем ОП, определяемый длиной программы (включая PSP).

Что касается динамического распределения памяти, то может возникнуть вопрос о том, зачем оно вообще нужно в однопрограммной системе, каковой является MS-DOS. Первой причиной его применения является то, что даже при выполнении единственной прикладной программы ее по-

требности в памяти для хранения своих данных могут превышать объем памяти, которая может быть предоставлена программе. Поэтому программа может выполнять освобождение областей памяти, занимаемых одними данными, для того, чтобы позже запрашивать области памяти для других своих данных.

Вторая причина применения динамического распределения состоит в том, что MS-DOS имеет зачатки мультипрограммных операционных систем, позволяя нескольким прикладным программам одновременно находиться в ОП. Это происходит, во-первых, тогда, когда одна прикладная программа запускается по запросу другой прикладной программы. А во-вторых, прикладная программа может выполнять какие-то системные функции, дублируя или дополняя подпрограммы ОС, и поэтому должна находиться в памяти и во время выполнения других прикладных программ. Подобные запуски прикладных программ будут рассмотрены в следующих подразделах, а пока лишь заметим, что для их реализации требуется, чтобы одна прикладная программа добровольно отказывалась бы от своей неиспользуемой памяти с целью обеспечения будущей загрузки другой программы (программ).

Динамическое распределение памяти выполняется MS-DOS по запросу самой прикладной программы. Для этого прикладная программа передает в ОС один из трех системных вызовов, рассматриваемых далее.

INT 21h (функция **48h)** – выдать программе область памяти требуемого размера. Для передачи размера области, измеряемого в параграфах, используется регистр BX. В случае успеха MS-DOS передает в регистре AX адрес-сегмент выделенной области памяти. Иначе – устанавливает флаг переноса FC=1.

Пример. Для создания своего буфера программа запрашивает область памяти длиной 500h параграфов:

```
          . . . . . .
Bufseg    DW        ?          ; Адрес-сегмент новой области
          . . . . . .
          MOV       AH, 48h    ; Номер функции
          MOV       BX, 500h   ; Размер области (в параграфах)
          INT       21h        ; Запрос памяти
          JC        Error      ; Переход при ошибке
          MOV       Bufseg, AX ; Сохранить адрес-сегмент области
          . . . . . .
```

INT 21h (функция **49h**) - освободить область памяти, выделенную ранее динамически программе с помощью системного вызова INT 21h (функция 48h). Адрес-сегмент области программа передает в регистре ES. В случае ошибки MS-DOS устанавливает флаг переноса FC=1.

Пример. Программа освобождает область памяти, адрес-сегмент которой находится в слове Bufseg:

```
                . . . . . .
Bufseg          DW          ?               ; Адрес-сегмент области
                . . . . . .
                MOV         AH, 49h         ; Номер функции
                MOV         ES, Bufseg      ; Адрес-сегмент области
                INT         21h             ; Освобождение памяти
                JC          Error           ; Переход при ошибке
                . . . . . .
```

INT 21h (функция **4Ah**) – изменить (увеличить или уменьшить) область памяти, принадлежащую программе. В регистре ES программа передает адрес-сегмент модифицируемой области, а в регистре BX – новый размер области памяти в параграфах. В случае ошибки MS-DOS устанавливает флаг переноса FC=1.

Пример. Программа просит уменьшить размер области, выделенной ранее (в примере для функции 48h) до 200h параграфов (8К):

```
                . . . . . .
Bufseg          DW          ?               ; Адрес-сегмент области
                . . . . . .
                MOV         AH, 4Ah         ; Номер функции
                MOV         BX, 200h        ; Новый размер области
                MOV         ES, Bufseg      ; Адрес-сегмент области
                INT         21h             ; Изменить размер области
                JC          Error           ; Переход при ошибке
                . . . . . .
```

Перейдем к рассмотрению принципов реализации динамического распределения памяти в MS-DOS. Во-первых, вся ОП в системе разбита на области неодинаковой длины. Это разбиение со временем может изменяться, так как одни области могут разбиваться, а другие (соседние), наоборот, могут соединяться. Любая прикладная программа занимает минимум две области ОП. В одной из них находится сама программа (PSP, код, данные и стек), а во второй области находится блок окружения программы (размер этой области кратен 512 байтам). Число областей ОП, занимаемых программой, увеличивается в результате системных вызовов INT 21h (функция 48h).

Во-вторых, каждая область памяти начинается с 16-байтового (один параграф) ***блока управления области***, который содержит:

1) в байте 0 – символ **M**, если область не последняя; **Z**, если область последняя в ОП;
2) в байтах 1 и 2 – адрес-сегмент PSP программы, являющейся владельцем данной области ОП. Если область свободна, то 0000h;
3) в байтах 3 и 4 – длина области в параграфах;
4) байты 5-15 – зарезервированы.

В-третьих, динамический запрос программы на получение дополнительной памяти MS-DOS выполняет одним из трех способов:

1) двигаясь в сторону больших адресов, последовательно просматривает блоки управления областей до тех пор, пока не встретится область памяти, не меньшая требуемой. Начальный адрес-сегмент этой области возвращается в программу, а ее излишек оформляется в новую свободную область. При отсутствии требуемой области устанавливается флаг FC;

2) последовательно просматриваются все свободные области памяти с целью нахождения минимальной области, которая не меньше требуемой;

3) отличается от способа 1 тем, что просмотр областей производится от больших адресов к меньшим.

Обычно MS-DOS ищет требуемую область, пользуясь первым способом. Но с помощью специального системного вызова программа может запросить у ОС другой алгоритм поиска.

Следует отметить, что при выполнении системных вызовов распределения памяти MS-DOS «попутно» проверяет целостность всей цепочки блоков управления памятью. Поэтому нельзя портить содержимое этих блоков. Иначе ОС завершит работу системы, выдав соответствующее сообщение на экран.

В последующих разделах будут приведены примеры программ, содержащих системные вызовы для динамического распределения памяти.

3.4. Запуск прикладных программ

Работа интерпретатора команд MS-DOS основана на использовании системного вызова **EXEC - INT 21h** (функция **4Bh**, подфункция **00h**). Именно этот вызов обеспечивает загрузку заданного com- или exe-файла, его настройку (для exe-файла), выполнение полученной машинной программы, а затем возврат в ту точку программы ИК, откуда был сделан этот системный вызов.

Применив вызов EXEC в своей прикладной программе, мы сможем запустить (загрузить и выполнить) любую другую прикладную или системную обрабатывающую программу. В результате наша прикладная программа начнет выполнять некоторые функции ИК, а при некоторой доработке сможет даже заменить его. В этом нет ничего удивительного, так как основная часть ОС (ядро ОС) рассматривает ИК как обычную обрабатывающую программу (лингвистический процессор). Коль скоро вызов EXEC так важен, то внимательно рассмотрим его.

Во-первых, обратим внимание на то, что отношение между «родительской» и «дочерней» программами логически эквивалентно отношению между программой и подпрограммой, вызываемой с помощью команды CALL или команды INT. Подобно подпрограмме, дочерняя программа яв-

ляется по отношению к родительской программе логической процедурой (см. прил.1), так как на время выполнения дочерней программы выполнение родительской программы приостанавливается, а затем продолжается с той команды, которая расположена в программе сразу же после вызова EXEC.

Во-вторых, подобно обычным программным процедурам, дочерние программы могут быть вложенными. То есть дочерняя программа сама может содержать вызов EXEC, порождающий другую программу, и так далее.

В-третьих, прежде, чем выдавать вызов EXEC, родительская программа должна позаботиться о том, чтобы имелась в наличии свободная память, достаточная для размещения дочерней программы. С этой целью родительская программа должна освободиться от излишков памяти, ранее выделенной ей, используя системные вызовы, рассмотренные в п.3.3.

В-четвертых, запуск программы с помощью вызова EXEC предполагает наличие между родительской и дочерней программами не только управляющего, но и информационного взаимодействия. При этом информация может быть передана как в прямом направлении – от родительской программы к дочерней, так и в обратном направлении.

Передача информации к дочерней программе выполняется с помощью ее PSP, который создается при выполнении вызова EXEC. Если информация небольшая, то она может быть размещена в «хвосте». Большой объем информации (до 32 К) может быть размещен в блоке окружения программы. Кроме того, для передачи данных на вход дочерней программы могут использоваться два поля PSP, предназначенные для хранения устаревших типов блоков управления файлами.

Для того, чтобы выполнение вызова EXEC привело к созданию требуемого PSP, в момент вызова регистры должны содержать следующие данные:

DS : DX – (адрес-сегмент : адрес-смещение) символьной строки, содержащей имя com- или exe-файла, подлежащего загрузке. Сразу же после имени файла должен находиться нулевой байт (00h);

ES : BX - (адрес-сегмент : адрес-смещение) блока параметров.

Блок параметров содержит:

1) 2 байта - адрес-сегмент блока окружения. Если эти байты содержат 0, то дочерняя программа получает копию блока окружения родительской программы;

2) 4 байта - адрес-смещение и адрес-сегмент символьной строки, содержащей «хвост» команды. Эта строка заканчивается символом «возврат каретки» (0Dh), который не учитывается при подсчете длины «хвоста»;

3) 8 байтов - адрес-смещение и адрес-сегмент двух блоков управления файлами (по 4 байта).

В простейшем случае, когда ни «хвост», ни блоки управления файлами не используются для передачи информации на вход дочерней программы, а блок окружения такой же, как и у родительской программы, блок параметров может быть оформлен в программе как последовательность 14-и байтов, заполненных нулями.

В случае ошибочного завершения вызова EXEC в родительскую программу возвращается флаг FC=1, а в регистре AX возвращается код ошибки.

В случае успешного запуска и выполнения дочерней программы, она может передать в родительскую программу любой объем информации, используя для этого ту область ОП, адрес которой был получен ей ранее в составе другой информации, переданной от родительской программы.

Пример. Следующая программа выполняет роль простейшего интерпретатора команд MS-DOS. Она запрашивает у пользователя имя файла, содержащего загрузочный модуль требуемой программы, а затем эту программу запускает. При этом могут запускаться только такие программы, которые не требуют от родительской программы каких-то входных данных. После запуска дочерней программы (успешного или неуспешного) на экран опять выводится приглашение ввести имя файла. Для прекращения работы программы достаточно вместо набора имени файла нажать клавишу <Enter>.

```
;          Программа запускает другие программы
;          ----------------------------------------------------
;
                ASSUME      CS:_Text
_Text  SEGMENT     PUBLIC       'CODE'
                ORG   100h                  ; Следующий байт имеет адрес-смещение 100h
Start:          JMP   Begin
;       Данные программы
Buff            DB    81                    ; Максимальная длина имени файла
Lname           DB    0                     ; Фактическая длина имени файла
Namefile        DB    81      DUP (0)       ; Здесь размещается имя файла
Msg1            DB    'Ошибка распределения памяти', 0Dh, 0Ah, '$'
Msg2            DB    'Введите имя com или exe-файла', 0Dh, 0Ah, '$'
Msg3            DB    'Ошибка запуска программы',  0Dh, 0Ah, '$'
Blocpar         DB    14      DUP (0)       ; Блок параметров
;       Освобождение  лишней памяти
Begin:          MOV   SP,  OFFSET  Lprog    ; SP ← новая база стека
                MOV   AH, 4Ah               ; Функция изменения области памяти
                MOV   BX, (Lprog + 0Fh)/16  ; Новый размер области памяти
                INT   21h                   ; Изменение размера области
                JNC   M1                    ; Переход при отсутствии ошибки
                MOV   AH, 09h               ; Функция вывода строки
                MOV   DX, OFFSET Msg1       ; DX ← адрес сообщения о ошибке
                INT   21h                   ; Вывод строки
                JMP   Exit                  ; Переход на завершение программы
;          Ввод имени файла
```

```
M1:         MOV  AH, 09h                ; Функция вывода строки
            MOV  DX, OFFSET Msg2        ; DX ← адрес строки-приглашения
            INT  21h                    ; Вывод строки
            MOV  AH, 0Ah                ; Функция ввода строки
            MOV  DX, OFFSET Buff        ; DX ← адрес-смещение буфера ввода
            INT  21h                    ; Ввод строки
            MOV  AH, 2                  ; Функция вывода символа
            MOV  DL, 0Ah                ; DL ← символ перевода строки
            INT  21h                    ; Вывод символа
            CMP  Lname, 0               ; Имя файла отсутствует ?
            JZ   Exit                   ; Если да
;       Запуск новой программы
            XOR  BX, BX                 ; BX ← 0
            MOV  BL, Lname              ; BL ← длина имени файла
            MOV  Namefile[BX], 0        ; Запись нуля после имени файла
            MOV  DX, OFFSET Namefile    ; DX ← смещение имени файла
            MOV  BX, OFFSET Blocpar     ; BX ← смещение блока параметров
            MOV  AX, 4B00h              ; Функция и подфункция запуска
            INT  21h                    ; Запуск дочерней программы
            JNC  M2                     ; Переход при отсутствии ошибки
            MOV  AH, 9                  ; Функция вывода строки
            MOV  DX, OFFSET Msg3        ; Адрес строки-сообщения о ошибке
            INT  21h                    ; Вывод строки
M2:         JMP  M1                     ; Повторение для нового файла
;       Завершение программы
Exit:       MOV  AX, 4C00h              ;  Возврат в DOS с
            INT   21h                   ;    кодом завершения 0
;       Определение области стека
Stek        DW   64      DUP (?)        ; Новая область стека
Lprog       EQU  $ - Start + 100h       ; Длина программы (включая PSP)
_Text   ENDS
END         Start
```

Приведенная выше программа выполняет обмен с экраном и клавиатурой, используя следующие системные вызовы MS-DOS:

INT 21h (функция **9**) - вывод строки. Данный вызов уже использовался в п.3.2;

INT 21h (функция **2**) - вывод символа. Перед применением вызова программа помещает в регистр DL код выводимого символа. В программе данный системный вызов используется для перевода строки, что исключает наложение на экране строк, принадлежащих родительской и дочерней программам;

INT 21h (функция **0Ah**) - ввод с клавиатуры символьной строки. В качестве последнего символа строки вводится символ 0Dh (возврат каретки). (Этот символ помещается в водимую строку драйвером клавиатуры при нажатии клавиши <Enter>.) Перед применением вызова программа помещает в регистры DS и DX соответственно адрес-сегмент и адрес-

смещение для младшего байта буфера, в который должна быть введена строка. Данный буфер имеет три поля:

1) младший байт содержит максимальное число вводимых символов. Запись этого числа выполняет сама программа;

2) второй байт содержит число фактически введенных символов. Запись в этот байт выполняет MS-DOS;

3) остальные байты буфера содержат введенную строку, заканчивающуюся кодом 0Dh.

Первый исполнительный оператор JMP «перепрыгивает» через данные программы на ее фрагмент, выполняющий возврат в систему «лишней» памяти. Вспомним, что для com-программы MS-DOS выделяет всю ОП, оставшуюся после загрузки самой ОС. Так как наша программа использует лишь небольшую часть этой памяти, то она желает сообщить об этом системе. Простейший вариант этого – оставить за программой те 64К, в которые первоначально загружается com-программа. Но наша программа делает больше: она создает свой новый стек, разместив его сразу за своим кодом, а от всей остальной памяти отказывается.

Само освобождение памяти выполняет ранее рассмотренный системный вызов INT 21h (функция 4Ah). При этом количество запрашиваемых параграфов памяти определяется в результате целочисленного деления длины программы (константа Lprog), увеличенной на 15 (0Fh), на длину одного параграфа – 16.

После того как память освобождена, программа выводит на экран просьбу ввести имя загружаемого файла. Если искомый файл находится в текущем каталоге, то достаточно ввести имя файла и расширение имени (com или exe). Иначе – следует ввести имя-путь файла. Если файл находится на другом логическом диске, то имя-путь должно предваряться именем этого диска, например: C:\DISTANT\PR1.COM .

Введенное имя файла передается на вход системного вызова INT 21h (функция 4Bh, подфункция 0). Этому предшествует небольшая корректировка имени: в последний байт (с кодом 0Dh) записывается число 0.

Интересно отметить, что данная программа может запускать сама себя. В результате создается новый экземпляр программы, обладающий всеми ее свойствами. Вы можете проверить практически это и другие свойства данного простейшего интерпретатора команд.

3.5. Резидентные программы

Обычные прикладные программы присутствуют в ОП только во время своего выполнения. Когда это выполнение завершено (например, в результате системного вызова INT 21h, функция 4Ch), вся занимавшаяся программой память освобождается и может быть распределена другой программе. Но если данная программа может потребоваться в дальнейшем для многократного

обслуживания других прикладных программ и (или) для особой обработки каких-то событий, возникающих в ВС, то желательно сделать данную программу резидентной. ***Резидентная программа*** – программа, постоянно находящаяся в ОП. Например, резидентна основная часть ОС, называемая ***ядром***.

Для того, чтобы сделать свою программу резидентной, достаточно воспользоваться системным вызовом **INT 27h**. Этот вызов завершает выполнение программы, оставив ее в памяти. Перед применением вызова программа должна поместить в регистр DX адрес-смещение для байта, расположенного сразу же после резидентной программы.

После своего создания резидентная программа занимает место в памяти, но ничего не делает (не выполняется на ЦП). Это продолжается до тех пор, пока программа не будет инициирована (запущена). Единственно возможным источником такого инициирования является поступление в ЦП сигнала прерывания. Поэтому резидентная программа обязательно является обработчиком прерываний.

Вспомним (см. п.2.6.4), что обязательным условием запускаемости обработчика прерываний является размещение его стартового адреса в векторе прерываний, номер которого совпадает с номером прерывания. Для такого размещения следует использовать системный вызов **INT 21h** (функция **25h**). Перед применением вызова программа должна поместить в регистр AL номер прерывания, а в регистры DS и DX соответственно адрес-сегмент и адрес-смещение для обработчика прерываний.

Выбор номера прерывания зависит от того, какую функцию будет выполнять обработчик прерываний. Если выполняемая им функция уже реализована в системе, то новый обработчик прерываний должен ориентироваться на уже используемый номер прерывания. Иначе для новой функции необходимо использовать и новый номер прерывания. Вначале рассмотрим второй случай.

Для выбора свободного номера прерывания можно использовать отладчик Debug. С помощью его команды **D 0:0** просмотрим начало области памяти, занимаемой таблицей векторов прерываний. (Для продолжения просмотра достаточно вводить команду **D**.) В этой области выберем любую последовательность из четырех нулевых байтов, номер младшего из которых делится нацело на четыре. Тогда частное от такого деления и есть искомый номер прерывания. Например, пусть выбранная последовательность четырех байтов начинается с номера 200h = 512. Тогда искомый номер прерывания n = 200h/4 = 80h = 128.

Пример. Сделаем резидентной простую com-программу, рассмотренную в п.3.2.2. Для ее инициирования будем использовать программное прерывание с номером 80h. Текст программы на ассемблере:

```
;          Резидентная программа выводит сообщение “Здравствуйте” на экран
;          ---------------------------------------------------------------------------------
;       Вызов программы:  команда   INT 80h
;
cr       EQU   0Dh                          ; Код ASCII возврата каретки
lf       EQU   0Ah                          ; Код ASCII перевода строки
         ASSUME   CS:_Text
_Text  SEGMENT   PUBLIC  ‘CODE’
         ORG   2Ch
Blocokr  DW  ?                              ; Адрес блока окружения
         ORG   100h
Start:  JMP    Init                         ; Переход на инициализацию
;      Обработчик прерываний
Msg     DB      ‘Здравствуйте’, cr, lf, ‘$’    ; Выводимое сообщение
Begin: STI                                  ; Разрешить маскируемые прерывания
         PUSH  AX                           ; Сохранение содержимого
         PUSH  DX                           ;     регистров
         PUSH  DS                           ;          в стеке
         MOV   AX, CS                       ; DS будет адресовать данные
         MOV   DS, AX                       ;     резидентной программы
         MOV   DX, OFFSET Msg               ; DX ← адрес строки-сообщения
         MOV   AH, 9                        ; Вывод строки
         INT     21h                        ;   на экран
         POP     DS                         ; Восстановление содержимого
         POP     DX                         ;     регистров
         POP     AX                         ;          из стека
         IRET                               ; Возврат из прерывания
;    Инициализационная часть программы
Init:    MOV   AX, 2580h                    ; Запись стартового адреса обработчика
         MOV   DX, OFFSET Begin             ;     в вектор прерываний
         INT     21h                        ;          с номером 80h
         MOV   AH, 49h                      ; Освобождение области памяти,
         MOV   ES, Blocokr                  ;     занимаемой
         INT     21h                        ;          блоком окружения
         MOV   DX, OFFSET Init              ; Возврат в DOS, оставшись
         INT     27h                        ;   резидентным
_Text     ENDS
END    Start
```

Данная программа состоит из двух основных частей: 1) обработчик прерываний (резидентная часть); 2) инициализационная часть. Инициализационная часть нерезидентна по той причине, что она выполняется всего один раз и нет смысла постоянно держать ее в памяти. При этом следует учитывать, что так как резидентная программа находится в ОП во время выполнения других программ, то занимаемую ею память следует минимизировать.

Другим средством уменьшения затрат памяти, которое применено в программе, является освобождение области, выделенной для блока окру-

жения программы. (Вспомним, что такая область статически назначается любой программе, и что ее размер кратен 512 байтам.) Подобное освобождение возможно потому, что наш обработчик прерываний (как и любой другой) эту область не использует.

Обратим внимание, что первые команды обработчика прерываний запоминают в стеке, а последние команды извлекают из стека содержимое тех регистров, в которые выполняется запись в программе обработчика. Среди этих регистров есть и регистр сегмента данных DS, в который обработчик записывает новое значение, совпадающее с содержимым регистра CS. Это обусловлено тем, что при вызове обработчика (через прерывание) новое значение записывается лишь в единственный сегментный регистр CS (это значение берется из вектора прерываний). Остальные регистры сегментов (DS, SS и ES) сохраняют свое значение, полученное ранее в вызывающей программе. Поэтому для того, чтобы программа обработчика могла использовать свои данные, она помещает в DS соответствующий адрес-сегмент (для com-программ он совпадает с содержимым CS).

Интересно добавить, что так как содержимое регистров SS и SP поступает в обработчик прерываний из вызывающей программы без изменений, то обработчик использует тот же самый стек, что и программа. Это свойство позволяет достаточно просто выполнять информационный обмен между программой и обработчиком, используя стек.

Перейдем к рассмотрению более сложного случая, когда обработчик прерываний должен использовать уже занятый вектор прерываний. В этом случае требуется выполнить ***перехват прерываний*** – запись в вектор прерываний стартового адреса нового обработчика. Это нетрудно сделать, используя рассмотренный ранее системный вызов INT 21h, функция 25h. Но дополнительная задача заключается в том, что обычно требуется сохранить прежнее содержимое вектора прерываний для того, чтобы прежний обработчик прерываний можно было бы впоследствии инициировать.

Для чтения прежнего содержимого вектора прерываний используется системный вызов **INT 21h**, функция **35h**. Перед выполнением вызова программа должна поместить в регистр AL номер прерывания. В результате выполнения вызова регистры ES и BX содержат соответственно адрес-сегмент и адрес-смещение, находящиеся в векторе прерываний. Далее эти адреса могут быть переписаны для сохранения в любое место ОП.

Инициирование прежнего обработчика прерываний может потребоваться в двух случаях. Во-первых, в том случае, если новый обработчик прерываний не заменяет, а лишь расширяет функции, выполнявшиеся прежним обработчиком. При этом после того, как новый обработчик выполнится, он должен передать управление старому обработчику. Во-вторых, часто обработчик прерываний должен находиться в ОП не до конца работы системы. В этом случае перед тем, как освободить память, необходимо восстановить старый обработчик прерываний. Это нетрудно сде-

лать с помощью системного вызова INT 21h, функция 25h. Рассмотрим реализацию первого случая.

Пример. Следующая резидентная программа “дополняет” обработку прерывания с номером 0. Системный обработчик данного прерывания-исключения прекращает выполнение текущей программы, если ее команда выполнила деление на 0. Наш обработчик спрашивает пользователя о желании завершить программу. В случае нажатия клавиши y (или Y) выполнение программы прекращается. При нажатии любого другого символа выполнение программы продолжается.

```
;          Резидентная программа выполняет обработку исключения 0 (деление на 0)
;          ---------------------------------------------------------------------------------------
;
cr      EQU   0Dh                                ;  Код ASCII возврата каретки
lf      EQU   0Ah                                ;  Код ASCII перевода строки
        ASSUME   CS:_Text
_Text   SEGMENT   PUBLIC  ‘CODE’
        ORG   2Ch
Blocokr   DW  ?                                  ; Адрес блока окружения
        ORG   100h
Start:  JMP   Init                               ;  Переход на инициализацию
;       Обработчик прерываний
Msg     DB    ‘Завершить программу ?’, cr, lf, ‘$’     ; Выводимое сообщение
Old     DD    ?                                  ; Прежнее содержимое вектора пр-й
Begin:  STI                                      ; Разрешить маскируемые прерывания
        PUSH  AX                                 ; Сохранение содержимого
        PUSH  DX                                 ;     регистров
        PUSH  DS                                 ;         в стеке
        PUSHF                                    ; Сохранение регистра флагов
        MOV   AX, CS                             ; DS будет адресовать данные
        MOV   DS, AX                             ;     резидентной программы
        MOV   DX, OFFSET Msg                     ;  DX ← адрес строки-сообщения
        MOV   AH, 9                              ;  Вывод строки
        INT   21h                                ;    на экран
        MOV   AH,1                               ; Ввод символа с
        INT   21h                                ;    клавиатуры
        OR    AL, 20h                            ; Перевод символа на нижний регистр
        CMP   AL, ‘y’                            ; Введен символ “y” ?
        JNZ   Exit                               ; Если нет – то выход из прерывания
        POPF                                     ; Восстановление регистра флагов
        POP   DS                                 ; Восстановление содержимого
        POP   DX                                 ;     регистров
        POP   AX                                 ;         из стека
        JMP   Old                                ; Вызов прежнего обработчика
Exit:   MOV   AH, 2                              ; Перевод
        MOV   DL, lf                             ;    строки
        INT   21h                                ;       экрана
        POPF                                     ; Восстановление регистра флагов
        POP   DS                                 ; Восстановление содержимого
```

```
        POP   DX                    ;     регистров
        POP   AX                    ;         из стека
        IRET                        ; Возврат из прерывания
;   Инициализационная часть программы
Init:   MOV   AX, 3500h             ; Сохранение прежнего содержимого
        INT   21h                   ;   вектора 0 в регистрах ES и BX
        MOV   WORD  PTR Old, BX     ; А затем
        MOV   WORD  PTR Old+2, ES   ;    в поле Old
        MOV   AX, 2500h             ; Запись стартового адреса обработчика
        MOV   DX, OFFSET Begin      ;     в вектор прерываний
        INT   21h                   ;          с номером 0
        MOV   AH, 49h               ; Освобождение области памяти,
        MOV   ES, Blocokr           ;     занимаемой
        INT   21h                   ;         блоком окружения
        MOV   DX, OFFSET Init       ;  Возврат в DOS, оставшись
        INT   27h                   ;   резидентным
_Text   ENDS
END   Start
```

Как и предыдущая программа, данная программа состоит из инициализационной и резидентной частей. Инициализационная часть начинает свою работу с сохранения прежнего содержимого вектора прерываний 0 в двойном слове ОП с меткой Old. Затем она помещает в вектор 0 стартовый адрес нашего обработчика прерываний и сокращает объем занимаемой им памяти. В завершение инициализационная часть выполняет возврат в MS-DOS, оставив в памяти обработчик прерываний.

Обработчик прерываний (резидентная часть) запускается в результате прерывания-исключения с номером 0. В начале своего выполнения он разрешает маскируемые внешние прерывания, а также сохраняет в стеке используемые им регистры. Обратите внимание, что среди сохраняемых регистров находится регистр флагов. Это делается для того, чтобы прежний обработчик прерываний получил такое содержимое этого регистра, которое установила ранее программа. (На тот случай, если некоторые флаги используются для передачи информации на вход обработчика.)

Далее резидентная часть выводит на экран вопрос о желании прекратить выполнение программы, выполнившей деление на 0. Ответ пользователя (один символ) переводится в строчный символ. Такой перевод выполняется установкой единственного бита 5, которым отличаются коды строчной и прописной одноименных букв. Если пользователь не желает прекращать выполнение программы, то выполняется возврат из прерывания.

Иначе управление передается системному обработчику. Это делается с помощью команды JMP, выполняющей дальний безусловный переход на стартовый адрес этого обработчика. Закономерен вопрос: как системный обработчик возвратит управление (если пожелает) в прерванную программу? Ответ: обычным способом, то есть с помощью команды IRET. Это

объясняется тем, что системный обработчик «наследует» от нашего обработчика стек прерванной программы, в который был помещен ранее адрес возврата (в момент прерывания).

Для проверки работоспособности рассмотренного обработчика прерываний необходимо: после того как приведенная выше программа будет выполнена (сделав обработчик прерываний резидентным), запустить на выполнение простейшую прикладную программу, выполняющую деление на нуль в бесконечном цикле.

Если резидентная программа полностью заменяет ОС в обработке каких-то типов прерываний, то по истечении какого-то времени у пользователя (или у прикладной программы) может возникнуть желание вернуться к системной обработке этих прерываний. Для этого следует выполнить следующие три действия:

1) закрыть файлы, открытые резидентной программой;
2) восстановить векторы прерываний так, чтобы они указывали на системные обработчики;
3) освободить память, занимаемую резидентной программой.

Перечисленные действия должна выполнить сама резидентная программа при получении ею команды о завершении. Например, если резидентная программа выполняет обработку прерываний от клавиатуры, то код одной из клавиш (комбинации клавиш) может использоваться в качестве такой команды.

Для восстановления векторов прерываний можно использовать рассмотренные ранее системные вызовы INT 21h (функции 35h и 25h). Первый из этих вызовов позволяет запомнить, а второй – восстановить прежнее содержимое вектора прерывания.

Для освобождения памяти, занимаемой резидентной программой, можно воспользоваться системным вызовом INT 21h (функция 49h), который требует задания в регистре ES адреса-сегмента области резидентной программы. Так как адрес-сегмент резидентной программы находится во время ее выполнения в регистре CS, следующий фрагмент программы показывает, как можно осуществить такой вызов:

```
PUSH  CS          ; Копирование CS
POP   ES          ;   в ES
MOV   AH, 49h     ; Освобождение
INT   21h         ;    памяти
. . . . . .       ; Восстановление регистров из стека
IRET              ; Возврат из прерывания
```

4.ФАЙЛОВАЯ ОРГАНИЗАЦИЯ ИНФОРМАЦИИ

4.1. Файлы

На любом носителе (ВП или ОП) информация хранится в виде длинной битовой строки, т.е. в виде последовательности нулей и единиц. Но в отличие от ОП, в которой битовая строка разбита на байты, информация на носителе ВП, например на диске, разбивается на фиксированные части большей длины, называемые ***секторами***. (Обычно длина сектора составляет 512 байтов.) Разбиение пространства носителя на секторы выполняется во время ***физического форматирования*** носителя. Это форматирование не зависит от используемой ОС и зависит только от свойств носителя.

Подобно тому, как пронумерованы (имеют реальный адрес) все ячейки (байты) ОП, пронумерованы также все секторы носителя ВП. Пользуясь номером требуемого сектора, любая прикладная или системная программа может выполнить чтение или запись этого сектора, обратившись к драйверу соответствующего устройства ВП. Драйверы позволяют программам выполнять информационный обмен не только с устройствами ВП, но и с устройствами ввода-вывода. Вопросы построения драйверов будут рассмотрены в п.5, а пока перечислим недостатки выполняемого ими информационного обслуживания программ:

1) если объем информации на носителе достаточно велик, то вся работа по поиску нужных секторов в этом объеме ложится на соответствующую системную или прикладную программу;
2) секторы часто неудобны для обработки их в программе;
3) при смене типа ПУ, используемого для ввода или вывода информации данного вида, приходится вносить существенные изменения в программу из-за влияния интерфейсов соответствующих драйверов. Примером является вывод выходных данных программы в одном случае на принтер, а в другом – на экран.

Первые два недостатка относятся только к устройствам ВП, а последнее – ко всем ПУ. Для устранения перечисленных трудностей информационное обслуживание программ в большинстве ОС выполняется с помощью файлов. Далее будем различать физические и логические файлы.

Физический файл – информация, расположенная в непрерывной или разрывной области носителя ВП, имеющая имя, уникальное для данной ВС. Информация (битовая строка) на носителе делится на части-файлы по смысловому принципу. Например, один файл может содержать текст исходной программы, второй – ее объектный модуль, а третий – загрузочный модуль. Кроме того, в большинстве современных ОС каждое устройство ввода-вывода также считается физическим файлом. В результате все информационное обслуживание программ выполняют подпрограммы ОС для

работы с файлами (рис.24). Что касается информации в ОП, то доступ к ней любая программа имеет и без помощи этих подпрограмм.

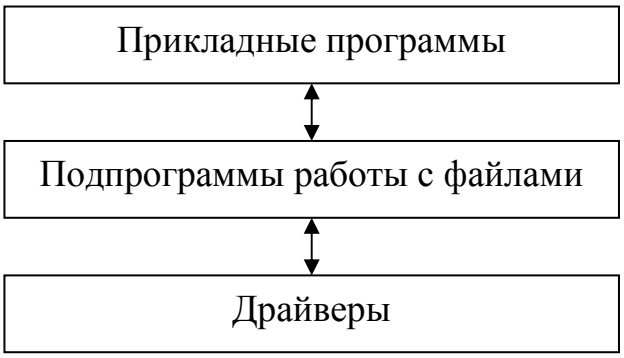


Рис.24. Информационное обслуживание программ

Битовая строка, образующая физический файл, делится на части, называемые ***физическими записями***. Для файлов на носителях ВП такими записями являются секторы. Для файлов-устройств ввода/вывода физическими записями обычно являются байты.

Емкость некоторых носителей ВП, например, жестких дисков, очень велика и составляет десятки Гбайт (1Гбайт = $1К^3$) . Поэтому для упрощения работы с ними операционная система, во-первых, предоставляет своим пользователям возможность представлять реальный носитель ВП в виде совокупности нескольких виртуальных носителей, называемых ***разделами*** или ***логическими дисками***. На каждом из этих виртуальных носителей может существовать свое множество файлов, причем эти множества не пересекаются.

Во-вторых, для того чтобы уменьшить число адресуемых участков носителя, ОС объединяет соседние секторы диска в более крупные информационные единицы, называемые ***кластерами***. Размер кластера зависит от типа носителя и равняется размеру сектора диска, умноженному на степень двойки: 512байт x N, где N = 1, 2, 4, ...

Что касается имени физического файла, то их два: простое и полное. ***Простое имя файла –*** имя, которое дает пользователь физическому файлу при его создании. В достаточно старых операционных системах используется короткое простое имя файла. Например, в MS-DOS (за исключением версии 7.0) длина простого имени не может превышать 12 символов по схеме ‘8.3’. При этом до восьми символов имеет ***собственно имя файла***, до трех символов – ***расширение имени файла***, один символ – разделительная точка. В MS-DOS версии 7.0 и во многих других современных ОС длина простого имени файла может достигать 255 символов.

В отличие от простого имени файла, которое не отвечает требованию уникальности в пределах всей ВС, таким свойством обладает ***полное имя***

файла, называемое также ***именем-путем***. Подобные имена будут рассмотрены подробно в п.4.2.1.

Любая ОС предоставляет прикладным и системным обрабатывающим программам возможность работать не с физическими, а с логическими (виртуальными) файлами. ***Логический файл*** содержит ту же самую битовую строку, что и физический файл, но имеет отличия:

1) имя файла другое (более простое). В качестве такого имени обычно используется ***номер логического файла*** среди логических файлов данной программы. Этот номер может принимать значения: 0; 1; …L_{MAX};

3) возможно другое разбиение файла на записи;

4) логический файл не связан с конкретным носителем информации.

Перечисленные отличия существенно упрощают работу программы с файлом. Первое из них упрощает идентификацию файла при выполнении операций над ним, а последнее отличие делает излишними переделки программы при смене обрабатываемых ею физических файлов. Перейдем к рассмотрению второго отличия.

Прежде всего заметим, что для программы очень неудобна работа с физическими записями (секторами или кластерами) из-за отсутствия смысловой связи между такой записью и содержащейся в ней информацией. Для любой обрабатывающей программы требуется, чтобы файл был разбит на ***логические записи***, выделяемые по смыслу.

Прежние ОС поддерживали разбиение логических файлов на логические записи. В результате программа могла, например, выполнить чтение или запись логической записи, используя для ее идентификации или номер записи в файле, или ее ключ (***ключ*** – символьное имя записи). Современные ОС не поддерживают разбиение логических файлов на записи. Они позволяют программе работать с логическим файлом так, как с длинной последовательностью байтов. Естественно, что разбиение логического файла на записи есть. Но это разбиение известно только самой обрабатывающей программе. Для ОС оно не известно, и ее информационное общение с программой сводится к обмену между ними цепочками байтов, которые ОС или помещает в файл, или считывает из него.

Информационное обслуживание программ и пользователей на уровне логических записей, а также на уровне полей этих записей выполняют **СУБД** – ***системы управления базами данных***. Эти системные обрабатывающие программы пользуются услугами подпрограмм ОС по работе с файлами и поэтому могут рассматриваться в качестве надстройки над этими подпрограммами (рис.25).

4.2. Файловые системы

4.2.1. Структура файловой системы

Физический файл существует в ВС не обособленно, а как часть более крупной информационной единицы, называемой ***файловой системой*** (**ФС**). Эту систему образуют файлы и таблицы, расположенные на конкретном носителе ВП или расположенные на его части (в разделе или на логическом диске). Иногда в понятие файловой системы включают и совокупность подпрограмм ОС, выполняющих операции с файлами. Но мы для избежания путаницы этого делать не будем.

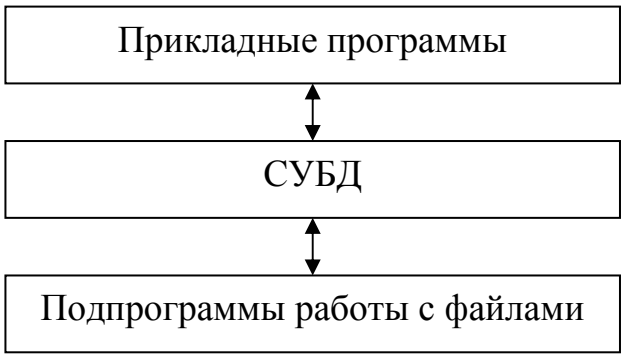


Рис.25. Информационное обслуживание программ с помощью СУБД

Например, файловые системы, поддерживаемые (обслуживаемые) MS-DOS, имеют тип **FAT**. Этот тип объединяет три родственных типа файловых систем: FAT12, FAT16 и FAT32 (последняя поддерживается только MS-DOS версии 7.0).

Несмотря на большое количество существующих типов ФС, между этими системами много общего. Во-первых, кроме самих физических файлов, любая ФС содержит информацию о размещении этих файлов на носителе. Во-вторых, для каждого файла ФС содержит его свойства, называемые ***атрибутами файла***. Основные различия между ФС сводятся к различиям характеристик:

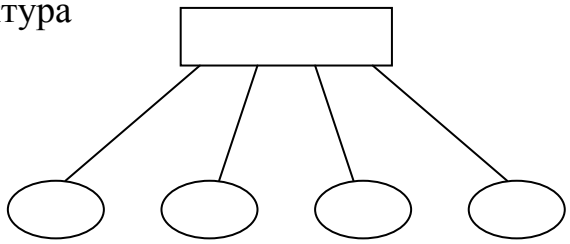
1) структура ФС;
2) состав атрибутов файла и место их хранения;
3) размещение элементов ФС на носителе;
4) место хранения информации о размещении файла;
5) длина простого имени файла.

Из перечисленных характеристик длина простого имени файла была рассмотрена ранее. Перейдем к рассмотрению других характеристик.

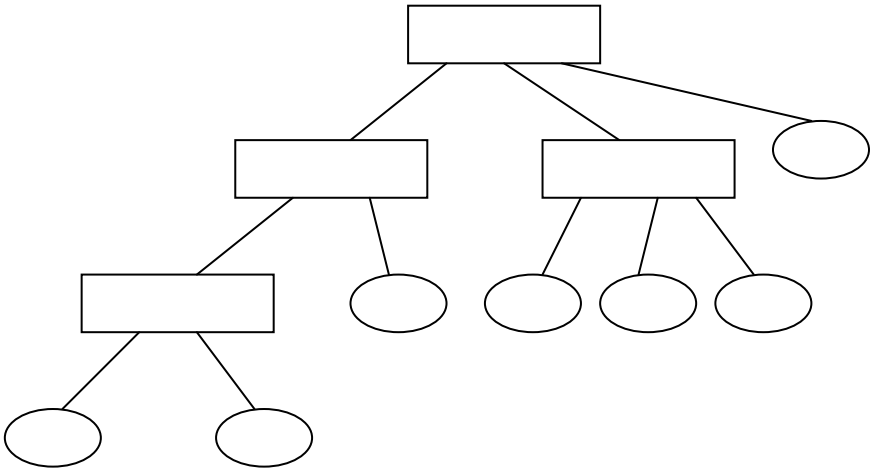
На рис.26 показаны основные типы структур ФС: линейная, иерархическая, иерархическая с пересечениями**.** ***Линейная структура*** предполагает наличие в ФС одного служебного файла, называемого ***каталогом***.

Этот файл содержит сведения о размещении на носителе файлов данных. Так как имя физического файла должно обладать уникальностью (в пределах ФС), то при применении линейной структуры требуется, чтобы уникальностью обладало простое имя файла. На практике такое требование очень трудно выполнить, так как число файлов в ФС обычно достаточно велико. Поэтому ФС с линейной структурой в настоящее время почти не используются.

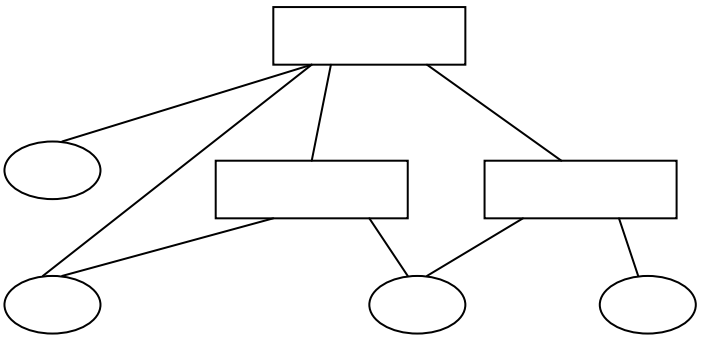
а) линейная структура

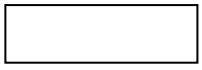


б) иерархическая (древовидная) структура



в) иерархическая структура с пересечениями



 - каталог

 - файл данных

Рис.26. Основные типы структур ФС

Иерархическая (древовидная) структура предполагает наличие в ФС многих каталогов. Корнем дерева является ***корневой каталог***. На следующем уровне дерева находятся те файлы и каталоги, данные о местоположении которых содержатся в корневом каталоге. Аналогично, каталоги первого уровня дерева “порождают” файлы и каталоги второго уровня и т.д. Большим достоинством древовидной структуры является то, что она позволяет пользователю не заботиться об уникальности простых имен файлов. Это объясняется тем, что ОС работает не с этими именами, а с путями. ***Имя-путь файла*** представляет собой последовательность всех имен, начиная с корневого каталога и кончая простым именем файла.

Следует отметить, что ОС, поддерживающая иерархическую ФС, в любой момент времени “помнит” не только текущий логический диск, но и ***текущий каталог*** на этом диске. (С помощью команды ОС пользователь может сменить текущий каталог.) Поэтому если искомый файл связан с текущим каталогом, то его можно задать для ОС не с помощью имени-пути, а пользуясь его простым именем. ОС сама получит имя-путь файла, соединив имя-путь каталога с именем файла. Кроме того, пользователь или программа может использовать ***промежуточное имя файла***. Такое имя представляет из себя часть полного имени и включает простое имя файла, а также “смещение” файла относительно текущего каталога.

Иерархическая структура с пересечениями отличается от обычной иерархической структуры тем, что один и тот же файл может быть зарегистрирован в нескольких каталогах. Подобную структуру имеют многие ФС, поддерживаемые многопользовательскими ОС.

ФС типа FAT имеют иерархическую структуру. На рис.27 приведен пример такой структуры. В ней корневой каталог имеет имя ‘\’ (обратный слеш), а имя каждого следующего каталога завершается этим символом.

Пример имени-пути:

\DISTANT\IVANOV\PROB1.ASM

Нетрудно заметить, что другой файл с таким же простым именем PROB1.ASM, но расположенный в другом каталоге, имеет другое имя-путь:

\DISTANT\PETROV\PROB1.ASM

Допустим, что текущим каталогом является DISTANT, тогда последний файл может быть указан с помощью промежуточного имени:

PETROV\PROB1.ASM

Что касается структуры каталога, то этот файл состоит из 32-байтовых логических записей. В FAT12 и в FAT16 одна такая запись соответствует одному файлу. В ней находятся: простое имя файла (собственно имя и расширение имени); номер первого кластера логического диска, занимаемого данным файлом; атрибуты файла. В FAT32 каждому файлу соответствуют несколько 32-байтовых записей каталога. Структура первой из них аналогична записи каталога в FAT12 и в FAT16, а в последующих записях содержится длинное (до 255 байтов) простое имя файла. Таким

образом, один и тот же файл может обрабатываться как системными вызовами, предназначенными для работы с короткими, так и системными вызовами для работы с длинными именами файлов.

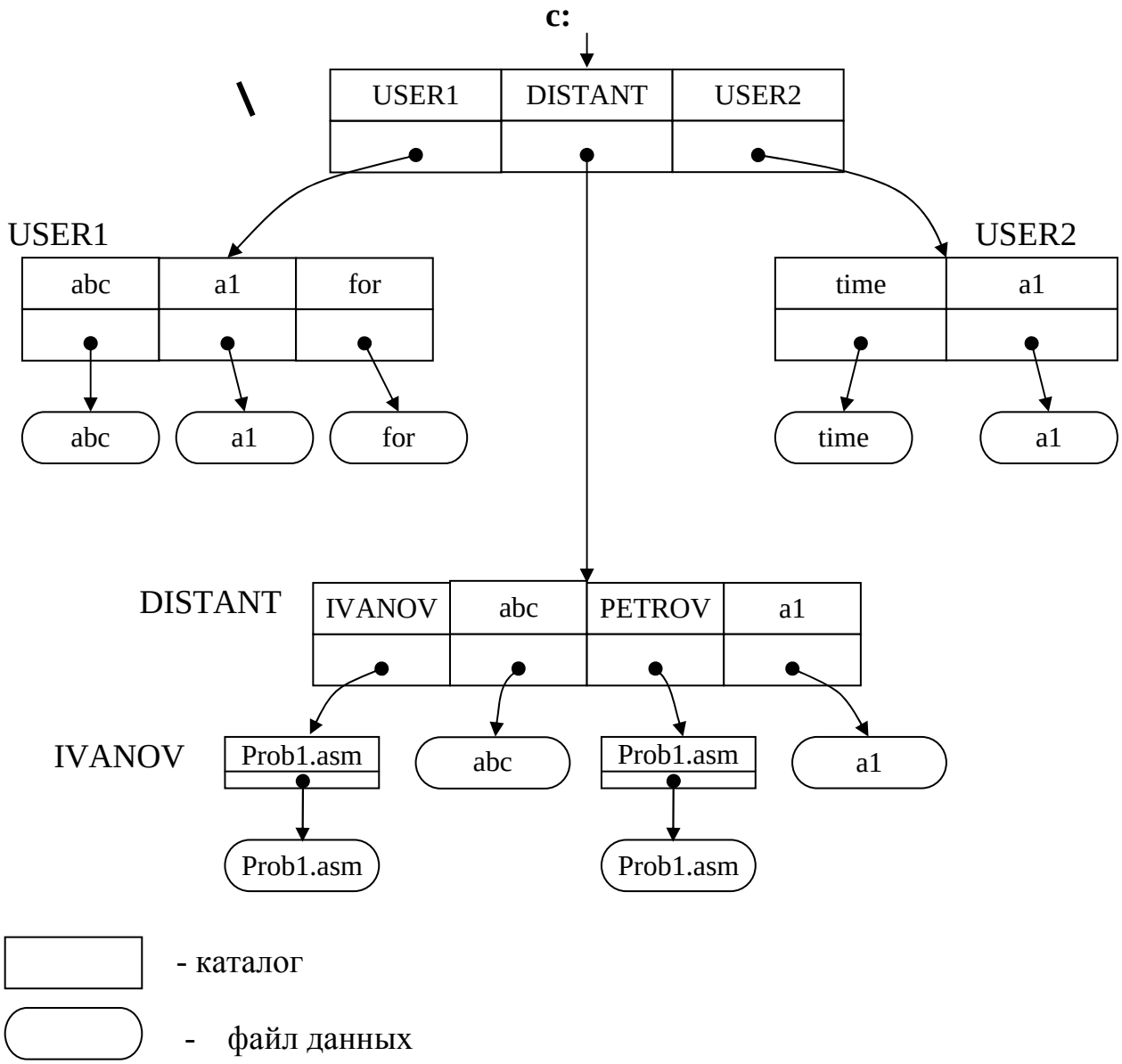


Рис.27. Пример иерархической структуры ФС типа FAT

4.2.2. Атрибуты файла

Любая ФС содержит для каждого из своих файлов его атрибуты. Они используются самой ОС, а также пользователями и обрабатывающими программами. Наименьшее число атрибутов имеют файлы в ФС, поддерживаемых однопользовательскими однопрограммными системами. Например, в FAT файл имеет атрибуты:

1) дата и время последней модификации файла;
2) текущий размер файла;

3) флаг “каталог”;
4) флаг “только для чтения”;
5) флаг “скрытый файл”;
6) флаг “системный файл”;
7) флаг “архивный файл”.

Атрибуты файла могут находиться или в записи каталога, соответствующей файлу, или в отдельной таблице (в части таблицы). Как уже говорилось, в FAT атрибуты файла содержатся в 32-байтовой записи каталога, соответствующей этому файлу.

4.2.3. Размещение элементов файловой системы

Различные ФС различным образом распределяют пространство носителя ВП (или его раздела) между своими элементами (файлами и таблицами).

Первоначальное распределение пространства носителя производится во время ***логического форматирования***. В отличие от физического форматирования, которое не зависит не только от используемой ФС, но и от типа ОС, логическое форматирование носителя (или его раздела) ориентировано на применение конкретной ФС. В результате такого форматирования на носителе создаются корневой каталог и таблицы ФС. Кроме того, возможно, на носитель записывается начальный загрузчик – программа, предназначенная для загрузки используемой ОС в ОП (в начале работы или после сбоя).

В зависимости от типа ФС некоторые из ее элементов дублируются для повышения надежности: в случае повреждения одной области носителя может использоваться копия этой информации, расположенная в другой области. (Дублирование не спасает от программной порчи информации. Так как изменение информации на носителе программой синхронно выполняется для всех копий этой информации.)

На рис.28 приведено распределение пространства носителя ВП для ФС типа FAT.

Рис.28. Размещение системы FAT на носителе

Пространство ВП, выделенное в конкретной ФС для размещения файлов (в том числе, для размещения каталогов), может распределяться между ними двумя основными способами: 1) непрерывными разделами; 2) в виде совокупности блоков памяти, необязательно соседних друг с другом. Непрерывное распределение имеет серьезные недостатки:

1) вся память назначается файлу ***статически*** – в момент его создания. Для многих файлов нельзя заранее предсказать их будущую длину и поэтому память для них должна выделяться ***динамически*** – по мере надобности во время выполнения операций записи в файл;

2) имеет место ***внешняя фрагментация памяти*** – наличие в ВП пустых участков, полученных в результате удаления ненужных файлов. Несмотря на значительную суммарную длину этих участков, отсутствуют непрерывные разделы, позволяющие размещать достаточно большие файлы.

Перечисленные недостатки привели к тому, что в настоящее время непрерывное распределение почти не используется для ФС на магнитных дисках. Оно применяется лишь для магнитных лент и лазерных дисков, так как пространство этих носителей распределяется между файлами только статически. (А для лазерных дисков отсутствует и внешняя фрагментация памяти.)

Что касается файлов на магнитных дисках, то память для них назначается динамически блоками, в качестве которых используются кластеры. Размер кластера для носителя (раздела) фиксируется при выполнении логического форматирования и составляет от 512 байт (1 сектор) до 64 К (128 секторов). Размер кластера зависит как от размера носителя (раздела), так и от типа ФС.

Например, в FAT12, FAT16 и FAT32 для кодирования номера кластера используются соответственно 12, 16 и 32 бита (отсюда различия в названиях ФС). Поэтому максимальный номер кластера в FAT12 равен 4095, в FAT16 – 65535, в FAT32 – 4 Г. Размер кластера для каждого носителя (раздела) выбирается с таким расчетом, чтобы произведение размера кластера на его максимальный номер не было меньше, чем размер носителя (раздела). Иначе на носителе будут неадресуемые кластеры.

Другим фактором, влияющим на выбор размера кластера, является наличие ***внутренней фрагментации*** – из всех кластеров ВП, выделенных файлу, в среднем, половина одного кластера (последнего) не используется. Поэтому, чем меньше размер кластера, тем меньше суммарный объем неиспользуемого пространства ВП.

Основным следствием из двух названных факторов является то, что FAT12 используется для небольших дисков (не более 16 Мбайт), что позволяет не делать кластеры более 4 Кбайт. FAT16 целесообразна для дисков не более 512 Мбайт. Для больших дисков целесообразна FAT32, которая позволяет для дисков емкостью до 8 Гбайт ограничиться размером кластера 4 Кбайт. Для больших дисков приходится применять размер кластера 8, 16 или 32 Кбайт.

4.2.4. Расположение информации о размещении файла

При размещении файла в непрерывном пространстве ВП информация о его местоположении сводится к двум численным параметрам: номер первого кластера файла и длина файла (в кластерах). Но при размещении файла в разрывном пространстве надо знать, где находится каждый его кластер. В различных ФС такая информация содержится:

1) в записи каталога;
2) в самих кластерах файла;
3) в специальной таблице.

Первый из этих способов предполагает, что запись каталога, соответствующая файлу, содержит перечень номеров всех его кластеров. Недостаток очевиден: размер записи каталога должен зависить от размера файла, что очень неудобно. Поэтому в записи каталога обычно хранится только номер самого первого кластера файла или указатель на место, где этот номер хранится.

Второй способ предполагает, что предыдущий кластер файла содержит номер следующего кластера. В результате все кластеры файла связаны в единый список. Считав из каталога номер первого кластера файла, по цепочке нетрудно прочитать номера всех остальных его кластеров. Основным недостатком данного способа является то, что для доступа к кластеру, расположенному в середине файла, необходимо прочитать все предыдущие кластеры.

Третий способ предполагает, что номера кластеров файла содержатся в специальной таблице. Данный способ отличается от предыдущего тем, что поиск нужного кластера ОС производит не путем считывания многих кластеров носителя, а путем просмотра содержимого одного и того же кластера (в котором находится таблица).

Реализация такого метода в MS-DOS основана на использовании таблицы размещения файлов FAT (File Allocation Table), название которой используется в качестве названия всей ФС. Таблица FAT имеет столько 12, 16- или 32-битных элементов, сколько кластеров носителя могут распределяться между файлами. Иными словами, FAT представляет собой уменьшенную модель распределяемой части носителя. Ее наличие позволяет размещать файл в разрывной области ВП. Для этого каждому файлу ставится в соответствие вспомогательный линейный связанный список, построенный из элементов таблицы FAT.

Вспомним, что одно из полей записи каталога, описывающей файл, содержит его начальный адрес. Этот адрес представляет собой номер первого кластера файла и, следовательно, номер соответствующего элемента в таблице FAT. Содержимым этого элемента является номер следующего элемента связанного списка, который совпадает с номером следующего кластера файла. Если элемент FAT-таблицы соответствует последнему

кластеру файла, то он содержит специальное число (FFFh, FFFFh или FFFFFFFFh).

4.2.5. Объединение файловых систем

В ВС обычно имеется несколько (разнотипных и однотипных) устройств ВП. На каждом из установленных носителей ВП имеется своя ФС. Более того, общепринято, что на каждом разделе (логическом диске) может располагаться своя собственная ФС. Причем ФС, располагаемые на одном носителе, могут быть как однотипными, так и разнотипными. Единственное требование: все ФС должны поддерживаться используемой ОС.

Возможны два способа отношения ОС к множеству ФС, находящихся под ее управлением в данный момент времени: 1) ФС существуют автономно и не связаны друг с другом; 2) деревья отдельных ФС объединяются в единое дерево с помощью операции, называемой ***монтированием***.

Первый подход применяется, в частности, в MS-DOS. При этом для того, чтобы ОС «знала», с какой ФС ведется работа в данный момент времени, используется понятие ***текущего логического диска***. С помощью команды MS-DOS пользователь или программа всегда может сменить этот диск. Если имя файла, используемое пользователем или программой при обращении к ОС, соответствует файлу, расположенному не на текущем логическом диске, то оно должно предваряться именем логического диска, например: C:\DISTANT\IVANOV\PROB1.ASM.

Второй из указанных подходов (монтирование ФС) широко используется в многопользовательских системах. При этом отметим, что несмотря на объединение деревьев ФС, работа с каждой из ФС производится в соответствии с ее типом, то есть разными подпрограммами ОС.

4.3. Операции над файлами

4.3.1. Создание и открытие файла

Работа с любым файлом начинается с его создания. Оно выполняется ОС по соответствующей просьбе со стороны программы. Во время создания файла добавляется новая запись в “родительский” каталог, заданный программой. В нее помещается простое имя файла, а также другая информация, соответствующая типу используемой ФС. Например, при включении файла в систему FAT, в запись каталога помещаются атрибуты нового файла. (Многие другие файловые системы размещают атрибуты файла не в каталоге, а в своих таблицах.)

Кроме того, при создании файла в ФС со статическим распределением ВП файлу выделяется необходимое пространство на носителе. В ФС с динамическим распределением памяти (к которым относится и FAT) па-

мять файлу при его создании не выделяется, а его длина принимается нулевой.

Создание файла является необходимым, но недостаточным условием выполнения информационного обмена с ним. Для этого требуется выполнить ***открытие файла***. Во время этой операции происходит связывание реального физического файла на носителе ВП (или файла-устройства ввода/вывода) с соответствующим логическим файлом.

Программа, по запросу которой ОС выполняет открытие файла, должна сообщить в системном запросе не только имя открываемого физического файла, но и тип последующей работы с ним (чтение или запись). В качестве результата открытия файла в программу возвращается логический номер файла. Рассмотрим теперь внутреннюю работу ОС при открытии файла.

Во-первых, во время открытия файла ОС создает для работы с ним столько буферов, сколько типов операций информационного обмена с ним будут выполняться. Поэтому при открытии файла и на чтение и на запись ему выделяются два буфера, а при открытии только на чтение – один. Размер системных буферов кратен одному сектору и обычно равен одному кластеру.

Во-вторых, ОС помещает **БУФ** (***блок управления файлом***) в свою ***таблицу открытых файлов***. БУФ содержит атрибуты файла (переписываются из каталога), а также указатели на системные буферы для работы с ним.

В-третьих, ОС просматривает ***таблицу логических номеров файлов*** для данной программы и выбирает из этой таблицы первый свободный элемент (содержит FFh). В этот элемент помещается номер соответствующего БУФ в таблице открытых файлов. После этого ОС возвращает управление в программу, передав ей в качестве логического номера файла номер элемента в таблице логических номеров. Таким образом, в результате открытия файла создается путь от логического номера файла в программе до физического файла на носителе (рис.29).

Для программы, выполняемой в среде MS-DOS, таблица логических номеров файлов содержится в PSP программы, начиная с адреса-смещения 18h. Размер этой таблицы равен 20 байтов (по одному байту на номер файла). Поэтому общее число файлов, открытых в программе, не может превышать 20.

Из этих номеров первые пять MS-DOS использует сама, открывая для программы (до начала ее выполнения) следующие файлы:

0 – стандартное устройство ввода (обычно клавиатура);

1 – стандартное устройство вывода (обычно экран);

2 – устройство вывода сообщений об ошибках (экран);

3 – последовательный порт (обычно COM1);

4 – параллельный порт (обычно LPT1).

Нетрудно заметить, что все перечисленные файлы являются файлами-устройствами. Приняв их во внимание, максимальное число файлов, открытых самой программой, не может превышать число 15. При этом следует учесть, что программа наследует от родительской программы все файлы, открытые ею к моменту создания новой программы. Другим ограничителем числа открытых файлов является строка FILES=N в командном файле конфигурации системы Config.sys. Число N задает предельное суммарное число файлов, открытых во всей системе. Именно столько строк имеет таблица открытых файлов. Последнее ограничение нетрудно скорректировать, используя любой текстовый редактор.

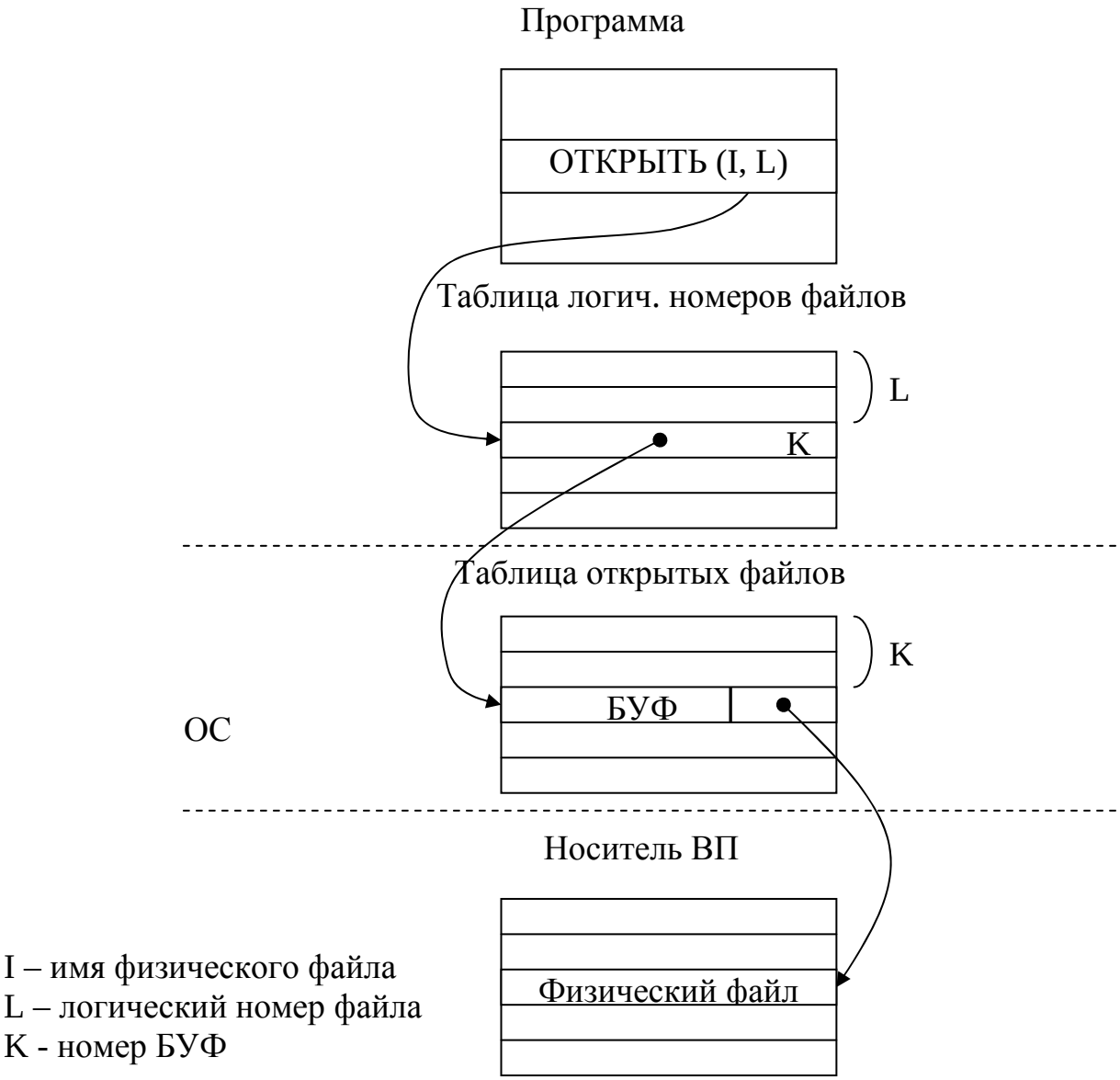


Рис.29. Результат операции открытия файла

В отличие от таблицы логических номеров, таблица открытых файлов является внутренней таблицей MS-DOS, к которой прикладные программы не имеют непосредственного доступа. Работу с этой таблицей выполняют только подпрограммы ОС, инициируемые в результате системных вызовов. Для создания и открытия файлов используются следующие системные вызовы MS-DOS: INT 21h (функция 3Ch) и INT 21h (функция 3Dh).

INT 21h (функция **3Ch**) – создание и открытие файла. Данный вызов создает и открывает новый файл. Если указанный файл уже существует, то он усекается до нулевой длины. Перед применением вызова программа помещает в какую-то область ОП символьную строку, представляющую собой имя создаваемого файла. Эта строка должна заканчиваться нулевым байтом. При этом имя файла может быть задано в любой из допустимых форм: имя-путь, предваряемое именем логического диска; имя-путь; промежуточное имя файла; простое имя.

В регистры DS и DX программа должна поместить соответственно адрес-сегмент и адрес-смещение области памяти с именем файла. В регистр CX должна быть помещена битовая строка, задающая особые свойства создаваемого файла. При этом установка каждого бита задает одно особое свойство (атрибут) файла. Например, бит1=1 требует, чтобы файл был скрытым (его имя не должно выводиться на экран при выполнении утилиты DIR). Если создается обыкновенный файл, то все биты-атрибуты должны быть нулевыми.

При успешном завершении системного вызова в программу возвращается флаг FC=0. При этом в регистре AX возвращается логический номер файла.

INT 21h (функция **3Dh**) – открытие существующего файла. Перед применением вызова программа задает в регистрах DS и DX адрес имени открываемого файла (аналогично функции 3Ch). В регистр AL она помещает ***режим доступа к файлу***. Младшие биты 0-1 режима доступа задают допустимые операции с открытым файлом:

00 – открыть для чтения;

01 – открыть для записи;

10 – открыть для чтения и для записи.

При успешном завершении системного вызова в программу возвращается флаг FC=0, а в регистре AX передается логический номер файла.

4.3.2. Операции чтения и записи

После того как программа открыла файл, она может выполнять информационный обмен с ним, используя операции чтения и записи. Выполняя эти операции с помощью соответствующих системных вызовов программа рассматривает файл как строку байтов. При этом все байты файла

пронумерованы начиная с нуля. Для того чтобы ОС «знала», к какому месту файла относится следующая команда чтения или записи, используется переменная, называемая ***указателем файла***. Она содержит номер байта файла, начиная с которого файл будет обрабатываться следующей командой.

Сразу после открытия файла его указатель содержит 0. Используя специальный системный вызов, программа может записать в эту переменную любое другое значение, что позволяет выполнять информационный обмен с внутренней частью файла напрямую, не просматривая его предыдущих байтов. В результате завершения операции чтения или записи указатель файла содержит номер байта, расположенного сразу же за строкой, обработанной командой. Поэтому, если программа выполняет последовательное чтение (запись) всего файла, отдельная установка его указателя не требуется.

Каждому открытому файлу всегда соответствует только один указатель, несмотря на то что файл может быть одновременно открыт в нескольких программах. (Вспомним, что однопрограммная система допускает одновременное нахождение в ОП нескольких программ, одни из которых находятся между собой в отношении «предок – потомок», а другие являются резидентными.) Наследование открытых файлов дочерней программой, а также наличие единых указателей файлов делают файлы удобным средством информационного обмена между родительской и дочерней программами в том случае, если объем передаваемой информации достаточно велик.

При выполнении операции чтения (записи) выполняется копирование строки байтов из файла (из прикладного буфера) в прикладной буфер (в файл). ***Прикладной буфер*** – область памяти в пространстве данных программы. При этом фактический информационный обмен выполняется, с одной стороны, между файлом на носителе и системным буфером, а с другой - между системным и прикладным буферами. Поэтому очередная операция чтения (записи) необязательно сопровождается фактическим информационным обменом с файлом. Подобный обмен отсутствует, если имеющееся содержимое системного буфера соответствует тому месту файла, которое указано в команде чтения (записи).

Следующие системные вызовы MS-DOS выполняют установку указателя файла, а также используются для информационного обмена с ним: INT 21h (функция 42h), INT 21h (функция 3Fh), INT 21h (функция 40h).

INT 21h (функция **42h**) – установить указатель файла. Перед применением вызова программа помещает в регистр AL тип требуемого перемещения указателя файла:

0 – перемещение относительно начала файла;

1 – перемещение относительно текущего положения указателя;

2 – перемещение относительно конца файла.

Величина перемещения (в байтах) представляет собой число со знаком, которое программа должна поместить в пару регистров: CX (старшая часть величины перемещения) и DX (младшая часть).

При успешном завершении системного вызова в программу возвращается флаг FC=0, а в регистрах CX и DX возвращается новое значение указателя файла (число байтов от начала файла).

Примечание. В результате выполнения данного вызова указатель может быть установлен не только внутри, но и за пределами файла. Если он принял отрицательное значение, то первая же операция чтения/записи закончится с ошибкой. Если же его величина положительна, но превосходит длину файла, то очередная операция записи увеличит размер файла.

Этот вызов часто используется для определения длины файла: достаточно выполнить вызов с CX=0, DX=0, AL=2, тогда в CX:DX будет возвращена длина файла в байтах.

INT 21h (функция **3Fh**) – выполнить чтение из файла. Чтение начинается с той позиции, на которую ранее был установлен указатель файла. Перед применением вызова программа помещает в регистр BX логический номер файла, в регистр CX – число байтов, читаемых из файла, а в регистры DS и DX – соответственно адрес-сегмент и адрес-смещение начала прикладного буфера, в который должно быть выполнено чтение.

При успешном завершении системного вызова в программу возвращается флаг FC=0, а в регистре AX возвращается число фактически считанных байтов. Указатель файла установлен на первый несчитанный байт.

Примечание. Если в результате выполнения данного вызова число фактически считанных байтов (в AX) меньше, чем число заказанных байтов (в CX), то, следовательно, был достигнут конец файла.

INT 21h (функция **40h**) – выполнить запись в файл. Запись начинается с той позиции, на которую ранее был установлен указатель файла. Перед применением вызова программа помещает в регистр BX логический номер файла, в регистр CX – число байтов, записываемых в файл, а в регистры DS и DX – соответственно адрес-сегмент и адрес-смещение прикладного буфера, содержимое которого должно быть записано в файл.

При успешном завершении системного вызова в программу возвращается флаг FC=0, а в регистре AX возвращается число фактически записанных байтов. Указатель файла установлен на байт, следующий за последним записанным байтом.

4.3.3. Закрытие и уничтожение файла

После того как программа завершила работу с файлом, она может его закрыть. ***Закрытие файла*** – операция, обратная открытию файла. В результате ее выполнения освобождаются ресурсы, выделенные для того, чтобы конкретная программа могла работать с конкретным файлом. Сюда

относятся системный буфер (буферы), а также логический номер файла. Если закрываемый файл был открыт ранее на запись, то при его закрытии, во-первых, выполняется копирование системного буфера записи в файл на диске. Во-вторых, корректируются атрибуты файла: дата и время последней модификации, размер файла.

В принципе, все файлы, открытые программой, закрываются при ее завершении с помощью системных вызовов, выполняющих возврат в ОС (а точнее – в родительскую программу). Но следующие два фактора способствуют тому, чтобы программа сама выполняла закрытие файла. Во-первых, так как суммарное число открытых файлов ограничено, то освобождение логического номера файла в результате его закрытия позволяет использовать этот номер для другого файла. Во-вторых, так как при закрытии файла производится сброс буфера записи на диск, то это гарантирует правильность данных в файле при аварийном завершении программы.

Если файл больше не нужен не только для текущего выполнения программы, но и для последующего использования, его следует удалить. ***Удаление файла*** – операция, обратная созданию файла. В результате нее удаляется запись в родительском каталоге, а также освобождается пространство на носителе ВП, выделенное ранее для файла. Заметим, что удаляемый файл обязательно должен быть закрыт. В противном случае возможны операции информационного обмена с несуществующим файлом.

Применительно к FAT удаление файла выполняется следующим образом. Во-первых, первый символ имени файла в родительском каталоге заменяется кодом E5h. Во-вторых, все элементы таблицы FAT, соответствующие кластерам файла, помечаются как свободные. Что касается самих кластеров файла, то их содержимое не меняется до тех пор, пока данный кластер не будет распределен другому файлу.

Следующие системные вызовы MS-DOS используются для закрытия и удаления файлов: INT 21h (функция 3Eh) и INT 21h (функция 41h)

INT 21h (функция **3Eh**) – закрытие файла. Перед применением вызова программа помещает в регистр BX логический номер закрываемого файла.

При успешном завершении системного вызова в программу возвращается флаг FC=0.

INT 21h (функция **41h**) – удалить файл. Перед применением вызова программа помещает в регистры DS и DX соответственно адрес-сегмент и адрес-смещение имени файла (оно задается аналогично функции 3Ch). При успешном завершении системного вызова в программу возвращается флаг FC=0.

4.3.4. Пример программы

Следующая программа выполняет копирование файла. Имя копируемого файла содержится в хвосте команды MS-DOS, запускающей данную программу. Имя файла-копии программа запрашивает у пользователя сама.

```
  ;          Программа копирует файл
  ;          --------------------------------
  ;      Имя копируемого файла – в хвосте программы, имя копии вводится с клавиатуры
  ;
cr        EQU   0Dh                                  ;  Код ASCII возврата каретки
lf        EQU   0Ah                                  ;  Код ASCII перевода строки
          ASSUME   CS:_Text
_Text   SEGMENT   PUBLIC  'CODE'
                  ORG   80h                  ; Следующий байт имеет адрес-смещение 80h
Lparam            DB    ?                          ; Длина хвоста команды
Param             DB    ?                          ; Первый байт хвоста
                  ORG   100h
Start:            JMP   Begin                      ; Переход через данные
;         Данные программы
Msg1              DB    'Нет имени файла', cr, lf, '$'          ; Выводимое сообщение
Msg2              DB    'Копируемый файл отсутствует', cr, lf, '$'    ;  --//--
Msg3              DB    'Введите имя файла-копии', cr, lf, '$'        ;  --//--
Msg4              DB    'Ошибка создания файла', cr, lf, '$'          ;  --//--
Msg5              DB    'Ошибка чтения файла', cr, lf, '$'            ;  --//--
Msg6              DB    'Ошибка записи в файл', cr, lf, '$'           ;  --//--
Msg7              DB    'Копирование выполнено', cr, lf, '$'          ;  --//--
Lname0            DB    81                         ; Максимальная длина имени файла
Lname             DB    0                          ; Фактическая длина имени файла
Namefile          DB    81       DUP (0)           ; Здесь размещается имя файла
Bufer             DB    512      DUP(?)            ; Буфер для копирования
Lognum1           DW    ?                          ; Логический номер файла 1
Lognum2           DW    ?                          ; Логический номер файла 2
;         Команды программы
Begin:
;       Поиск и открытие копируемого файла
                  MOV   CL, Lparam                 ;  CL ← Длина хвоста
                  CMP   CL, 0                      ; Хвост отсутствует ?
                  JNZ   M1                         ; Если хвост есть
                  MOV   DX, OFFSET Msg1            ; Вывод сообщения
                  MOV   AH, 9                      ;    об отсутствии имени
                  INT   21h                        ;          копируемого файла
                  JMP   Exit                       ; На завершение программы
M1:               XOR   BX, BX                     ; BX ← 0
                  MOV   BL, Lparam                 ; BL ← длина хвоста команды
                  MOV   Param[BX], 0               ; Хвост завершается нулем
                  MOV   AH, 3Dh                    ; Открытие
                  MOV   DX, OFFSET Param+1         ;     копируемого
```

```
                MOV  AL, 0                  ;       файла
                INT  21h                    ;           на чтение
                JNC  M2                     ; Если успешно
                MOV  DX, OFFSET Msg2        ;  Вывод сообщения
                MOV  AH, 9                  ;    об отсутствии
                INT  21h                    ;         копируемого файла
                JMP  Exit                   ; На завершение программы
M2:             MOV  Lognum1, AX            ; Сохранение логич. номера файла
;       Создание и открытие файла-копии
                MOV  DX, OFFSET Msg3        ;  Вывод сообщения с просьбой
                MOV  AH, 9                  ;     ввести имя
                INT  21h                    ;         файла-копии
                MOV  AH, 0Ah                ; Функция ввода строки
                MOV  DX, OFFSET Lname0      ; DX ← адрес-смещение буфера ввода
                INT  21h                    ; Ввод строки
                XOR  BX, BX                 ; BX ← 0
                MOV  BL, Lname              ; BX ← длина имени файла
                MOV  Namefile[BX], 0        ; Запись нулевого байта
                MOV  AH, 3Ch                ; Создание
                MOV  DX, OFFSET Namefile    ;     и
                MOV  CX, 0                  ;       открытие
                INT  21h                    ;           файла
                JNC  M3                     ; Если успешно
                MOV  DX, OFFSET Msg4        ;  Вывод сообщения
                MOV  AH, 9                  ;    об ошибке
                INT  21h                    ;         создания файла
                JMP  Exit                   ; На завершение программы
M3:             MOV  Lognum2, AX            ; Сохранение логич. номера файла
;         Копирование файла
M4:             MOV  AH, 3Fh                ; Функция чтения из файла
                MOV  BX, Lognum1            ; BX ← логич. номер читаемого файла
                MOV  CX, 512                ; CX ← число читаемых байтов
                MOV  DX, OFFSET Bufer       ;  DX ← адрес-смещение буфера
                INT  21h                    ; Чтение из файла в буфер
                JNC  M5                     ; Если успешно
                MOV  DX, OFFSET Msg5        ;  Вывод сообщения
                MOV  AH, 9                  ;    об ошибке
                INT  21h                    ;         чтения файла
                JMP  Exit                   ; На завершение программы
M5:             MOV  BX, Lognum2            ; BX ← логич. номер файла-копии
                MOV  CX, AX                 ; CX ← число записываемых байтов
                MOV  AH, 40h                ; Функция записи в файл
                MOV  DX, OFFSET Bufer       ;  DX ← адрес-смещение буфера
                INT  21h                    ; Запись из буфера в файл
                JNC  M6                     ; Если успешно
                MOV  DX, OFFSET Msg6        ;  Вывод сообщения
                MOV  AH, 9                  ;    об ошибке
                INT  21h                    ;         записи в файл
                JMP  Exit                   ; На завершение программы
```

```
M6:         CMP   AX, 512                ; Конец файла?
            JZ    M4                     ; Если нет
;     Завершение программы
            MOV   DX, OFFSET Msg7        ; Вывод сообщения
            MOV   AH, 9                  ;   о завершении
            INT   21h                    ;      копирования
Exit:       MOV   AX, 4C00h              ; Возврат в DOS с
            INT    21h                   ;  кодом завершения 0
_Text    ENDS
END         Start
```

Данная программа сначала открывает копируемый файл. Его имя должно поступить в программу в качестве «хвоста» (за исключением первого байта «хвоста», содержащего пробел). Логический номер файла, полученный в результате открытия, сохраняется для последующего чтения из файла. Далее создается новый файл, имя которого вводится с клавиатуры. Одновременно этот файл открывается для последующей записи в него.

Собственно копирование выполняется циклически. За одну итерацию цикла одна логическая запись (512 байтов) считывается из исходного файла в буфер программы Bufer, а затем переписывается из него в выходной файл. После этого число фактически записанных байтов сравнивается с числом 512. При неравенстве этих чисел делается вывод о достижении конца файла, и копирование прекращается.

Обратим внимание, что программа не выполняет закрытие файлов. Это объясняется, во-первых, небольшим количеством файлов, открываемых в программе. Во-вторых, вскоре после последней операции записи в файл программа завершается.

Рассмотренная программа пригодна для копирования любых обычных (не каталогов) файлов. Например, она может копировать себя. Допустим, что мы поместили загрузочный модуль программы в файл copir.com, тогда следующая команда выполняет подобное копирование:

```
copir.com  copir.com
```

Полученному файлу-копии можно дать любое имя (например, copir1.com) и содержащаяся в нем программа также может выполнять копирование (проверьте это).

4.3.5. Другие операции

Рассмотренные выше системные вызовы для работы с файлами являются основными, но не единственными. Кратко перечислим некоторые другие функции системного вызова INT 21h, используемые в MS-DOS для работы с обычными файлами (не каталогами), имеющими короткие двенадцатибайтовые имена:

43h – получить или установить атрибуты файла;

56h – переименовать файл;

57h – получить или установить дату и время создания файла.

Кроме того, MS-DOS предоставляет отдельные системные вызовы для работы с логическими дисками и с каталогами:

0Eh – установить текущий логический диск;

19h – получить текущий логический диск;

39h – создать каталог;

3Ah – удалить каталог;

3Bh – установить текущий каталог;

47h – получить текущий каталог.

MS-DOS версии 7.0 предоставляет также системные вызовы для работы с файлами в FAT32, используя их длинные имена. Так как эти же файлы имеют и короткие двенадцатибайтовые имена, то с ними могут работать и ранее рассмотренные системные вызовы.

5. УПРАВЛЕНИЕ ПЕРИФЕРИЙНЫМИ УСТРОЙСТВАМИ

5.1. Введение

Никакая прикладная или системная обрабатывающая программа не может выполняться без операций с ПУ. Кроме стандартных устройств ввода-вывода (например, экран и клавиатура) и внешней памяти (дисководы), существует огромное количество нестандартных периферийных устройств. Подобные устройства используются, например, для управления технологическими процессами. Сюда относятся различные датчики (устройства ввода), а также различные задвижки и вентили (устройства вывода).

Как отмечалось в п.1.3, для работы со стандартными ПУ программа может использовать системные подпрограммы ОС и BIOS, используя для этого соответствующие системные вызовы. При этом программе предоставляется возможность работать с ПУ не только как с устройствами, но и как с файлами. Подобные системные вызовы широко использовались нами в рассмотренных ранее примерах программ, заметно сокращая время программирования.

К сожалению, системных вызовов часто бывает недостаточно для управления ПУ по следующим причинам. Во-первых, время выполнения драйверов BIOS, и особенно MS-DOS, слишком велико. Во-вторых, для многих типов ПУ, например, для нестандартных, соответствующие системные драйверы отсутствуют. Следствием этого является то, что многие прикладные программы выполняют управление ПУ на аппаратном уровне, то есть фактически используют свои собственные драйверы. Далее мы рассмотрим вопросы построения таких драйверов.

Допустим, что ЭВМ имеет структуру с общей шиной (см. рис.6). С помощью одного ИУ к ОШ подсоединяются однотипные ПУ. Простейшее ИУ состоит из одного или более регистров, в состав которых обязательно входят регистр состояния и управления и буферный регистр данных. ***Буферный регистр данных*** (**RD**) предназначен для временного хранения одного или двух байтов данных, передаваемых на ПУ при выводе или принимаемых с ПУ при вводе.

Регистр состояния и управления (**RS**) играет для ИУ и подключенных к нему ПУ ту же роль, что регистр FLAGS играет для ЦП, то есть фактически является их блоком управления. Количество регистров состояния и управления, а также назначение их битов для разных ИУ существенно отличаются. На рис.30 приведен пример типичного RS, который будет использоваться нами в дальнейшем. В этом RS биты b_0–b_3 - ***управляющие биты***. Они устанавливаются программами, которые исполняются на ЦП, и предназначены:

1) если $b_0 = 1$, то разрешается работа ПУ по вводу (выводу) единицы информации;

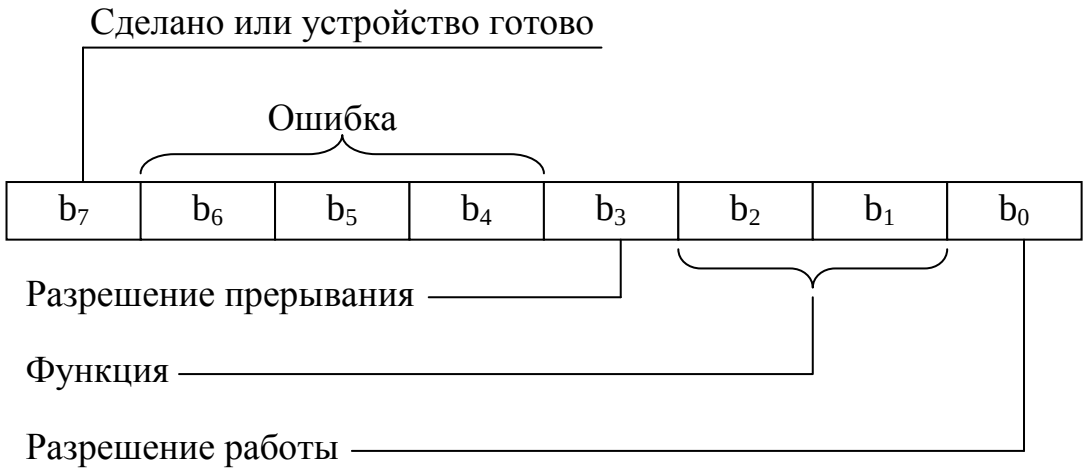


Рис.30. Пример регистра состояния и управления RS

2) b_1–b_2 используются для уточнения операции (функции), выполняемой устройством;

3) если $b_3 = 1$, то в ЦП может быть выдан сигнал прерывания.

На рис.30 биты b_4–b_7 – ***биты состояния*** **ПУ**. Они устанавливаются ПУ или его ИУ. При этом b_7 описывает состояние буферного регистра данных. Для устройства ввода $b_7 = 1$ означает, что RD заполнен данными, которые могут быть перенесены в ЦП. Для устройства вывода это означает, что выходной буфер данных сейчас пуст и ждет, когда ЦП загрузит в него новые данные.

Биты b_4–b_6 содержат информацию о том, произошла ли ошибка при выполнении последней операции ввода-вывода. Нулевое содержимое всех этих битов сообщает об отсутствии ошибки, иначе – эти биты содержат код ошибки.

При использовании простейшего ИУ всю работу по управлению ПУ выполняет сам ЦП. Поэтому он отвлекается от выполнения самих программ и, таким образом, значительно снижается его фактическая производительность. Применение более сложных ИУ позволяет возложить на них значительную часть работы по управлению ПУ, которая теперь в принципе может выполняться параллельно работе ЦП. Наиболее сложным ИУ является ***контроллер*** – специализированный процессор ввода-вывода. (Как и всякое ИУ, контроллер имеет свои RS и RD.)

Регистры ИУ, непосредственно доступные из программ ЦП, называются ***портами***. Портами являются многие RS и RD (но не все), а также служебные регистры, используемые для программного доступа к тем RS и RD, к которым нет непосредственного доступа. Каждый порт в ЭВМ имеет свой уникальный номер, называемый ***адресом порта***. Для некоторых типов ЦП программные операции с портами (чтение и запись) выполняются точно так же, как и операции с ячейками ОП. При этом для адресации пор-

тов используется часть адресного пространства ОП, в которой каждый реальный адрес соответствует не ячейке ОП, а порту.

В i8086 используется другой подход, при котором существуют два адресных пространства – одно для ячеек ОП, а второе – для портов. При этом адрес порта – число в диапозоне от 0 до 65535, а сам порт представляет собой 8–битный регистр. Следствием наличия двух адресных пространств является использование для работы с портами специальных команд ЦП: **IN** для ввода из порта и **OUT** – для вывода в порт.

Одной командой IN можно или ввести в регистр AL байт, или ввести в регистр AX 16-битное слово. Во втором случае ввод осуществляется из двух портов, имеющих смежные адреса, причем порт с меньшим адресом содержит младший вводимый байт. Адрес порта в команде IN можно задать двумя способами: 1) в виде константы; 2) в качестве содержимого регистра DX. Первый из этих способов позволяет использовать в команде IN лишь младшие адреса портов (от 0 до 255). Второй способ позволяет задавать любые порты (от 0 до 65535). Примеры:

```
IN    AL, 50h          ; AL ← (50h)
IN    AX, 50h          ; AX ← (51h : 50h)
IN    AL, DX           ; AL ← ((DX))
IN    AX, DX           ; AX ← ((DX)+1 : (DX))
```

Одна команда OUT позволяет вывести (скопировать) байт из регистра AL в заданный порт или вывести слово из регистра AX в два порта со смежными адресами. Адрес порта задается аналогично команде IN с той лишь разницей, что теперь порт является получателем и должен записываться в качестве первого операнда. Примеры:

```
OUT  50h, AL          ; (50h) ← (AL)
OUT  50h, AX          ; (51h : 50h) ← (AX)
OUT  DX, AL           ; ((DX)) ← (AL)
OUT  DX, AX           ; ((DX)+1 : (DX)) ← (AX)
```

Любой драйвер должен выполнять три основные функции:

1) подготовка ПУ и, возможно, самого драйвера к последующим операциям по обмену данными;

2) инициирование очередной операции ПУ по вводу (выводу) единицы информации;

3) обеспечение завершения очередной операции ввода-вывода.

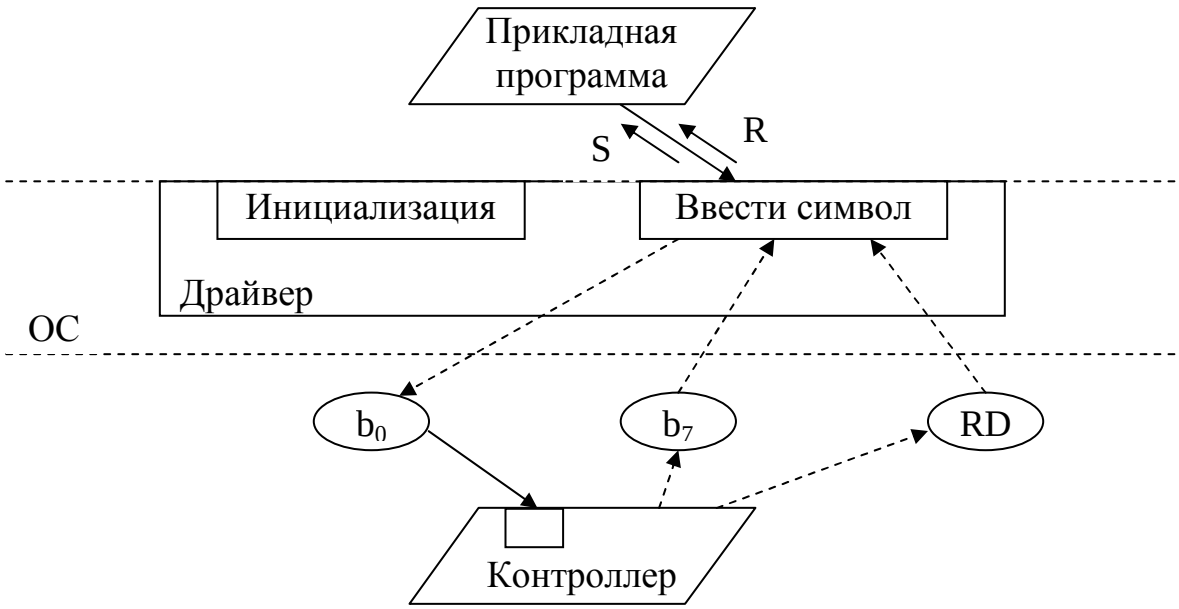
Реализация перечисленных функций в значительной степени зависит от принципа ввода-вывода, принятого при разработке драйвера. Различают следующие принципы: синхронный, асинхронный с прерываниями, прямой доступ в память, асинхронный с общей памятью.

5.2. Синхронный ввод-вывод

При синхронном вводе-выводе драйвер ожидает завершения текущей операции ввода-вывода активно, занимая в процессе ожидания ЦП. В результате этого нет никакого физического параллелизма между ЦП, с одной стороны, и контроллером и устройством – с другой. Реализацию синхронного принципа рассмотрим на следующем гипотетическом примере. (Реальный пример потребовал бы рассмотрения второстепенных технических деталей, присущих конкретному реальному ПУ.)

Пусть требуется разработать драйвер для управления работой считывателя перфоленты. Структура RS (адрес порта - 90h) приведена на рис.30. Код вводимого символа помещается в RD (адрес порта 91h). Результаты разработки драйвера рассматриваются в следующей последовательности: 1) логическая схема; 2) программа на ассемблере.

Логическая схема драйвера приведена на рис.31. Драйвер представляет собой совокупность двух интерфейсных (то есть доступных для вызова извне драйвера) логических процедур. Процедура “Инициализация” без параметров. Она выполняет два действия: 1) делает процедуру “Ввести символ” резидентной; 2) запрещает прерывания от считывателя перфоленты.



S – код символа;
R – признак результата (0 – успешно; 1 – ошибка);
b_0 – бит разрешения работы;
b_7- бит готовности.

Рис.31. Логическая схема синхронного драйвера

Вспомним, что инициирование резидентной подпрограммы возможно только через прерывание. Так как подпрограмму “Ввести символ” вызывают прикладные программы, то они должны использовать для этого команду программного прерывания INT. В качестве номера прерывания (этот номер совпадает с операндом команды INT и с номером вектора прерываний) можно взять любой неиспользуемый в системе номер, например, EEh.

Запрет аппаратных прерываний от управляемого ПУ выполняется потому, что при синхронном вводе-выводе они не нужны. Для выполнения такого запрета в нашем примере достаточно сбросить бит b_3 в регистре RS (см. рис.30). (На практике для этого можно выдать команду маскирования в программируемый контроллер прерываний (ПКП), который рассматривается в п.5.3.1.)

Логическая процедура «Ввести символ» обслуживает запросы прикладных и системных программ по вводу символов с перфоленты. После своего вызова эта логическая процедура устанавливает в единицу бит b_0 в RS и тем самым инициирует устройство ввода, которое продвигает перфоленту на один шаг. В ожидании завершения этой операции опрашивается в цикле бит b_7 в RS. Затем, если нет ошибки, код символа переписывается из RD в S. После этого следует возврат управления в прикладную программу.

Программа синхронного драйвера:

```
;          Синхронный драйвер ввода символов с перфоленты
;          ----------------------------------------------------------------
;       Содержит: 1) п/п-у инициализации (вызов из MS-DOS)
;                 2) п/п-у вывода символа (вызов командой INT 0EEh)
;
        ASSUME  CS:_Text
_Text   SEGMENT  PUBLIC  ‘CODE’
        ORG   2Ch
Blocokr  DW  ?                          ; Адрес блока окружения
        ORG   100h
Start:  JMP    Init                     ;  Переход на инициализацию
;
;     Ввод символа - обработчик программного прерывания с номером EEh
;     ---------------------------------------------------------------------------
;     Выходы:  флаг  CF – признак результата (0 – успешно; 1 – ошибка)
;                AL – код символа
Begin: STI                              ; Разрешить маскируемые прерывания
        PUSH  AX                        ; Сохранение содержимого
        PUSH  DS                        ;     регистров в стеке
        MOV   AX, CS                    ; DS будет адресовать данные
        MOV   DS, AX                    ;     резидентной программы
        IN      AL, 90h                 ; Чтение RS в AL
        OR      AL, 00000001b           ; Запуск
        OUT   90h, AL                   ;    устройства
Povt:   IN      AL, 90h                 ; Чтение RS в AL
```

```
        TEST  AL, 10000000b          ; Символ готов?
        JZ    Povt                   ; Нет
        TEST  AL, 01110000b          ; Ошибка есть?
        JNZ   Error                  ; Да
        IN    AL, 91h                ; Прием символа из RD
        CLC                          ; Сбрасывает флаг переноса CF
        JMP   Exit
Error:  STC                          ; Устанавливает флаг переноса CF
Exit:   POP   DS                     ; Восстановление содержимого
        POP   AX                     ;    регистров из стека
        IRET                         ; Возврат в программу из прерывания
;    Инициализационная часть программы
Init:   MOV   AX, 25EEh              ; Запись стартового адреса обработчика
        MOV   DX, OFFSET Begin       ;     в вектор прерываний
        INT   21h                    ;          с номером EEh
        IN    AL, 90h                ; Чтение RS в AL
        AND   AL, 11110111h          ; Запрет прерываний
        OUT   90h, AL                ;    от устройства
        MOV   AH, 49h                ; Освобождение области памяти,
        MOV   ES, Blocokr            ;     занимаемой
        INT   21h                    ;          блоком окружения
        MOV   DX, OFFSET Init        ;  Возврат в DOS, оставшись
        INT   27h                    ;   резидентным
_Text    ENDS
END   Start
```

Инициализационная часть этой программы выполняется в результате запуска программы из MS-DOS. Так как эта часть нерезидентна, то она размещена в конце программы. В начале своего выполнения она помещает стартовый адрес резидентной части в выбранный вектор прерываний, а затем запрещает прерывания от устройства ввода перфоленты, сбросив бит 3 в RS. Далее производится подготовка резидентной части путем освобождения памяти, занимаемой блоком окружения программы. После этого инициализационная часть выполняет возврат в MS-DOS, сопровождаемый просьбой сделать резидентной начало программы (до метки Init).

Резидентная часть программы инициируется в результате программного прерывания с номером EEh. Выполнение этой части начинается так же, как начинается любой обработчик прерываний: 1) разрешение маскируемых прерываний; 2) сохранение содержимого используемых регистров в стеке; 3) запись в DS адреса-сегмента данных резидентной части. После этого производится запуск устройства ввода перфоленты (установкой бита 0 в RS). Далее в цикле опрашивается бит 7 в RS до тех пор, пока он не будет установлен в 1. В зависимости от наличия ошибки, перед возвратом в прикладную программу сбрасывается или устанавливается флаг FC.

5.3. Асинхронный ввод-вывод с прерываниями

5.3.1. Контроллер прерываний

При асинхронном вводе-выводе с прерываниями драйвер ожидает завершения текущей операции ввода-вывода пассивно, не занимая для этого ЦП. Поэтому программа, сделавшая запрос на ввод-вывод, может выполнять в это же время (то есть параллельно) какую-то другую свою работу на ЦП. При завершении операции ввода-вывода ИУ выдает в ЦП сигнал прерывания.

Прерывания от ИУ относятся к маскируемым внешним аппаратным прерываниям (см.п.2.6.1). В большинстве ВС сигналы таких прерываний поступают не сразу в ЦП, а сначала в ***программируемый контроллер прерываний***. Данный контроллер упорядочивает поступление сигналов внешних прерываний в ЦП, освобождая его от необходимости наводить порядок среди этих сигналов. Далее речь будет идти о программируемом контроллере прерываний i8259A.

Контроллер i8259A имеет 8 входов – IRQ0, IRQ1…IRQ7, позволяющих принимать запросы на прерывание от восьми ИУ:

IRQ0 – таймер;
IRQ1 – клавиатура;
IRQ2 – второй контроллер прерываний;
IRQ3 – последовательный порт COM2;
IRQ4 – последовательный порт COM1;
IRQ5 – параллельный порт LPT2;
IRQ6 – гибкий диск;
IRQ7 – параллельный порт LPT1.

Данные входы имеют приоритеты: чем меньше номер входа, тем приоритет выше. Поэтому самыми приоритетными являются сигналы прерываний от таймера. Приоритет сигнала используется контроллером прерываний для определения возможности направления этого сигнала в ЦП. При этом, если в контроллер одновременно поступили несколько сигналов прерываний, то из них в ЦП будет отправлен тот сигнал, приоритет которого выше.

Кроме того, если ЦП уже занят обработкой какого-то прерывания, то новый сигнал будет отправлен контроллером прерываний только в том случае, если его приоритет выше, чем у того прерывания, обработчик которого выполняется на ЦП. (Вспомним, что для того, чтобы новое прерывание действительно прервало обработку старого прерывания, дополнительно требуется, чтобы в начале обработчика старого прерывания стояла команда STI.)

Номер прерывания, соответствующий конкретному ИУ, можно определить следующим образом: к базовому номеру, соответствующему кон-

троллеру прерываний (8), следует прибавить номер входа в этот контроллер. Например, номер прерывания от клавиатуры: N = 8 + 1 = 9.

Как и любое ИУ, контроллер прерываний имеет порты, которые используются программами (драйверами) для управления им. Эти порты имеют адреса 20h и 21h. Порт 21h используется, в частности, для маскирования (то есть для запрета) прерываний от устройств требуемого типа. Для этого требуется записать единицу в тот бит порта, который соответствует номеру входа для устройства. Например, следующие две команды выполняют маскирование прерываний от клавиатуры:

```
MOV    AL, 00000010b          ; Вход для клавиатуры - IRQ1
OUT    21h, AL
```

В конце обработчика внешних прерываний следует разрешить обработку прерываний с более низкими приоритетами. Для этого достаточно выполнить две команды:

```
MOV    AL, 20h                ; Число 20h используется для двух
OUT    20h, AL                ;  разных величин
```

Существуют другие команды управления контроллером прерываний, позволяющие размаскировать прерывания от любого ИУ (независимо от маскирования в ЦП), а также изменять приоритеты ИУ. Аналогичные команды используются и для управления вторым (ведомым) контроллером прерываний.

Ведомый контроллер подсоединяется к входу IRQ2 ведущего контроллера. Поэтому ИУ, обслуживаемые ведомым контроллером, выдают сигналы прерываний с приоритетами, расположенными между приоритетами клавиатуры и последовательного порта COM2. Для управления ведомым контроллером используются порты A0h и A1h, аналогичные портам 20h и 21h для ведущего контроллера. Базовый номер прерываний, соответствующий ведомому контроллеру прерываний, равен 70h.

Перечислим два типа устройств, обслуживаемых ведомым контроллером прерываний (с указанием его входов):

IRQ13 – математический сопроцессор;

IRQ14 – жесткий магнитный диск.

5.3.2. Алгоритм обработки прерываний

Использование при выдаче внешних маскируемых прерываний контроллера прерываний требует уточнения алгоритма обработки прерываний, рассмотренного в п.2.6.4 :

1) поступление сигнала прерывания в ЦП. Для этого должны быть выполнены действия:

– ИУ посылает сигнал прерывания в i8259A;

– если данный тип прерываний незамаскирован, i8259A выдает сигнал прерывания на вход INTR процессора;

2) собственно прерывание. Оно выполняется ЦП по окончанию выполнения текущей машинной команды и включает действия:

– ЦП возвращает в ИУ сигнал подтверждения;

– ИУ передает по ОШ в ЦП номер прерывания N;

– ЦП помещает текущее содержимое FLAGS, CS и IP в программный стек;

– в IP загружается слово ОП, имеющее адрес 4*N, а в CS – 4*N +2;

– ЦП запрещает маскируемые прерывания путем сброса в нуль флажка IF в регистре FLAGS;

3) начальный этап программной обработки прерывания. Он выполняется обработчиком прерываний и включает действия:

– разрешение маскируемых прерываний с помощью команды STI;

– сохранение в программном стеке содержимого тех регистров, с которыми будет работать программа обработчика;

– запись в сегментный регистр данных DS значения, которое соответствует адресу-сегменту данных обработчика прерываний;

4) действия обработчика прерываний, определяемые типом прерывания;

5) завершение обработки прерывания. Сюда относятся действия:

– запись обработчиком прерываний кода 20h в порт 20h;

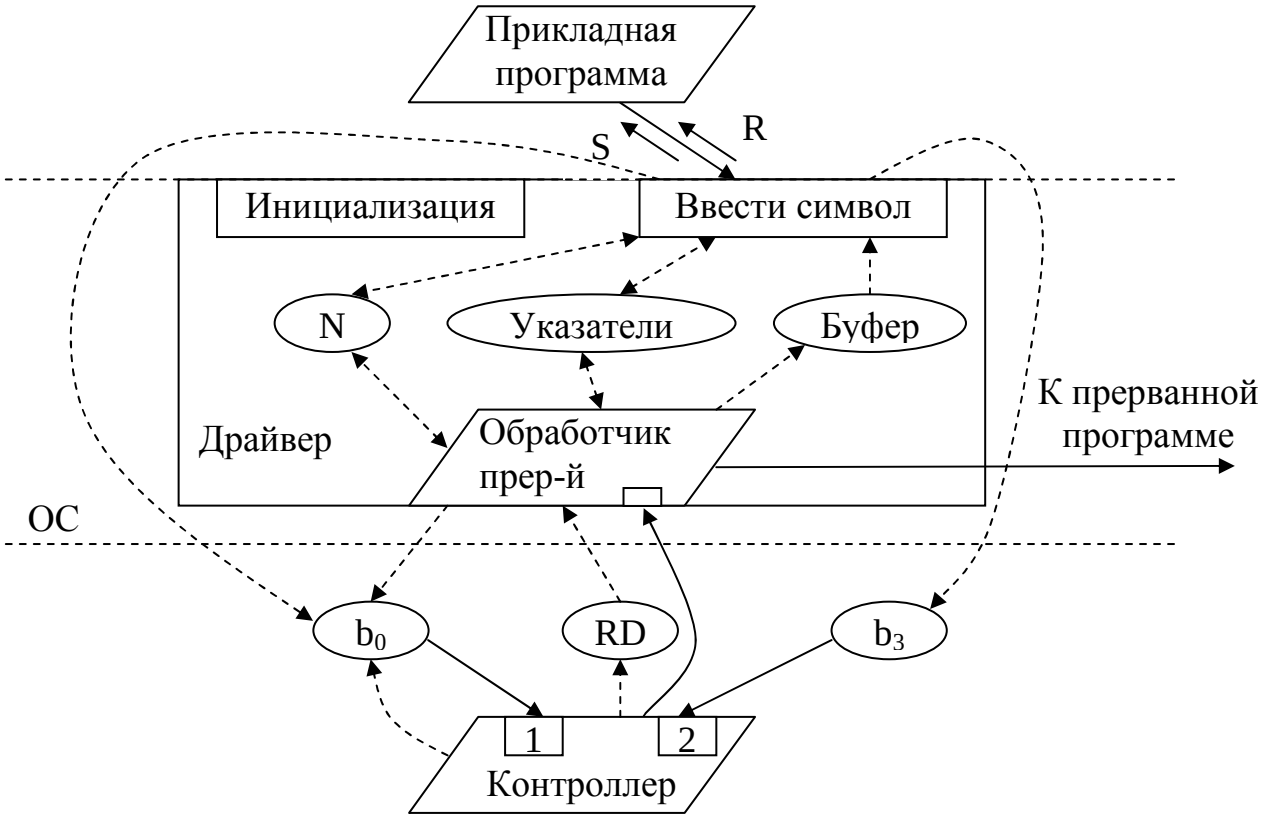
– восстановление содержимого регистров, запомненного ранее в стеке;

– выполнение обработчика прерываний завершается командой IRET, которая извлекает из стека прежнее содержимое IP, CS и FLAGS, тем самым возвращая управление в прерванную программу.

5.3.3. Пример драйвера

В качестве примера опять рассмотрим драйвер ввода с перфоленты. Для связи с ИУ используются порты 90h (RS) и 91h (RD). Структура RS приведена на рис.30. Допустим, что устройству ввода с перфоленты соответствует вектор прерываний с номером 60h. Кроме того, предположим, что контроллер данного устройства соединен с входом IRQ5 ведущего контроллера прерываний.

Логическая схема драйвера представляет собой активный пакет (данный термин поясняется в приложении), состоящий из двух интерфейсных логических процедур, логического процесса и трех структур данных (рис.32).



S – код символа;
R – признак результата (0 – успешно, 1 - символа пока нет);
N – число занятых позиций в буфере;
b_0 – бит разрешения работы;
b_3 – бит разрешения прерываний.

Рис.32. Логическая схема асинхронного драйвера с прерываниями

Логическая процедура “Инициализация” без параметров. Она выполняет три действия: 1) делает остальные модули драйвера резидентными; 2) заполняет вектор прерываний с номером EEh (для последующего инициирования резидентной процедуры «Ввести символ»; 3) заполняет вектор прерываний с номером 60h (для последующего инициирования обработчика прерываний от перфоленты).

Логическая процедура «Ввести символ» обслуживает запросы прикладных и системных программ по вводу символов с перфоленты. Обработчик прерываний перфоленты – логический процесс без параметров. Рассмотрим совместную работу перечисленных модулей по логической схеме.

После своего вызова логическая процедура «Ввести символ» проверяет значение N. Если N = 0 (в буфере ничего нет), то устанавливаются в «1» биты b_0 и b_3 RS. После этого данная логическая процедура выполняет возврат в вызвавший ее модуль, то есть в прикладную программу, с признаком результата R = 1.

Значение b_0 = 1 инициирует работу контроллера (вход 1) по вводу очередного символа. Контроллер помещает код символа в RD (сбрасывая при этом бит b_0 RS) и оказывается в состоянии “вход 2”. Так как бит разрешения прерывания b_3 = 1, то контроллер инициируется по данному входу и выдает сигнал прерывания. Обработчик прерываний считывает код символа из RD в буфер и увеличивает значение N на единицу. Если при этом буфер оказывается полным ($N = N_{MAX}$), то управление сразу возвращается прерванной программе. Иначе сначала установкой b_0 = 1 инициируется работа контроллера по вводу следующего символа.

Если при вызове логической процедуры «Ввести символ» $0<N<N_{MAX}$, то из буфера выбирается очередной символ и устанавливается R = 0. Если $N=N_{MAX}$, то перед выполнением данных действий устанавливается $b_0=1$ (этот бит не был установлен обработчиком прерываний).

В основе алгоритмов логической процедуры «Ввести символ» и обработчика прерываний лежат операции с буфером. Обычно используют буфер в виде несвязанной циклической очереди. Подобно стеку, ***очередь*** является линейным списком, исключение элементов из которого производится только из начала списка. Но в отличие от стека добавление нового элемента производится не в начало, а в конец списка. Поэтому для работы с очередью используются два указателя, один из которых (S) указывает на начало очереди, а второй (F) – на ее конец. В отличие от обычной несвязанной очереди, ***циклическая несвязанная очередь*** позволяет лучше использовать пространство памяти. Поясним ее организацию на примере.

Пусть имеется область памяти длиной десять ячеек, предназначенная для размещения буфера (рис.33). Кроме того, есть две ячейки ОП, являющиеся указателями: F указывает на очередную пустую ячейку буфера, в которую можно занести новый элемент (символ), а S указывает на ячейку, содержащую первый из еще не взятых элементов. Цикличность буфера заключается в том, что при достижении максимального граничного адреса (на рис.33 этот адрес равен 9) указатель корректируется так, что вместо 9 он будет содержать 0 (то есть будет указывать на начало области памяти).

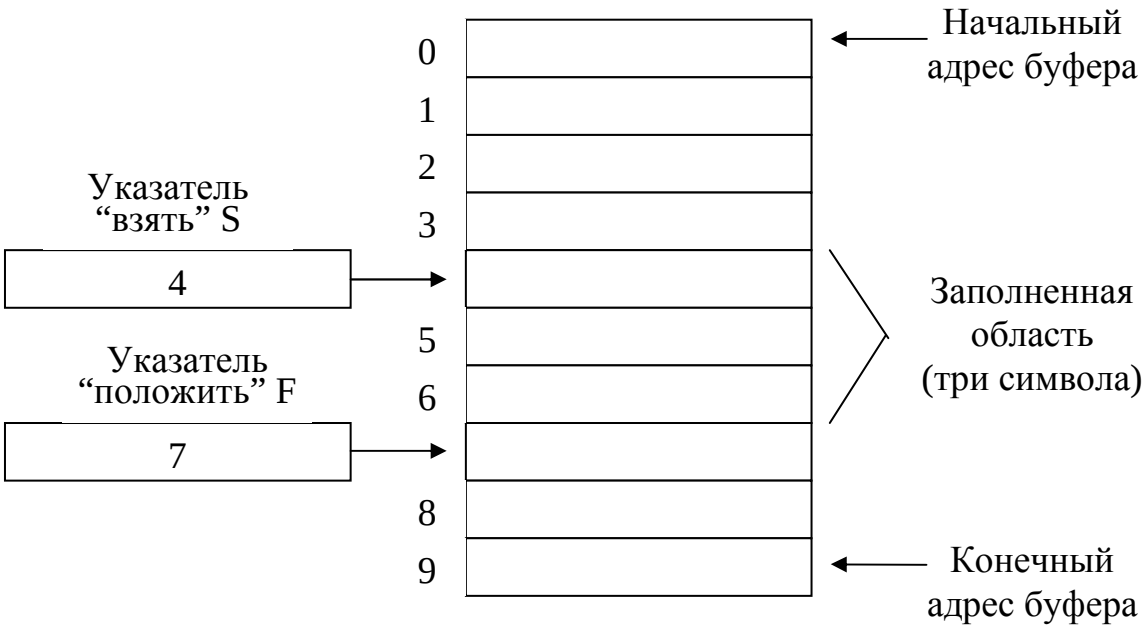


Рис.33. Циклический буфер и указатели

Заметим, что если указатель «взять» движется в сторону больших адресов (на рис.33 вниз) и становится равным указателю «положить», то очевидно, что буфер пуст, и счетчик элементов равен нулю. А если указатель «положить» «догонит» указатель «взять», то буфер полон и счетчик элементов равен максимально возможному числу (10).

Программа асинхронного драйвера с прерываниями:

```
;          Асинхронный драйвер ввода символов с перфоленты
;          ------------------------------------------------------------------
;       Содержит: 1) п/п-а инициализации (вызов из MS-DOS)
;                 2) п/п-а вывода символа (вызов командой INT 0EEh)
;                 3) обработчик прерываний перфоленты (инициируется аппаратно)
;
        ASSUME  CS:_Text
_Text  SEGMENT  PUBLIC  'CODE'
Nmax   EQU   20                          ; Емкость буфера = 20 символов
        ORG   2Ch
Blocokr  DW  ?                           ; Адрес блока окружения
        ORG   100h
Start:  JMP   Init                       ;  Переход на инициализацию
;       Определение данных
N       DB    0                          ; Счетчик символов в буфере
S       DW    0                          ; Указатель «взять»
F       DW    0                          ; Указатель «положить»
Buf     DB    Nmax  DUP (0)              ; Буфер
```

```
;       Ввод символа - обработчик программного прерывания с номером EEh
;       ---------------------------------------------------------------------------------
;       Выходы:  флаг  CF – признак результата (0 – успешно; 1 – символа пока нет)
;                AL – код символа
Begin: STI                              ; Разрешить маскируемые прерывания
       PUSH  AX                         ; Сохранение
       PUSH  BX                         ;       содержимого
       PUSH  DS                         ;            регистров в стеке
       MOV   AX, CS                     ; DS будет адресовать данные
       MOV   DS, AX                     ;     резидентной программы
       CMP   N, 0                       ; Буфер пуст ?
       JNZ   Simv                       ; Если не пуст, переход
       IN    AL, 90h                    ; Чтение RS в AL
       OR    AL, 00001001b              ; Запуск устройства
       OUT   90h, AL                    ;     и разрешение прерываний от п/л
       STC                              ; Установка флага переноса CF
       JMP   Exit1
Simv:  CMP   N, Nmax                    ; Буфер полон ?
       JNZ   M1                         ; Если не полон, переход
       IN    AL, 90h                    ; Чтение RS в AL
       OR    AL, 00000001b              ; Запуск
       OUT   90h, AL                    ;    устройства
M1:    MOV   BX, S                      ; Запись в AL
       MOV   AL, Buf[BX]                ;    символа из буфера
       DEC   N                          ; Уменьшение счетчика символов на 1
       CMP   S, Nmax –1                 ; S указывает на конец буфера ?
       JZ    M2                         ; Если да, то переход
       INC   S                          ; Увеличение S на 1
       JMP   M3
M2:    MOV   S, 0                       ; S ← 0
M3:    CLC                              ; Сбрасывает флаг переноса CF
Exit1: POP   DS                         ; Восстановление содержимого
       POP   BX                         ;     регистров
       POP   AX                         ;          из стека
       IRET                             ; Возврат в программу из прерывания
;
;   Обработчик аппаратных прерываний с номером 60h от устройства ввода перфоленты
;   ------------------------------------------------------------------------------------------------------
Obrpr:  STI                             ; Разрешить маскируемые прерывания
       PUSH  AX                         ; Сохранение
       PUSH  BX                         ;       содержимого
       PUSH  DS                         ;            регистров в стеке
       MOV   AX, CS                     ; DS будет адресовать данные
       MOV   DS, AX                     ;     резидентной программы
       IN    AL, 91h                    ; Прием символа из RD в AL
       MOV   BX, F                      ; Запись символа из
       MOV   Buf[BX], AL                ;    AL в буфер
       INC   N                          ; Увеличение счетчика символов на 1
       CMP   F, Nmax –1                 ; F указывает на конец буфера ?
       JZ    L1                         ; Если да, то переход
```

```
        INC   F                     ; Увеличение F на 1
        JMP   L2
L1:     MOV   F, 0                  ; F ← 0
L2:     CMP   N, Nmax               ; Буфер полон ?
        JZ    Exit2                 ; Если да, то переход
        IN    AL, 90h               ; Чтение RS в AL
        OR    AL, 00000001b         ; Запуск
        OUT   90h, AL               ;    устройства
Exit2:  MOV     AL, 20h             ; Разрешение менее приоритетных
        OUT     20h, AL             ;      прерываний
        POP   DS                    ; Восстановление содержимого
        POP   BX                    ;     регистров
        POP   AX                    ;        из стека
        IRET                        ; Возврат в программу из прерывания
;
;    Инициализационная часть программы
Init:   MOV   AX, 25EEh             ; Запись стартового адреса обработчика
        MOV   DX, OFFSET Begin      ;     в вектор прерываний
        INT   21h                   ;          с номером EEh
        MOV   AX, 2560h             ; Запись стартового адреса обработчика
        MOV   DX, OFFSET Obrpr      ;     в вектор прерываний
        INT   21h                   ;          с номером 60h
        MOV   AH, 49h               ; Освобождение области памяти,
        MOV   ES, Blocokr           ;     занимаемой
        INT   21h                   ;          блоком окружения
        MOV   DX, OFFSET Init       ; Возврат в DOS, оставшись
        INT   27h                   ;   резидентным
_Text    ENDS
END   Start
```

Приведенная программа состоит из трех модулей, предваряемых данными, совместно используемыми этими модулями. Первый по порядку модуль начинается с метки Begin и представляет собой программную реализацию логической процедуры «Ввести символ». Данный модуль выполнен как резидентный обработчик программных прерываний с номером EEh. (Напомним, что в качестве этого номера разработчик может взять любой неиспользуемый номер.)

Второй модуль начинается с метки Obrpr и представляет собой программную реализацию логического процесса «Обработчик прерываний перфоленты». Этот модуль выполнен как резидентный обработчик внешних аппаратных прерываний с номером 60h.

Третий модуль начинается с метки Init и представляет собой программную реализацию логической процедуры “Инициализация”. Этот модуль выполнен как фрагмент программы, инициируемый при запуске программы из MS-DOS.

5.4. Прямой доступ в память

В рассмотренных выше драйверах для передачи каждого байта данных между ИУ и ОП приходится выполнять несколько машинных команд. Если в программе требуется выполнить ввод-вывод не одного, а целой последовательности байтов, то затраты времени ЦП на их передачу будут весьма существенны. Например, обмен данными с магнитным диском производится посекторно (см. п.4.1), то есть блоками по 512 байт.

Применение специального ***контроллера прямого доступа в память*** **(ПДП)** позволяет существенно снизить затраты времени ЦП, так как программа драйвера будет выполнять операции инициирования и завершения ввода-вывода не каждого байта, а лишь всего блока. В процессе ввода-вывода с использованием ПДП обмен байтами данных между ИУ и ОП производится не через регистр RD и регистры ЦП, а через контроллер ПДП. При этом роль ЦП ограничивается выдачей разрешений на занятие ОШ (а точнее – шины данных).

Обычно контроллер ПДП рассчитан на одновременное обслуживание нескольких ИУ (каждое ИУ подключается к нему набором проводников). Часть контроллера ПДП, предназначенная для обслуживания одного ИУ, называется ***каналом*** **ПДП**. Например, совместно с ЦП типа i8086 наиболее часто используется 4-х канальный контроллер ПДП i8237. Этот контроллер расположен на системной плате и имеет следующее назначение каналов:

0 – предназначен для «освежения» памяти. Он постоянно восстанавливает заряд ячеек ОП;

1 – свободен;

2 – обмен с контроллером гибких дисков;

3 – обмен с контроллером жестких дисков.

Контроллер i8237 имеет общие для всех каналов регистры управления, а также для каждого канала свои регистры канала. В отличие от обычных RS, ***регистры управления*** не имеют битов состояния. К таким регистрам относятся регистр режима и регистр масок. ***Регистр режима*** (порт 0Bh) – 8-битный регистр (рис.34). Поясним смысл двух терминов: автоинициализация и тип передачи.

Автоинициализация. После завершения обычной передачи используемый канал ПДП маскируется и должен быть перепрограммирован для дальнейшей работы с ним. При автоинициализации маскировка канала после окончания передачи не происходит и контроллер ПДП полностью готов для повторения обмена по данному каналу.

Тип передачи. Контроллер ПДП поддерживает четыре различных режима передачи:

1) режим одиночной передачи – для передачи каждого байта в ОП (из ОП) контроллер ПДП запрашивает доступ к ОШ у ЦП;

2) режим блочной передачи – запрос на занятие ОШ делается один раз для всего блока;

3) режим передачи по требованию – контроллер ПДП ожидает подготовку байта контроллером устройства и делает запрос на занятие ОШ;

4) каскадный режим – используется в том случае, если ЭВМ имеет более одного контроллера ПДП.

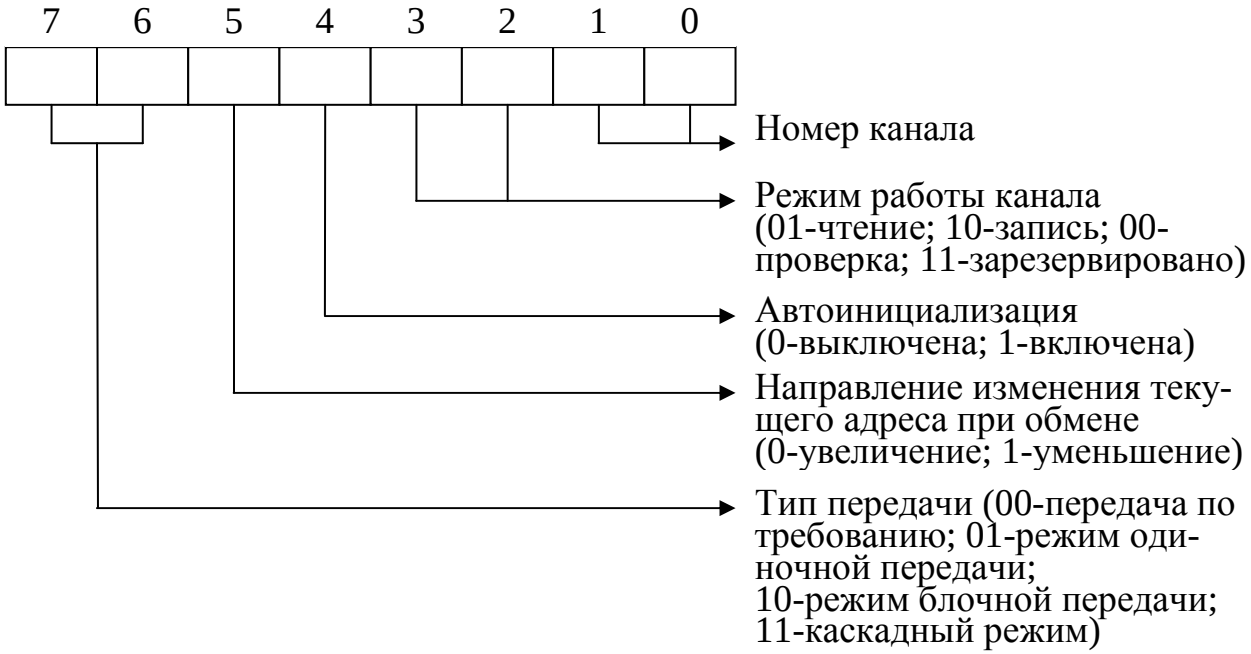


Рис.34. Структура регистра режима

Регистр масок – 4-х битный регистр. Значение одного бита маскирует (1) или демаскирует (0) соответствующий канал. Доступ к регистру масок осуществляется через 8-битный порт 0Ah (рис.35).

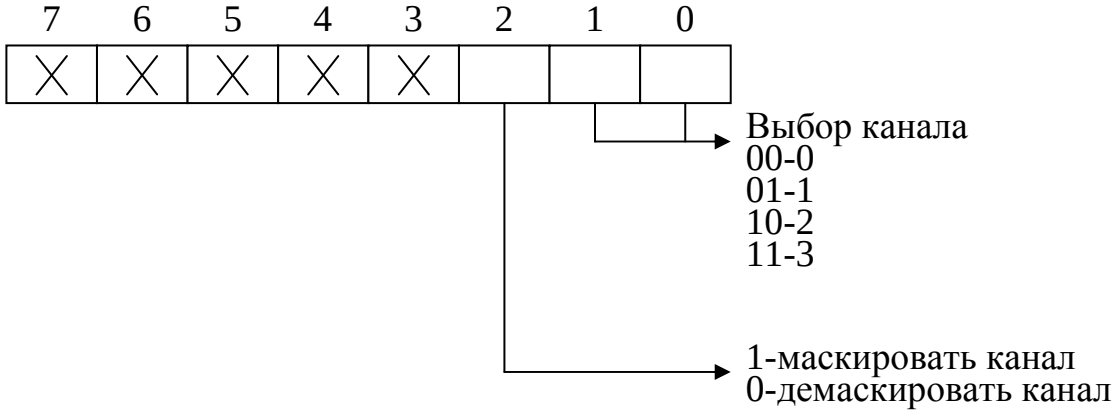


Рис.35. Структура порта для доступа к регистру масок

Регистры канала предназначены для размещения численных параметров предстоящего обмена между ИУ и ОП. Это регистры:

1) ***регистр начального адреса*** – содержит 16 младших битов начального адреса буфера – области ОП, в которую будет считываться (при чтении) блок байтов с ПУ. При записи, наоборот, информация из буфера будет переписываться на ПУ;

2) ***регистр страницы*** – содержит четыре старших бита 20-ти битового адреса буфера, с которым будет производиться обмен;

3) ***регистр счетчика –*** содержит число байтов, которое предстоит передать между ИУ и ОП, уменьшенное на 1.

Номера портов, используемых для доступа к регистрам канала, зависят от номера канала. Например, запись в регистры канала 2 производится через следующие 8-битные порты: регистр начального адреса – порт 4h, регистр страницы – порт 81h, регистр счетчика – порт 5h. Так как регистры начального адреса и счетчика 16-битные, то запись в них через 8-битные порты имеет следующую особенность.

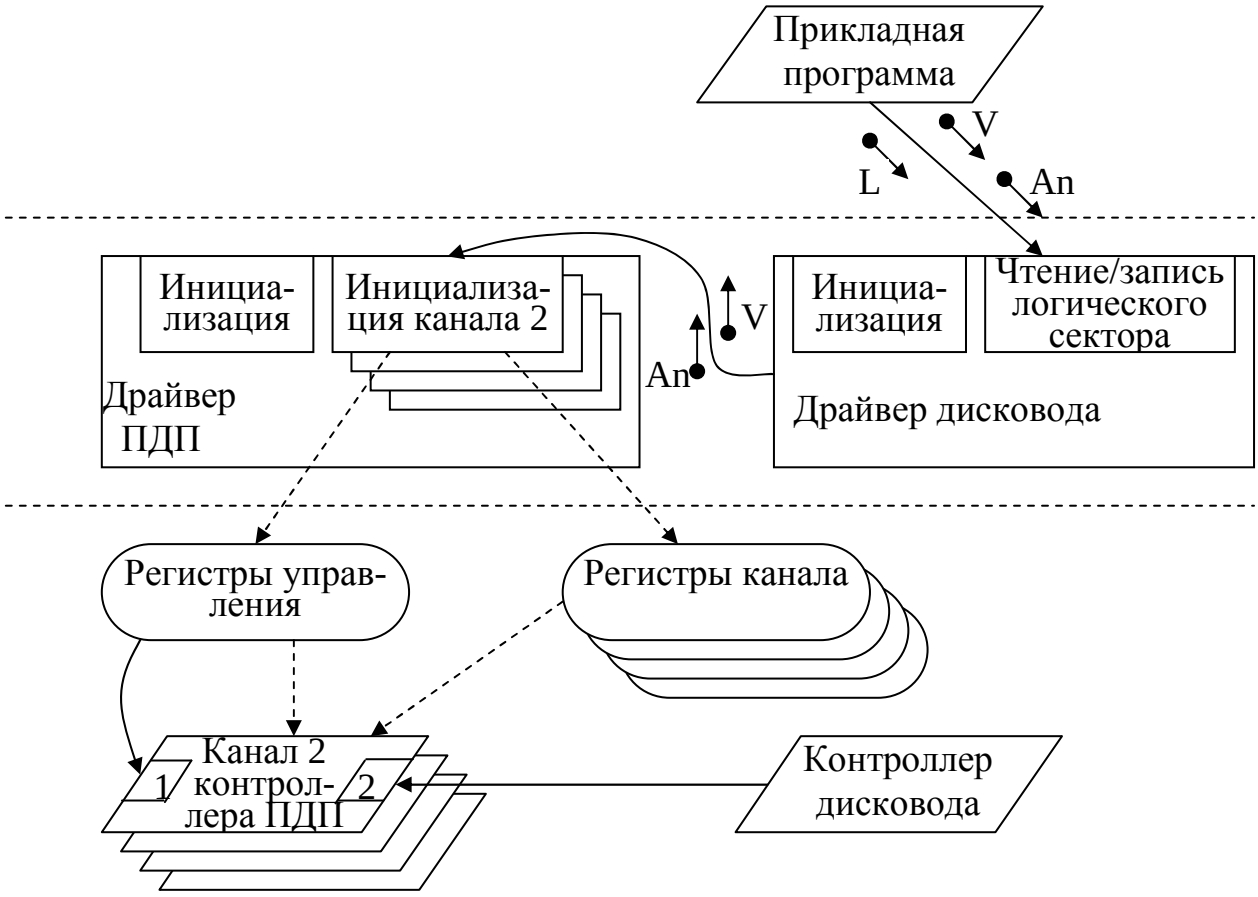
Специальный регистр, называемый «защелкой», направляет содержимое 8-битного порта в младший или старший байт 16-битного регистра канала. Перед записью младшего байта в порт, соответствующий регистру канала, необходимо записать какое-нибудь значение в порт 0Ch, что приведет к сбросу «защелки». После записи младшего байта в регистр канала, установка «защелки» производится автоматически, и следующий байт из порта переписывается в качестве старшего байта регистра канала.

Управление контроллером ПДП рассмотрим на примере использования его для информационного обмена с контроллером гибких дисков. Соответствующая логическая схема приведена на рис.36. (На этой схеме не показаны внутренние модули драйвера дисковода, а также опущены взаимосвязи между драйвером и контроллером дисковода.)

Драйвер дисковода предоставляет прикладным и системным программам возможность выполнять информационный обмен с установленным на дисковод магнитным диском. В результате одного обращения к драйверу производится чтение или запись одного сектора диска, то есть блока из 512 байтов, имеющего номер L ($0 \leq L \leq L_{MAX}$).

Получив от прикладной программы запрос на чтение или запись сектора, драйвер дисковода вызывает логическую процедуру «Инициализация канала 2», входящую в состав драйвера ПДП. Данная процедура имеет два входных параметра: «Начальный адрес прикладного буфера» и «Тип операции» (0 – чтение, 1 – запись). В процессе выполнения процедуры производится запись в регистры управления и в регистры канала 2. В частности, в регистры начального адреса и страницы помещается начальный адрес прикладного буфера, а в регистр счетчика – число передаваемых байтов, уменьшенное на единицу.

После завершения инициализации канал 2 контроллера ПДП переходит в состояние «Вход 2», ожидая прихода от контроллера дисковода команды «Начать передачу». Эта команда инициирует работу контроллера ПДП по побайтовой передаче данных между контроллером дисковода и ОП.



V – тип операции (0 – чтение, 1 – запись)
An – начальный адрес прикладного буфера
L – номер логического сектора

Рис.36. Логическая схема применения ПДП для информационного обмена с дисководом

Для передачи очередного байта в ОП (при чтении) или из ОП (при записи) контроллер ПДП выдает в ЦП запрос на занятие ОШ. Далее, при получении согласия он выводит содержимое регистров адреса и страницы на шину адреса, а в контроллер дисковода передает сигнал о том, чтобы тот вывел на шину данных очередной передаваемый байт (при чтении) или считал байт с шины данных (при записи). После этого контроллер ПДП

увеличивает (или уменьшает) на единицу содержимое регистра адреса и уменьшает на единицу счетчик байтов. При достижении счетчиком нуля контроллер ПДП или выдает в ЦП сигнал прерывания, или сообщает о завершении передачи в контроллер дисковода.

Логическая процедура «Инициализация» выполняет две функции: 1) делает процедуры инициализации каналов резидентными; 2) заполняет векторы прерываний для доступа к этим резидентным процедурам. Программная реализация логической процедуры «Инициализация» аналогична реализации соответствующих процедур, рассмотренных ранее (см. п. 5.2 и 5.3).

Логическая процедура «Инициализация канала 2» включает следующие шаги:

Шаг 1. В зависимости от типа операции (чтение или запись) сделать запись в регистр режима. При этом следует учесть следующие характеристики режима:

- автоинициализация выключена;
- текущий адрес при обмене увеличивается;
- тип передачи одиночный.

Шаг 2. Вычисление полного 20-битного адреса буфера в ОП и загрузка этого адреса в регистр начального адреса и в регистр страницы. Для вычисления полного адреса необходимо просуммировать адрес-сегмент, предварительно умноженный на 16, с адресом-смещением. Полученные четыре старших бита адреса записываются в регистр страницы, а остальные 16 битов – в регистр адреса.

Шаг 3. Запись в регистр счетчика числа байтов, предназначенных для пересылки, уменьшенного на единицу, то есть числа 511.

Шаг 4. Запись в регистр маски контроллера ПДП числа, разрешающего работу канала 2.

Шаг 5. Возврат из процедуры.

Ниже приведен фрагмент программы драйвера ПДП, выполняющий инициализацию канала 2. Так как данный фрагмент резидентен, то он вызывается через программное прерывание. В качестве номера прерывания выбрано число EDh.

```
; Подготовка канала 2  ПДП к чтению-записи сектора гибкого диска – обработчик
;              программного прерывания с номером EDh
; ---------------------------------------------------------------------------------------------------
;  Входы:      DX:BX – адрес буфера (сегмент:смещение)
;              AL – тип операции (0 – чтение, 1 – запись)
;
        STI                             ; Разрешить маскируемые прерывания
        PUSH  AX                        ; Сохранение
        PUSH  BX                        ;       содержимого
        PUSH  CX                        ;            регистров
        PUSH  BP                        ;                 в стеке
        CMP   AL, 0                     ; Операция – чтение ?
```

```
        JNZ   Sap                ; Нет
        MOV   AL, 01000110b      ; Задание режима для
        OUT   0Bh, AL            ;    чтения диска
        JMP   Adr
Sap:    MOV   AL, 01001010b      ; Задание режима для
        OUT   0Bh, AL            ;      записи на диск
Adr:    MOV   AX, DX             ; Адрес-сегмент → AX
        MOV   BP, 16
        MUL   BP                 ;  (AX)*(BP) → DX:AX
        ADD   BX, AX             ; Младшие 16 бит адреса → BX
        MOV   CX, 0              ;   0 → CX
        ADC   DX, CX             ; Старшие биты адреса → DX
        MOV   AL, DL             ; Запись в
        OUT   81h, AL            ;   регистр страницы
        OUT   0Ch, AL            ; Сброс «защелки»
        MOV   AL, BL             ; Запись младшего байта
        OUT   4h, AL             ;    в регистр начального адреса
        MOV   AL, BH             ; Запись старшего байта
        OUT   4h, AL             ;    в регистр начального адреса
        OUT   0Ch, AL            ; Сброс «защелки»
        MOV   AX, 511            ; Запись младшего байта
        OUT   5h, AL             ;    счетчика байтов
        MOV   AL, AH             ; Запись старшего байта
        OUT   5h, AL             ;    счетчика байтов
        MOV   AL, 2              ; Размаскирование
        OUT   0Ah, AL            ;    канала 2
        POP   BP                 ; Восстановление
        POP   CX                 ;      содержимого
        POP   BX                 ;           регистров
        POP   AX                 ;              из стека
        IRET                     ; Возврат в программу из прерывания
```

5.5. Асинхронный вывод с общей памятью

5.5.1. Видеоадаптер

Принцип асинхронного вывода с общей памятью заключается в том, что ЦП и ИУ имеют доступ к общей области памяти, используя ее в качестве буфера для информационного обмена между ними. При этом операции записи и чтения с общей памятью ЦП и ИУ производят асинхронно, то есть независимо друг от друга. Данный принцип используется широко для управления экраном (дисплеем).

Вспомним (см. п.3.3), что часть адресного пространства ОП (112 К, начиная с адреса A0000h) используется для адресации видеопамяти. Она является той самой областью памяти, с которой ЦП и ИУ работают асинхронно. ***Видеопамять*** входит в состав ИУ экрана, называемого ***видеоадаптером***. Конструктивно такой адаптер представляет собой печатную плату, вставляемую в разъем расширения компьютера. Среди сотен суще-

ствующих видеоадаптеров нами будет рассматриваться только один – **CGA** (Color Graphic Adapter). Являясь одним из первых видеоадаптеров (создан в 1981 году), CGA имитируется аппаратно многими другими, более современными видеоадаптерами.

Видеоадаптер (в том числе и CGA) имеет два принципиально разных режима работы – текстовый и графический. Для каждого из этих режимов CGA имеет несколько форматов. Примером формата для текстового режима является формат **80*25** – 25 строк по 80 символов в каждой строке.

Центральным модулем CGA является контроллер экрана **CRT**. Он устанавливает и поддерживает режим работы экрана, выполняет основную работу по интерпретации кодов ASCII, а также управляет курсором. Кроме контроллера экрана, CGA имеет порты ввода-вывода, ПЗУ с матрицей знаков, видеопамять, а также 18 регистров управления (рис.37).

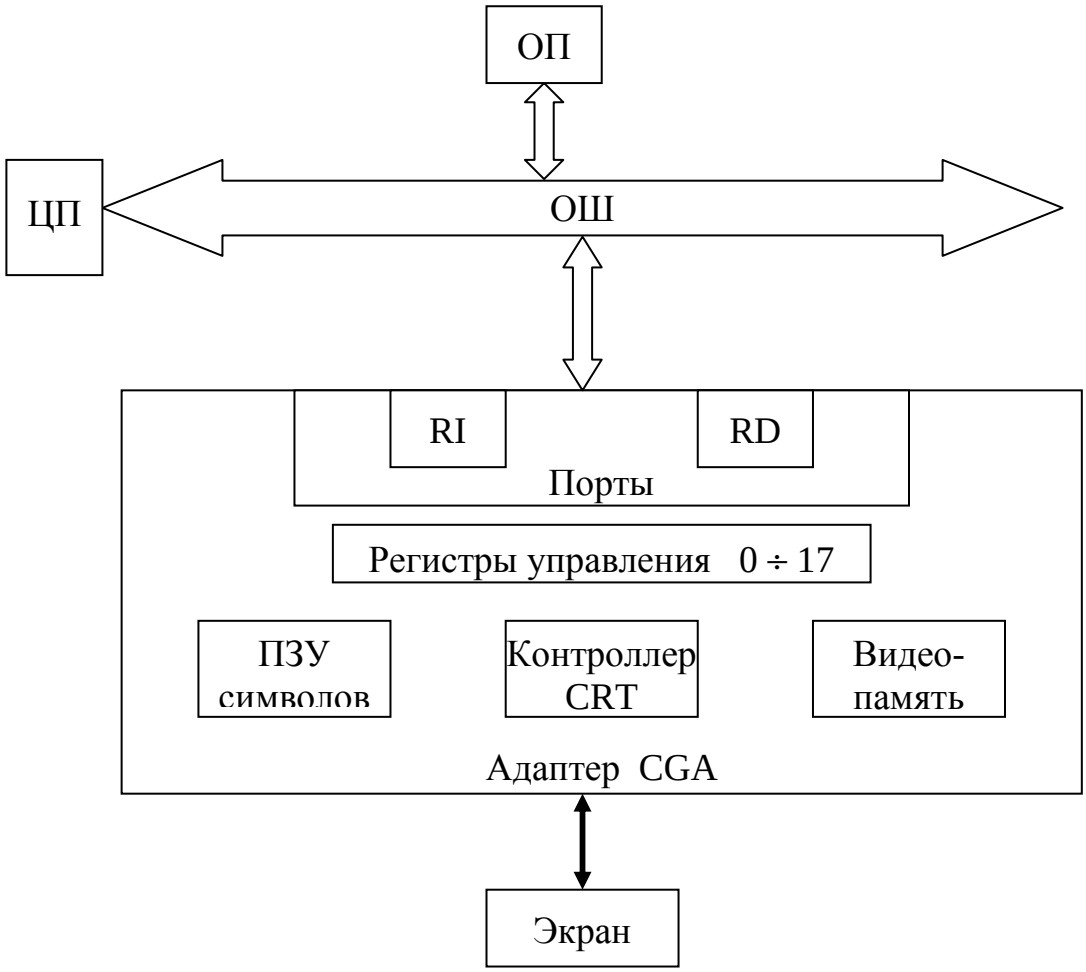


Рис.37. Структура адаптера дисплея

В зависимости от выполняемых функций все 18 управляющих регистра контроллера можно разделить на две группы:

1) регистры, фиксирующие горизонтальные и вертикальные параметры экрана;

2) регистры визуализации.

К первой группе относятся первые десять регистров с номерами от 0 до 9. Они устанавливаются один раз при задании режима работы экрана. (Это делает BIOS при включении ЭВМ в сеть.) Далее будем считать, что к началу выполнения любых программ на ЦП экран установлен в текстовый режим 80*25.

Регистры визуализации приведены в таблице 1. Они 8-битные. Некоторые из них связаны в пары, чтобы хранить 16-битовые величины. Все регистры визуализации можно разделить на три группы:

1) регистры управления курсором;

2) регистры светового пера;

3) регистры начального адреса сканирования. Этот адрес представляет собой порядковый номер в видеопамяти того символа, который будет выводиться в верхнем левом углу экрана. Например, это может быть номер 0.

Таблица 1

Регистры визуализации

Номер регистра	**Назначение регистра**	**Операции**
10	начало курсора	запись
11	конец курсора	запись
12	начальный адрес сканир-я (ст.)	запись
13	начальный адрес сканир-я (мл.)	запись
14	адрес курсора (ст.)	чтение/запись
15	адрес курсора (мл.)	чтение/запись
16	световое перо (ст.)	чтение
17	световое перо (мл.)	чтение

Что касается портов, то их в CGA два (см. рис.37):

1) порт с адресом 3D4h, называемый также индексным регистром RI;

2) порт с адресом 3D5h, называемый также регистром данных RD.

Регистр RI используется для задания требуемого управляющего регистра. А регистр RD используется для записи нового содержимого того управляющего регистра, на который указывает RI. Ниже будет рассмотрено применение этих портов для управления курсором.

5.5.2. Видеопамять

Видеопамять в CGA имеет размер 16 Кбайт и начинается по адресу B8000h (соответствующий логический адрес - B800h:0000h). Каждый символ занимает в видеопамяти два байта. В младшем из этих байтов содер-

жится код ASCII символа (информация о том, что выводить), а в старшем байте – атрибуты символа (информация о том, как выводить). Структура байта атрибутов приведена на рис.38. В таблице 2 приведены цвета, которые можно получить, задавая комбинации битов красного, зеленого и синего цветов, а также интенсивности.

На экране одновременно могут находиться 2000 символов (80*25). Эти символы занимают 4000 байтов (около 4 Кбайт) видеопамяти. Такая область памяти, занимаемая данными для вывода экрана, называется ***дисплейной страницей***. Нетрудно подсчитать, что в видеопамяти CGA объемом 16К могут разместиться 4 дисплейных страницы. При этом данные экрана располагаются в памяти построчно (начиная с верхнего левого угла), а небольшие пустоты между страницами усекаются аппаратно (рис.39).

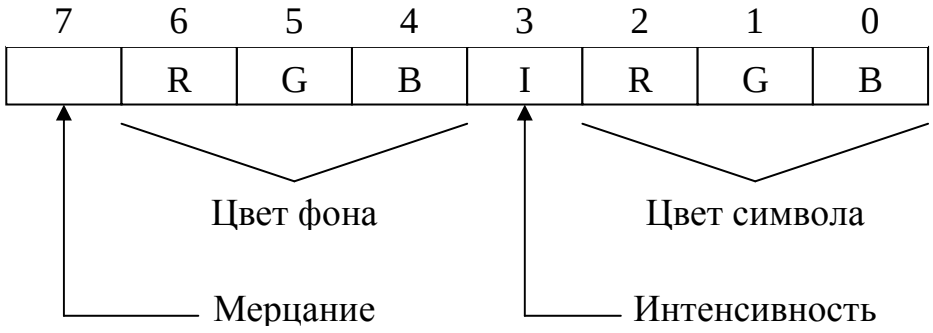


Рис.38. Структура байта атрибутов

Таблица 2

Генерация цвета

R	**G**	**B**	**I**	**Цвета**
0	0	0	0	черный
0	0	1	0	синий
0	1	0	0	зеленый
0	1	1	0	голубой
1	0	0	0	красный
1	0	1	0	сиреневый
1	1	0	0	коричневый
1	1	1	0	белый
0	0	0	1	серый
0	0	1	1	ярко - синий
0	1	0	1	ярко - зеленый
0	1	1	1	ярко - голубой
1	0	0	1	ярко - красный
1	0	1	1	ярко - сиреневый
1	1	0	1	желтый
1	1	1	1	белый (повышен. интенсивности)

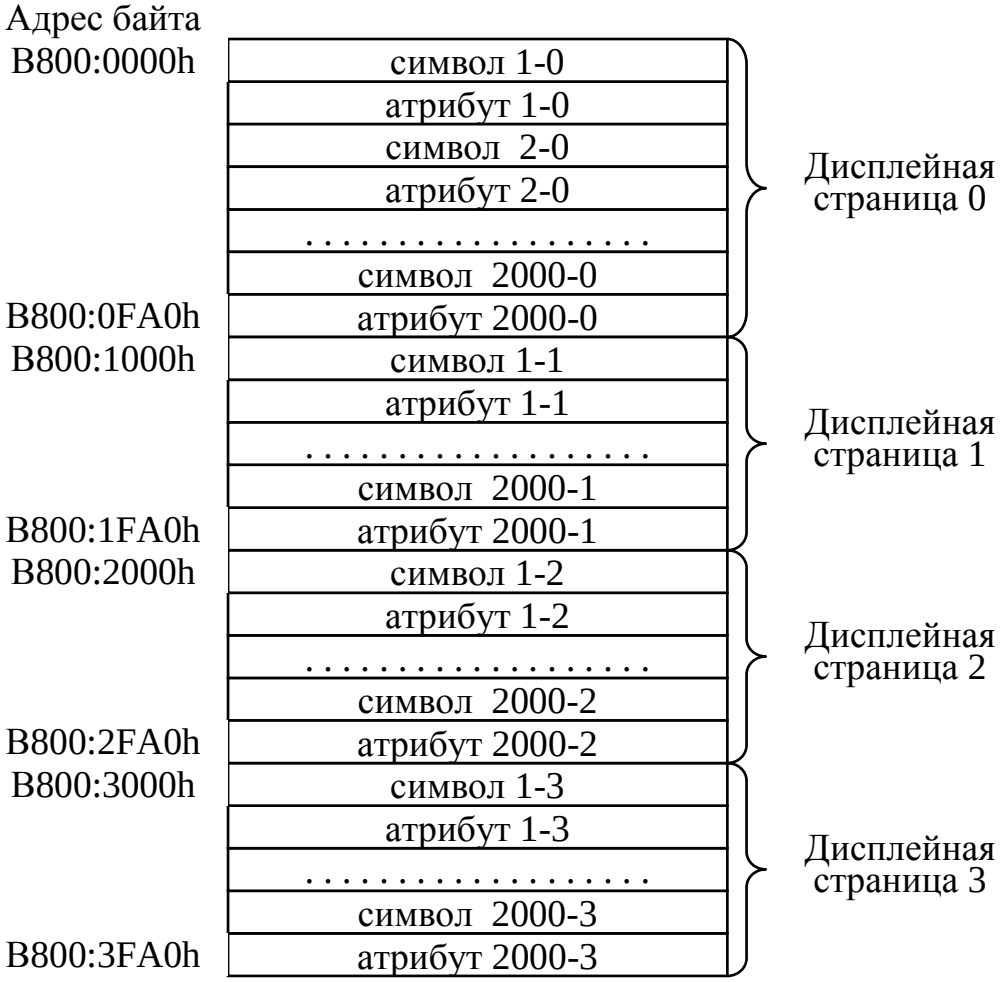


Рис.39. Содержимое видеопамяти

Аппаратура адаптера периодически считывает содержимое видеопамяти (а точнее – дисплейной страницы) и помещает его на экран. Электронный луч, управляемый системой отклонения, пробегает по экрану строка за строкой слева направо и сверху вниз (развертка). При этом контроллер включает и выключает интенсивность луча, повторяя «узор» битов в видеопамяти. За секунду электронный луч 50 раз пробегает по всему экрану (кадру). Между кадрами луч должен из правого нижнего угла вернуться в левый верхний угол. Это движение называется ***обратным ходом луча***.

Для того чтобы вывести символ с определенными атрибутами в заданную позицию экрана, необходимо выполнить последовательность действий:

1) рассчитать смещение L относительно начала видеопамяти;
2) записать по адресу B800:L код ASCII выводимого символа;
3) записать по адресу B800:L +1 байт атрибутов выводимого символа.

Для расчета L можно воспользоваться формулой:

$$L = (80*Y_T + X_T)*2,$$

где X_T ,Y_T – текущие координаты (столбец и строка) курсора.

5.5.3. Управление курсором

Под термином ***управление курсором*** понимается изменение формы курсора и его координат на экране. При этом аппаратно курсор никак не связан с выводом символов на экран. Например, если даже очистить от символов весь экран, курсор все равно останется неподвижным.

Общепринято, что курсор должен указывать на то место экрана, куда будет выведен следующий символ. Если мы используем системные вызовы MS-DOS или BIOS, то их подпрограммы сами устанавливают курсор в нужное положение. Но если мы программируем на уровне портов, то должны сами позаботиться об этом. Так как за отображение курсора на экране отвечает CRT, то для управления курсором необходимо выполнить действия по программированию этого контроллера.

Для того чтобы курсор был виден на экране, его координаты могут меняться в пределах 25 строк (0…24) и 80 столбцов (0…79), то есть в пределах экрана. При этом положение курсора содержится в регистрах 14 и 15 (см. табл.1) как число от 0 до 1999, что соответствует 2000 (25*80) позициям экрана. Если содержимое регистров 14 и 15 изменить, то положение курсора также изменится. Для этого достаточно выполнить действия:

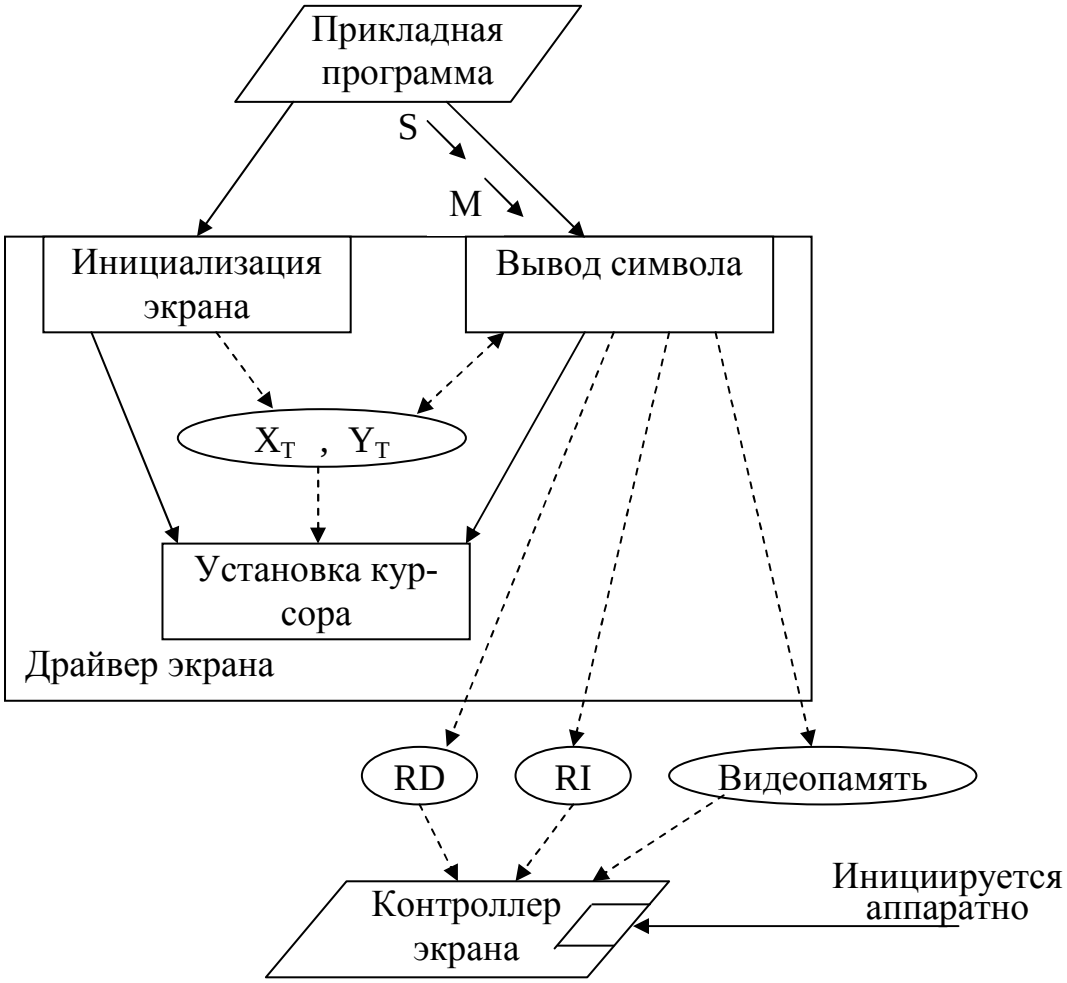
1) поместить в RI запрос на использование регистра 15;
2) поместить в RD младший байт позиции курсора;
3) поместить в RI запрос на использование регистра 14;
4) поместить в RD старший байт позиции курсора.

После выполнения этих действий курсор будет немедленно перемещен контроллером на заданную позицию экрана.

Форма курсора может меняться – от тонкой линии до максимального размера, отводимого под символ. Это обеспечивается за счет того, что курсор строится из коротких горизонтальных отрезков, верхний из которых называется начальной строкой курсора, а нижний – конечной строкой. В CGA для каждого символа (а, следовательно, и для курсора) отводятся только 8 строк, пронумерованных начиная сверху от 0 до 7. Значения начальной и конечной строк содержатся соответственно в управляющих регистрах 10 и 11. Запись в эти регистры выполняется точно также, как и изложенная выше запись в регистры 14 и 15. (Заметим, что интересный эффект получается при задании начальной строки больше конечной.)

5.5.4. Логическая схема

На рис. 40 приведена логическая схема простейшего драйвера экрана. Этот драйвер позволяет любой системной или прикладной программе вывести на экран символ с заданным цветом. Драйвер состоит из двух логических процедур, доступных извне драйвера: «Инициализация экрана», «Вывод символа». Кроме того, драйвер включает внутреннюю процедуру «Установка курсора», а также две структуры данных - переменные X_T иY_T, содержащие текущие координаты курсора. Использование этих переменных позволяет повысить скорость пересчета новых координат курсора при обработке символов “Возврат каретки” и “Перевод строки”.



S – код символа
M – цвет символа ($0 \leq M \leq 7$)
X_T ,Y_T – текущие координаты курсора

Рис.40. Логическая схема драйвера экрана

Логическая процедура “Инициализация экрана” не имеет параметров. Она выполняет первоначальную подготовку экрана (через его адаптер) к последующей работе с ним, а также выполняет первоначальную подготовку самого драйвера. Алгоритм этой логической процедуры включает шаги:

Шаг 1. Очистка используемой (нулевой) видеостраницы путем заполнения ее символами пробела с фоном требуемого цвета.

Шаг 2. Задание нулевой видеостраницы в качестве отображаемой на экране. Для этого в регистры начального адреса сканирования (см. табл.1) следует поместить 0.

Шаг 3. Задание максимальной толщины курсора.

Шаг 4. Запись нулевых значений в переменные X_T иY_T.

Шаг 5. Установка курсора в начальную позицию экрана.

Шаг 6. Объявление резидентным модуля “Вывод символа”.

В том случае, если драйвер экрана предназначен для обслуживания одной-двух прикладных программ, инициируемых достаточно редко, шаг 6 отсутствует. Так как в этом случае нецелесообразно постоянно держать драйвер в ОП. Гораздо лучше объединить его и прикладную программу в единый загрузочный модуль.

Логическая процедура “Вывод символа” имеет два входных параметра: S – код символа, M – цвет символа ($0 \leq M \leq 7$). Двоичное представление числа M определяет цвет в соответствии с таблицей 2. Например, M=2=010b задает зеленый цвет символов. Алгоритм данной логической процедуры:

Шаг 1. Если символ S есть «Возврат каретки» или «Перевод строки», то переход на шаг 3.

Шаг 2. Запись символа и его атрибутов в видеопамять.

Шаг 3. Запись новых значений в переменные X_T и Y_T .

Шаг 4. Установка следующей позиции курсора.

Внутренняя процедура «Установка курсора» не имеет параметров. Она выполняет установку курсора в ту позицию, которая соответствует значениям переменных X_T и Y_T.

Программная реализация данного драйвера выполняется студентами во время первой лабораторной работы.

6. ЛАБОРАТОРНЫЕ РАБОТЫ

Введение

В процессе выполнения данных лабораторных работ студенты должны получить навык по программированию драйверов для управления ПУ. При этом в качестве ПУ рассматриваются наиболее доступные и распространенные устройства – экран и клавиатура.

При выполнении лабораторных работ требуется доступ к MS-DOS, запускаемой в среде операционной системы Windows 95 или 98. А также требуется наличие системных программ: 1) Tasm – транслятор ассемблера; 2) Tlink – редактор связей; 3) Debug – отладчик.

Результаты лабораторных работ оформляются в виде файлов, пересылаемых в ТМЦДО на дискете. Все исходные программы (файлы с расширением asm) должны быть снабжены программными комментариями. При этом особое внимание следует уделить вводным комментариям, поясняющим назначение и интерфейсы программных модулей.

Лабораторная работа №1
ПРОГРАММИРОВАНИЕ ДРАЙВЕРА ЭКРАНА

Задание

Требуется реализовать программно нерезидентный драйвер экрана с видеоадаптером CGA, логическая схема которого приведена на рис.40. Кроме того, требуется разработать прикладную программу, выполняющую вывод на экран ваших фамилии и имени, и, возможно, другой информации (по вашему усмотрению), используя для вывода на экран ваш драйвер. Цвет символов, выводимых на экран, а также цвет фона должны быть выбраны из таблицы 3 в зависимости от номера вашего варианта.

Таблица 3

К	Цвет символов	Цвет фона
1	черный	зеленый
2	синий	коричневый
3	зеленый	красный
4	голубой	черный
5	красный	сиреневый
6	сиреневый	зеленый
7	коричневый	голубой
8	белый	черный
9	серый	ярко-красный
10	ярко-синий	ярко-зеленый

11	ярко-зеленый	ярко-белый
12	ярко-голубой	ярко-красный
13	ярко-красный	желтый
14	ярко-сиреневый	ярко-синий
15	желтый	ярко-голубой
16	ярко-белый	желтый
17	серый	ярко-красный
18	ярко-синий	серый
19	ярко-зеленый	ярко-синий
20	ярко-голубой	желтый

Результат выполнения работы оформляется в виде двух файлов с расширением asm (исходные тексты прикладной программы и драйвера) и одного файла с расширением com (загрузочный модуль прикладной программы и драйвера). Все файлы должны быть помещены в каталог LAB1.

Рекомендуемый план отладки

Рекомендуется выполнять отладку драйвера экрана не целиком, а постепенно наращивая его функции в соответствии со следующим планом.

Шаг 1. Прикладная программа выполняет вызов процедуры инициализации, за которым следуют ожидание нажатия клавиши и возврат в MS-DOS. Процедура инициализации выполняет только очистку экрана заданным цветом фона.

Шаг 2. Отличается от шага 1 только тем, что процедура инициализации выполняет не только очистку экрана, но и первоначальную установку курсора.

Шаг 3. Прикладная программа выполняет вызов процедуры инициализации, за которым следуют ожидание нажатия клавиши. Далее она в цикле вызывает процедуру вывода символа, передавая ей каждый раз на вход код следующего символа из символьной строки, заданной в прикладной программе. Данная строка завершается каким-то особым байтом, например, 24h. После завершения вывода прикладная программа ожидает нажатия клавиши и выполняет возврат в MS-DOS.

Шаг 4. Отличается от шага 3 тем, что выводимая на экран символьная строка содержит кроме других символов «возврат каретки» и «перевод строки».

Примечание 1. При выполнении отладки обычно требуется уметь остановить выполнение программы в заданной точке. В данной работе при выполнении прикладной программы подобная остановка используется дважды. Во-первых, после завершения процедуры инициализации, так как требуется время на перемещение курсора. Во-вторых, программа приостанавливается перед возвратом в MS-DOS. В противном случае сразу же по-

сле возвращения управления из прикладной программы в MS-DOS, последняя заменит на экране выходные данные программы своими данными.

Для выполнения остановки программы можно воспользоваться системным вызовом BIOS «ожидание ввода с клавиатуры” – INT 16h (функция 0), поместив в программу строки:

```
MOV AH, 0          ; Номер функции
INT  16h           ; Вызов подпрограммы BIOS
```

Примечание 2. Обе программы не должны содержать системных вызовов MS-DOS и BIOS, выполняющих вывод на экран.

Примечание 3. При отладке подпрограммы, выполняющей работу с курсором, надо помнить, что подпрограммы MS-DOS и BIOS, выполняющие вывод на экран, будут игнорировать вашу установку курсора и вернут его в то положение, которое он занимал до начала выполнения вашей программы (соответствующее 2-х байтовое значение хранится в области данных BIOS). Например, если вы пользуетесь отладчиком, то во время работы курсор перестает “слушаться” вашу программу и не двигается с места.

Примечание 4. Так как драйвер экрана и прикладная программа находятся в разных исходных файлах, то имена программных процедур, выполняющих реализацию логических процедур «Инициализация экрана» и «Вывод символа»должны быть перечислены в операторах PUBLIC и EXTRN, причем оператор EXTRN с атрибутом NEAR.

Примечание 5. Если программная процедура выполняет запись в регистр DS, то в начале этой процедуры обязательно требуется сохранить, а в конце ее восстановить прежнее содержимое этого регистра. Иначе в вызывающей программе будет нарушена адресация данных.

Лабораторная работа №2
ПРОГРАММИРОВАНИЕ ДРАЙВЕРА КЛАВИАТУРЫ

Задание

Требуется разработать программу нерезидентного драйвера клавиатуры, выполняющего запросы прикладных программ по вводу символов с клавиатуры. Драйвер должен обрабатывать управляющую клавишу <CapsLock>, выполняющую переключение регистров. Перед возвратом в MS-DOS выполняется восстановление системной обработки прерываний от клавиатуры.

Для проверки работоспособности драйвера клавиатуры используется прикладная программа, которая, используя драйвер клавиатуры, выполняет ввод символьной строки с клавиатуры в свой внутренний (прикладной) буфер. Получив код символа $ (24h), прикладная программа выводит со-

держимое своего буфера на экран, используя для этого драйвер экрана, полученный в результате первой лабораторной работы.

При правильной работе программы введенная с клавиатуры строка должна отобразиться на экране дважды: один раз при наборе на клавиатуре (эхо символов), а второй раз – как результат вывода на экран содержимого прикладного буфера.

Результат выполнения работы оформляется в виде двух файлов с расширением asm (исходные тексты прикладной программы и драйвера клавиатуры) и одного файла с расширением com (загрузочный модуль, включающий прикладную программу, а также драйверы экрана и клавиатуры). Все файлы должны быть помещены в каталог LAB2.

Логическая схема ввода с клавиатуры

На рис.41 приведена логическая схема, обеспечивающая посимвольный обмен с клавиатурой. В реализации этого обмена участвуют контроллер клавиатуры, а также разработанный в предыдущей работе драйвер экрана.

Для того чтобы выполнить ввод с клавиатуры, любая системная или прикладная программа вызывает интерфейсную логическую процедуру драйвера клавиатуры «Ввод символа». Если в буфере драйвера имеется хотя бы один символ (N>0), то этот символ в качестве выходного параметра передается в прикладную программу. При отсутствии символа в прикладную программу возвращается соответствующее значение признака результата r. Исходной причиной появления кода символа в буфере драйвера является нажатие соответствующей клавиши на клавиатуре. Взаимосвязь между этими двумя событиями обеспечивают контроллер клавиатуры и обработчик прерываний. Как показано на рис.41, логическая структура контроллера клавиатуры представляет собой совокупность двух процессов – «Прием» и «Выдача». Процесс «Прием» инициируется при нажатии, а также при отпускании любой клавиши и помещает генерируемый клавишей код (SCAN-код) во внутренний 4-позиционный буфер контроллера. В результате появления кода в буфере SCAN-кодов инициируется процесс «Выдача», находившийся в состоянии «Вход 1». Данный процесс выбирает код из буфера, помещает его в порт 60h и выдает в ЦП сигнал прерывания.

Если прерывание разрешено, инициируется модуль «Обработчик прерываний клавиатуры», который считывает SCAN-код из порта 60h и сообщает контроллеру о завершении этой операции установкой бита 7 в порте 61h. Это приводит к инициированию процесса «Выдача», находившегося в состоянии «Вход 2». Данный процесс возвращается в состояние «Вход 1». Далее опять выбирается SCAN-код из буфера, и описанные выше действия повторяются.

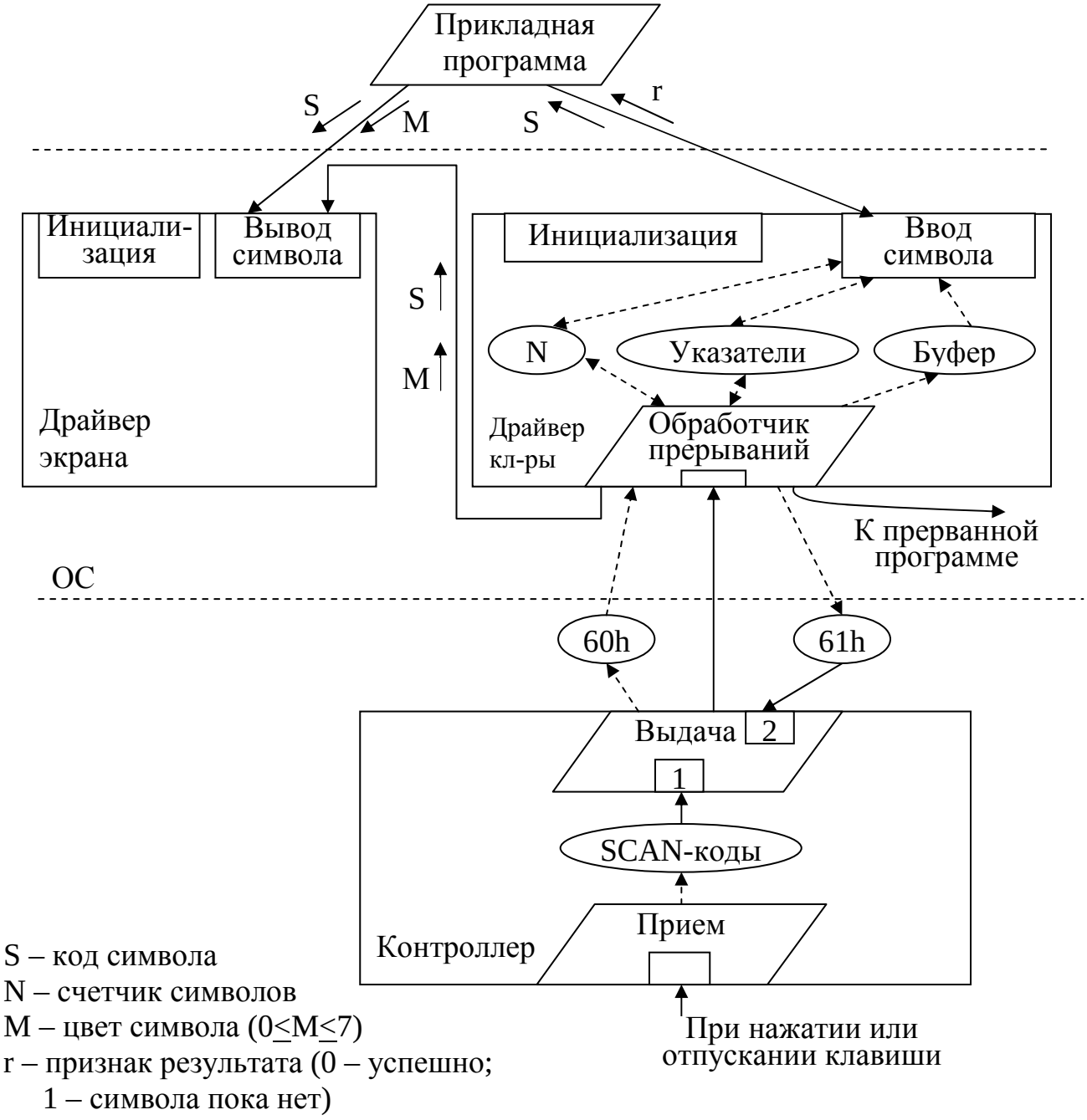


Рис.41. Логическая схема ввода с клавиатуры

После установки бита 7 в 61h обработчик прерываний выполняет обработку принятого SCAN-кода. Обработка кода, вызванного нажатием алфавитно-цифровой клавиши, сводится к преобразованию SCAN-кода в код ASCII, размещению полученного кода в буфер драйвера и к выводу эха символа.

Необходимость вывода «эха» обусловлена тем, что клавиатура и экран не связаны между собой на уровне контроллеров. Для того чтобы видеть на экране вводимый символ, необходимо выполнить связывание на уровне драйверов. Как видно из рис.41, для этого достаточно, чтобы при

нажатии отображаемого символа обработчик прерываний клавиатуры вызвал логическую процедуру «Вывод символа».

Логическая процедура «Инициализация» имеет единственный входной параметр u: u=0 - выполнить замену системного обработчика прерываний клавиатуры на свой обработчик; u=1 – выполнить замену своего обработчика прерываний клавиатуры на системный обработчик.

SCAN - коды

SCAN-код – однобайтовое число, младшие 7 бит которого представляют идентификационный код клавиши. Старший бит кода (бит 7) указывает на то, была ли клавиша нажата (0) или освобождена (1). Например, 7-битный SCAN-код клавиши «v» есть 47 (или 0101111b). Когда эту клавишу нажимают, в порт 60h контроллер записывает код 00101111b, а когда отпускают – 10101111b.

Значения SCAN-кодов приведены в таблице 4. В этой таблице приведены SCAN-коды только тех клавиш, которым соответствуют отображаемые на экране символы. Что касается управляющих клавиш, то приведен код только для клавиши <CapsLock>, которая обрабатывается нашим драйвером.

Таблица 4

SCAN- коды клавиш

Наименование клавиши		SCAN-код	Наименование клавиши		SCAN-код
1	!	0000010	a	A	0011110
2	@	0000011	s	S	0011111
3	#	0000100	d	D	0100000
4	$	0000101	f	F	0100001
5	%	0000110	g	G	0100010
6	^	0000111	h	H	0100011
7	&	0001000	j	J	0100100
8	*	0001001	k	K	0100101
9	(	0001010	l	L	0100110
0	)	0001011	;	:	0100111
-	_	0001100	'	“	0101000
=	+	0001101	\	|	0101011
q	Q	0010000	z	Z	0101100
w	W	0010001	x	X	0101101
e	E	0010010	c	C	0101110
r	R	0010011	v	V	0101111
t	T	0010100	b	B	0110000

y	Y	0010101	n	N	0110001
u	U	0010110	m	M	0110010
i	I	0010111	,	<	0110011
o	O	0011000	.	>	0110100
p	P	0011001	/	?	0110101
[	{	0011010	Пробел		0111001
]	}	0011011	Caps Lock		0111010
Enter		0011100			

Алгоритмы программных модулей

Алгоритмы программных модулей драйвера клавиатуры имеют много общего с алгоритмами программных модулей драйвера считывателя перфоленты (см. п.5.3.3). В частности, буфер драйвера клавиатуры представляет собой несвязанную циклическую очередь, алгоритмы работы с которой очень похожи на соответствующие алгоритмы драйвера ввода с перфоленты. Но между этими драйверами есть и существенные отличия, вызванные прежде всего техническими деталями управления конкретным устройством (клавиатурой).

Алгоритм программной реализации логической процедуры «Инициализация» состоит из следующих шагов:

Шаг 1. Если u=1 переход на шаг 4 .

Шаг 2. Сохранение в ОП прежнего содержимого вектора прерываний с номером 09h.

Шаг 3. Запись стартового адреса обработчика прерываний клавиатуры в вектор прерываний с номером 09h. Переход на шаг 5.

Шаг 4. Восстановление прежнего содержимого вектора прерываний с номером 09h.

Шаг 5. Возврат из процедуры.

Алгоритм обработчика прерываний клавиатуры включает шаги:

Шаг 1. Разрешение маскируемых прерываний.

Шаг 2. Сохранение в программном стеке содержимого регистров.

Шаг 3. Запись в сегментный регистр DS значения, которое соответствует адресу-сегменту данных обработчика прерываний.

Шаг 4. Чтение из порта 60h значения SCAN-кода.

Шаг 5. Установка бита 7 порта 61h.

Шаг 6. Сброс бита 7 порта 61h.

Шаг 7. Если была отпущена клавиша <CapsLock>, то инвертирование флага статуса клавиатуры. Переход на шаг 14.

Шаг 8. Если было отпускание клавиши, то переход на шаг 14.

Шаг 9. Если буфер полон, то переход на шаг 14.

Шаг 10. Перекодировка SCAN-кода в символ ASCII.

Шаг 11. Запись символа ASCII в буфер драйвера.

Шаг 12. Вывод «эха» полученного символа на экран.

Шаг 13. Если полученный символ является символом «возврат каретки» (этот символ есть результат нажатия <Enter>), то запись в буфер и вывод на экран дополнительного символа «перевод строки».

Шаг 14. Выдача в программируемый контроллер прерываний команды для разрешения менее приоритетных прерываний.

Шаг 15. Восстановление содержимого регистров из стека.

Шаг 16. Возврат из прерывания в прерванную программу.

Дополнение к шагам 5 и 6. Порт 61h используется не только клавиатурой (она использует только бит 7), но и другими устройствами. Поэтому к моменту завершения шага 6 содержимое этого порта должно быть тем же, что было до начала шага 5.

Дополнение к шагам 7 и 10. Большинству клавиш соответствует не один, а два символа. Например, одна и та же клавиша соответствует символам «1» и «!». Каждой большой (прописной) букве соответствует малая (строчная). Для того чтобы обработчик прерываний выполнял правильное преобразование SCAN-кода в код ASCII, необходимо ввести понятие “состояние клавиатуры”. В одном состоянии в буфер драйвера записывается код ASCII большой буквы, а во втором – малой. Часто состояния клавиатуры называют «нижним регистром» и «верхним регистром».

Для отслеживания состояния клавиатуры обработчик прерываний должен иметь двоичную переменную «флаг статуса клавиатуры». Для управления этим флагом рекомендуется использовать управляющую клавишу <CapsLock>. Нажатие, а затем отпускание <CapsLock> обеспечивает долговременное изменение флага статуса, которое действует до следующего нажатия этой же клавиши.

При программировании шага 10 удобно свести значения SCAN-кодов и соответствующих кодов ASCII в единую таблицу. Первая строка этой таблицы может иметь вид:

```
Tabl    DB    02h, ‘1’, ‘!’
```

Строка таблицы, соответствующая пробелу:

```
        DB    39h, ‘ ‘, ‘ ‘
```

В данной таблице отсутствует строка, соответствующая управляющему символу CapsLock. В качестве последнего байта таблицы рекомендуется взять нулевой байт.

Шаг11 рекомендуется реализовать в виде программной процедуры.

Алгоритм подпрограммы «Ввод символа» состоит из следующих шагов:

Шаг 1. Сохранение в программном стеке содержимого регистров.

Шаг 2. Если буфер пуст, то возврат в программу с соответствующим значением признака результата.

Шаг 3. Переписка символа из буфера в регистр, используемый для передачи символа из подпрограммы в вызывающую программу.

Шаг 4. Увеличение указателя «взять» и уменьшение счетчика символов.

Шаг 5. Если указатель «взять» вышел за границу буфера, то установка его на начало буфера.

Шаг 6. Восстановление регистров из стека.

Шаг 7. Возврат из подпрограммы с соответствующим значением признака результата.

Рекомендуемый план отладки

Отладку программы драйвера клавиатуры рекомендуется выполнить в два этапа.

Этап 1. Обеспечение вывода на экран содержимого прикладного буфера, а также обеспечение вывода на экран «эха» символов, вводимых с клавиатуры. В прикладной буфер с помощью псевдооператора DB записывается какая-то символьная строка, заканчивающаяся каким то особым символом, например, с кодом 24h. Прикладная программа сначала вызывает процедуру инициализации экрана, а затем процедуру инициализации клавиатуры с параметром u=0. Затем она выводит посимвольно содержимое прикладного буфера на экран с помощью соответствующей процедуры драйвера экрана. При достижении конечного символа (с кодом 24h) прикладная программа в цикле вызывает процедуру драйвера клавиатуры «Ввод символа» до тех пор, пока от нее не будет получен символ с определенным кодом, в качестве которого опять можно использовать 24h. После этого прикладная программа вызывает процедуру инициализации клавиатуры с параметром u=1 и делает возврат в MS-DOS. Обработчик прерываний клавиатуры должен правильно реагировать на нажатие каждой клавиши, помещая (если нужно) ее код в буфер драйвера клавиатуры и выводя эхо символа на экран (с помощью процедуры драйвера экрана).

Этап 2. Окончательная отладка драйвера клавиатуры. Для этого изменяется прикладная программа, использовавшаяся на шаге 1: после завершения инициализации экрана и клавиатуры прикладная программа выполняет ввод с клавиатуры (с помощью процедуры «Ввод символа») строки символов в свой прикладной буфер. Ввод в прикладной буфер производится до тех пор, пока прикладная программа не получит символ с кодом 24h. Далее повторяются действия этапа 1: прикладная программа выводит содержимое своего прикладного буфера на экран, а затем в цикле вызывает процедуру «Ввод символа». При получении кода 24h восстанавливается системный обработчик прерываний клавиатуры и делается возврат в MS-DOS.

Примечание. Разработанные программы не должны содержать системных вызовов MS-DOS и BIOS, выполняющих информационный обмен с клавиатурой и экраном.

КОНТРОЛЬНАЯ РАБОТА № 2

РАЗРАБОТКА ПРОГРАММЫ НА АССЕМБЛЕРЕ

Целью выполнения данной работы является проверка навыков программирования на ассемблере задач, описание которых приведено в настоящем пособии.

Результаты контрольной работы представляются в виде файлов на дискете:

1) файл (файлы) с расширением asm – исходный модуль (модули) разработанной программы (программ);

2) файл (файлы) с расширением com – загрузочный модуль (модули) разработанной программы (программ).

Общие требования к программам:

– обязательное наличие комментариев в исходных модулях - заголовков к программным модулям и построчных комментариев;

– текст каждого исходного модуля должен быть представлен в коде ASCII. (Такой текст может быть получен, например, с помощью текстового редактора, встроенного в Norton commander);

– запрещается применение команд для процессоров, отличных от i8086;

– обязательное наличие выходных сообщений программы, предваряющих ввод пользователя с клавиатуры;

– обязательное наличие выходных сообщений в случае ошибок при выполнении системных вызовов.

Разработка некоторых программ предполагает использование для ввода и вывода не системных вызовов MS-DOS и BIOS, а собственных драйверов экрана и клавиатуры, полученных ранее при выполнении лабораторных работ. Наличие такого требования обязательно отмечается в соответствующем задании на программирование. В этом случае результаты контрольной работы должны включать, кроме перечисленных выше файлов, загрузочные модули используемых драйверов (копии файлов, представленных в результатах лабораторных работ).

Кроме того, некоторые программы должны быть резидентными. Это требование также обязательно отмечается в задании.

Варианты контрольной работы №2

Вариант 1. Резидентная программа инициируется нажатием клавиши <F1> и выводит на экран ваши имя и фамилию, записанные английскими буквами. При нажатии клавиши <F2> программа уничтожается.

Примечание. Для вывода на экран следует использовать свой драйвер экрана.

Вариант 2. Резидентная программа инициируется нажатием клавиши <F2> и выводит на экран ваши имя и фамилию, записанные русскими буквами. При нажатии клавиши <F1> программа уничтожается.

Примечание. Для вывода на экран следует использовать свой драйвер экрана.

Вариант 3. Разработать простейший отладчик программ, который получает имя загрузочного модуля прикладной программы в качестве своего параметра, и выполняет эту программу покомандно, выдавая после завершения каждой ее команды на экран содержимое регистров AX и BX в шестнадцатеричной системе счисления. (Некоторый аналог команды T Debug.)

Примечание 1. Работа программы основана на запуске трассируемой программы и обработке исключения «Трассировка» (см. замечание в конце п.2.6.2).

Примечание 2. Для вывода на экран шестнадцатеричного содержимого регистров можно использовать программную процедуру, алгоритм которой рассматривается в [2].

Вариант 4. Разработать простейший отладчик программ, который вводит с клавиатуры имя загрузочного модуля прикладной программы, и выполняет эту программу покомандно, выдавая после завершения каждой ее команды на экран содержимое регистров CX и DX в двоичной системе счисления. (Некоторый аналог команды T Debug.)

Примечание 1. Работа программы основана на запуске трассируемой программы и обработке исключения «Трассировка» (см. замечание в конце п.2.6.2).

Примечание 2. Для вывода на экран двоичного содержимого регистров можно использовать программную процедуру, алгоритм которой рассматривается в [2].

Вариант 5. Разработать простейший интерпретатор команд, выполняющий обработку командных файлов (bat-файлов). Имя командного файла вводится с клавиатуры. Имя запускаемой программы (строка bat-файла) не имеет параметров. После завершения запуска очередной программы на экран выводится сообщение об успешности запуска.

Вариант 6. Разработать простейший интерпретатор команд, выполняющий обработку командных файлов (bat-файлов). Имя командного файла интерпретатор команд получает при своем запуске (в качестве параметра команды). Имя запускаемой программы (строка bat-файла) не имеет параметров. После завершения запуска очередной программы на экран выводится сообщение об успешности запуска.

Вариант 7. Прикладная программа выполняет запуск другой (дочерней) прикладной программы, получив предварительно ее имя с клавиатуры. При своем запуске дочерняя программа получает на входе (в PSP) строку символов, которую она выводит на экран.

Примечание. Для вывода на экран дочерняя программа должна использовать ваш драйвер экрана.

Вариант 8. Прикладная программа выполняет запуск другой (дочерней) прикладной программы, получив ее имя в качестве своего параметра (хвоста команды). При своем запуске дочерняя программа получает на входе (в PSP) строку символов, которую она выводит на экран.

Примечание. Для вывода на экран дочерняя программа должна использовать ваш драйвер экрана.

Вариант 9. Прикладная программа выводит на экран содержимое своего блока окружения.

Примечание. Для вывода на экран программа должна использовать ваш драйвер экрана.

Вариант 10. Прикладная программа выполняет уничтожение файла. Имя уничтожаемого файла вводится с клавиатуры.

Примечание. Для ввода с клавиатуры и для вывода на экран следует использовать ваши драйверы.

Вариант 11. Прикладная программа вводит с клавиатуры имя нового текстового файла, записывает в него содержимое своего блока окружения, а также «хвоста», а затем выводит содержимое этого файла на экран.

Вариант 12. Прикладная программа вводит с клавиатуры имя существующего текстового файла, «дописывает» в него содержимое своего блока окружения, а также «хвоста», а затем выводит содержимое этого файла на экран.

Вариант 13. Резидентная программа инициируется из прикладной программы, выполняя вывод на экран блока окружения и «хвоста» прикладной программы.

Примечание. Для того чтобы обрабатывать данные вызывающей программы, содержимое регистров сегментов данных должно соответствовать прикладной, а не резидентной программе.

Вариант 14. Прикладная программа вводит с клавиатуры имена двух существующих текстовых файлов, а затем инициирует резидентную программу (передав ей на вход через PSP имена файлов), которая «расширяет» первый файл, добавив в него содержимое второго файла.

Вариант 15. Прикладная программа вводит с клавиатуры имя существующего текстового файла, а затем выводит его содержимое на экран.

Примечание. Для вывода на экран следует использовать свой драйвер экрана.

Вариант 16. Прикладная программа получает в качестве параметра команды имя существующего текстового файла, а затем выводит его содержимое на экран.

Примечание. Для вывода на экран следует использовать свой драйвер экрана.

Вариант 17. Прикладная программа вводит с клавиатуры имя текстового файла, содержащего имена других текстовых файлов. А затем выводит на экран содержимое этих текстовых файлов.

Вариант 18. Прикладная программа получает в качестве параметра команды имя текстового файла, содержащего имена других текстовых файлов. А затем выводит на экран содержимое этих текстовых файлов.

Вариант 19. Прикладная программа выполняет копирование файла. Имена исходного файла и файла-копии вводятся с клавиатуры.

Примечание. Для вывода на экран следует использовать свой драйвер экрана.

Вариант 20. Прикладная программа выполняет копирование файла. Имя исходного файла вводится с клавиатуры, а имя файла-копии программа получает в качестве параметра команды.

Примечание. Для вывода на экран следует использовать свой драйвер экрана.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

Настоящее учебно-методическое пособие предполагает наличие предварительных знаний по программированию на языке ассемблера для процессора i8086. Для изучения этого языка рекомендуются пособия [1, 2], но могут использоваться и другие доступные источники.

Книга [3] содержит достаточно полные сведения об аппаратуре рассматриваемой однопрограммной ВС. В [4] содержится достаточно полный перечень системных вызовов MS-DOS и BIOS, а также рассматриваются вопросы программного использования этих вызовов.

Книги [5, 6] содержат подробное описание не только традиционных, но и достаточно новых системных вызовов, присущих последним версиям MS-DOS. Кроме того, эти источники могут использоваться как учебники по языку ассемблера.

1. Одиноков В.В. Информатика. Ассемблер для процессора i8086. Учебное пособие. -Томск, ТМЦДО, 2000. – 93 с.
2. Одиноков В.В. Информатика. Ассемблер для процессора i8086. Учебное методическое пособие. -Томск, ТМЦДО, 2000. – 100 с.
3. Лю Ю., Гибсон Г. Микропроцессоры семейства 8086/8088. – М., ”Радио и связь”, 1987. – 512 с.
4. Данкан Р. Профессиональная работа в MS-DOS. – М., «Мир», 1993. – 510 с.
5. Зубков С.В. Ассемблер для DOS, Windows и Unix. – М., ДМК, 1999. – 640 с.
6. Пирогов В.Ю. Assembler. Учебный курс. – М., «Нолидж», 2001. – 846 с.

ПРИЛОЖЕНИЕ

ГРАФИЧЕСКИЙ ЯЗЫК ПРЕДСТАВЛЕНИЯ ЛОГИЧЕСКИХ СТРУКТУР

Логическая структура информационной системы

Логическая структура информационной системы описывает состав модулей переработки информации, образующих эту систему, а также информационные и управляющие взаимосвязи между ними. Спроектировав логическую структуру системы, далее можно производить проектирование каждого модуля отдельно. При этом модуль может рассматриваться как система, для которой может быть получена своя логическая структура. Для успешного выполнения подобного проектирования очень важно обеспечить выполнение ***принципа модульности*** – представление системы в виде совокупности относительно независимых частей (модулей).

Системы переработки информации делятся на аппаратные, программные и программно-аппаратные. Программная и программно-аппаратная системы отличны только с точки зрения разработчика, так как реальная ВС всегда программно-аппаратная. Отличие состоит в том, что при разработке программной системы аппаратные средства считаются заданными, а не являются объектом разработки. Построение логической структуры необходимо при разработке системы любого из перечисленных трех типов. Причем при получении логической структуры программно-аппаратной системы разработчик может еще не знать окончательно, какой модуль будет реализован программно (и на каком процессоре), а какой – аппаратно.

Для того чтобы логическая структура могла быть использована в процессе проектирования системы, она должна быть представлена графически, то есть в виде рисунка, называемого далее ***логической схемой***. Частным случаем логической схемы является блок-схема. Ограниченность блок-схем состоит в том, что они применимы только для представления последовательных логических структур, называемых обычно последовательными алгоритмами. В такой структуре никакие два этапа не могут выполняться параллельно, то есть одновременно во времени. С учетом того, что реальные логические структуры часто бывают параллельными, для представления их логических схем должны использоваться другие графические языки. Примером графического языка, пригодного для представления параллельных логических структур, является язык, рассматриваемый далее.

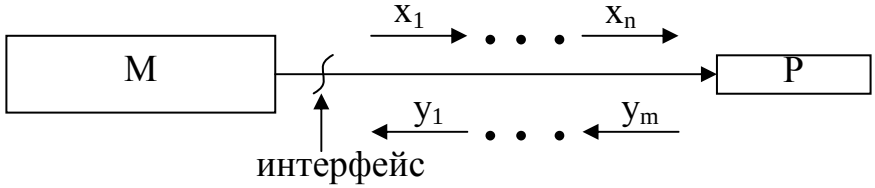
Данный графический язык включает модули пяти типов: 1) процедуры; 2) процессы; 3) структуры данных; 4) пассивные пакеты; 5) активные пакеты. Отличие модуля «структура данных» от остальных типов модулей

состоит в том, что с ним не связан никакой алгоритм (алгоритмы) и поэтому данный модуль не может быть инициирован, то есть не может быть «запущен в работу». Модули первых трех типов, а именно, процедуры, процессы и структуры данных являются элементарными. При этом модуль считается ***элементарным***, если он не может быть детализирован с помощью данного графического языка, то есть не может быть записана его логическая структура. Пассивные и активные пакеты не являются элементарными модулями и могут быть представлены в ходе проектирования своими логическими структурами.

Процедуры

Процедура - модуль, который имеется во всех универсальных языках программирования. Ее графическое изображение представляет собой небольшой прямоугольник (рис. 42).

Допустим, что модуль М «вызывает» процедуру Р. Это означает, что М выдает управляющее воздействие, которое инициирует Р, то есть “запускает в работу” ее алгоритм. Одновременно М передает процедуре Р ее входные параметры, которые представляют собой исходные данные для ее алгоритма. После того как алгоритм процедуры завершится, она возвращает управление в ту точку алгоритма модуля М, из которой она была инициирована. Передаваемые при этом от Р к М выходные параметры процедуры содержат выходные данные ее алгоритма.



Р – процедура;
М – модуль, инициирующий Р;
⟶ - управляющее воздействие;
x_1 ,…, x_n – входные параметры Р;
y_1 ,…, y_m – выходные параметры Р.

Рис.42. Графическое представление процедуры

Важным свойством процедуры является то, что ее алгоритм строго последователен. Это означает, что в любой момент времени может выполняться только один оператор алгоритма. Последовательный алгоритм процедуры может быть описан с помощью языка блок-схем или с помощью другого языка, используемого для представления последовательных программ.

Если два модуля (например, М и Р на рис.42) взаимодействуют друг с другом, то между ними можно провести границу, называемую ***интер-***

фейсом. Точное описание интерфейса называется ***спецификацией интерфейса***. Спецификация интерфейса для процедуры представляет собой описание ее параметров и способа их передачи.

Одну и ту же процедуру могут вызывать два и более модулей (рис.43). Обычная процедура в принципе не может вызываться одновременно несколькими модулями. Только после того, как она выполнится в интересах одного модуля, она может быть инициирована другим. Если одновременные вызовы возможны, то логическая схема ошибочна. Исключением из этого правила являются ***реентерабельные процедуры***.

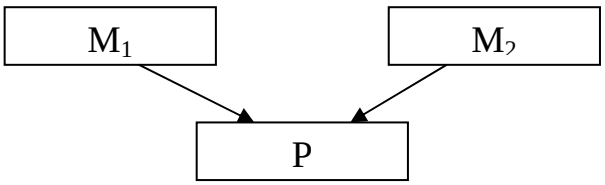
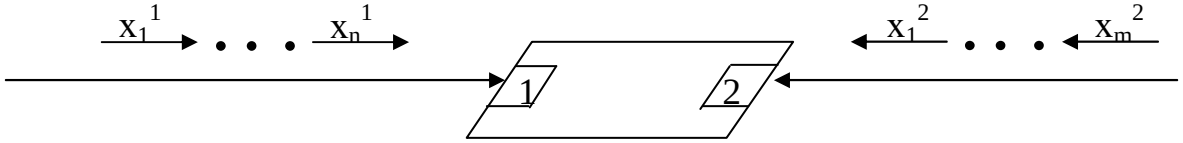


Рис. 43. Вызов процедуры несколькими модулями

Область ОП, занимаемая реентерабельной процедурой, не содержит изменяемых данных. Для хранения таких данных выделяются несколько областей памяти (по числу вызывающих модулей). При вызове реентерабельной процедуры один из ее входных параметров однозначно указывает на используемую область данных. Таким образом вызывающие модули могут одновременно вызывать процедуру, не мешая друг другу.

Процессы

Процесс изображается графически в виде параллелограмма (рис. 44). Как и процедура, процесс имеет последовательный алгоритм работы – никакие два этапа не могут выполняться одновременно. Поэтому внутренняя структура процесса представляет собой совокупность взаимосвязанных процедур. На логической схеме эта структура не изображается. Но при желании она может быть наглядно представлена с помощью любого графического языка, используемого для представления последовательных алгоритмов, например, в виде блок-схемы или в виде графа переходов.



⟶ - управляющее воздействие;
1, 2 – входы;
$x_1^1,\ldots, x_n^1, x_1^2,\ldots, x_m^2$ – параметры управляющих воздействий.

Рис.44. Графическое представление процесса

Точкой входа процесса или просто ***входом*** называется такая точка алгоритма процесса, в которой он может воспринимать внешние управляющие воздействия. Находясь в точке входа, алгоритм процесса «простаивает». Поступившее управляющее воздействие инициирует его. Далее выполнение алгоритма продолжается до тех пор, пока он не достигнет какой-либо точки входа и не перейдет таким образом опять в состояние «простаивания».

Одновременно с инициированием алгоритма процесса ему передаются значения входных параметров, которые соответствуют данному входу. Эти параметры уточняют характер входного управляющего воздействия и используются алгоритмом процесса в качестве своих входных данных.

Все сказанное выше о процессе не содержит сколько-нибудь существенных различий между ним и процедурой. Подобное различие состоит в том, как осуществляется возврат управления. Допустим, что процесс A может вызывать как процесс B, так и процедуру C (рис.45). Взаимодействие A с модулями B и C совершенно различно. После вызова C выполнение алгоритма A приостанавливается в той точке, откуда был сделан вызов. Выполнившись, процедура всегда возвращает управление туда, откуда она была вызвана. То есть никакой параллельности в выполнении A и C нет.

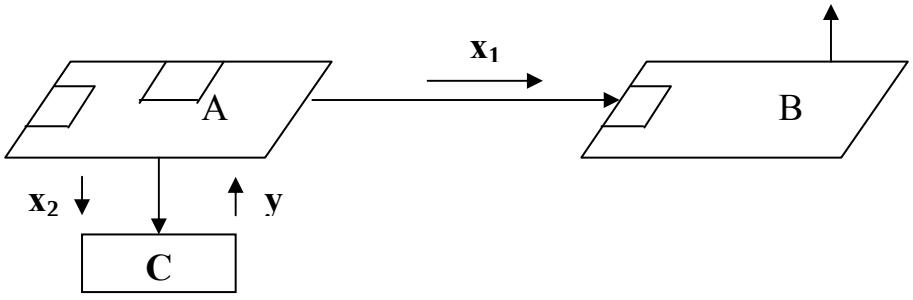


Рис.45. Инициирование процессом процедуры и процесса

После инициирования B модуль A сразу же оказывается логически отсоединенным от B. (Обычно алгоритм A оказывается в одной из точек своих входов и, следовательно, он может реагировать на внешние управляющие воздействия.) С другой стороны, процесс B после инициирования модулем A выполняется до тех пор, пока он не окажется в одной из своих входных точек. Таким образом, процессы A и B могут выполняться параллельно во времени. (Этим объясняется использование для изображения процесса параллелограмма.)

Логическая параллельность двух модулей, которая следует из логической структуры системы, является необходимым, но не достаточным условием их ***физической параллельности***. Физическая параллельность отсутствует, если оба логически параллельных модуля реализованы в виде

программ, выполняемых на одном и том же процессоре. На этапе логического проектирования неважно, будет ли на последующих этапах логическая параллельность отображена в физическую параллельность или нет.

На этапе логического проектирования также не существенны способы реализации управляющих воздействий и передачи входных параметров. Они определяются на последующих этапах проектирования и зависят от того, как реализованы взаимодействующие модули. Если модули А и В (см. рис. 45) реализованы программно и для их выполнения используется единственный процессор, то управляющее воздействие можно реализовать только в мультипрограммной системе (с помощью вызова EXEC). В однопрограммной системе один программный процесс не может запустить другой. Если А и В выполняются на разных процессорах, или если А – аппаратный процесс, то управляющее воздействие реализуется в виде сигнала прерывания.

Структуры данных

Структура данных - модуль, не имеющий алгоритма выполнения и используемый для связи между собой модулей других типов. Примеры структур данных: переменная, массив, список, файл. На логической схеме структуру данных будем изображать в виде овала.

На рис. 46 процесс А записывает данные в Y, а процесс В считывает их. Подобная структура данных, в которую один процесс («Писатель») записывает, а второй («Читатель») считывает информацию, называется ***буфером***.

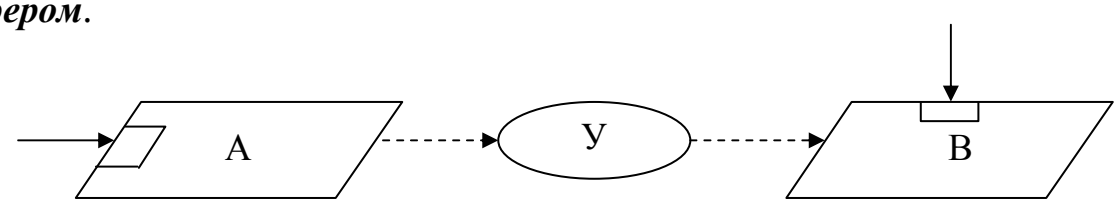


Рис.46. Взаимодействие двух процессов через буфер

Между модулями допустимо и смешанное взаимодействие – и по управлению и через общие структуры данных. На рис.47 процесс А заполняет Y, а затем вызывает В, чтобы В обработал содержимое Y.

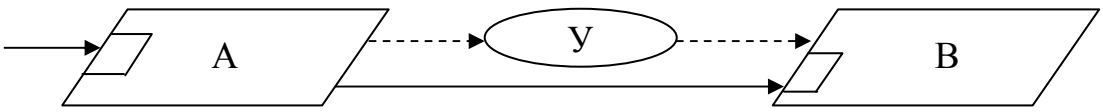


Рис.47. Взаимодействие двух процессов по управлению и через буфер

Иногда управляющее воздействие реализуется через структуру данных. На рис. 48 процесс А записывает в структуру данных Y такое содержимое, которое инициирует В. (Не следует путать такое взаимодействие с взаимодействием через буфер.)

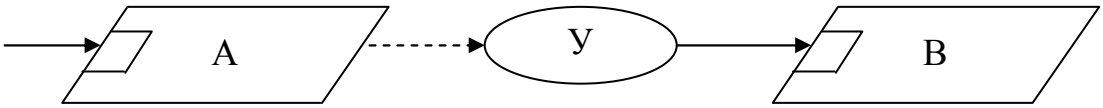


Рис.48. Реализация управляющего воздействия через структуру данных

Пакеты

Пакет – модуль, внутренняя структура которого описывается с помощью данного графического языка. На логической схеме пакет изображается в виде прямоугольника, имеющего больший размер по сравнению с изображением процедуры. Пакеты разделяются на пассивные и активные.

Пассивный пакет – модуль, содержащий процедуры и структуры данных. Часто такой модуль формируется с целью свести операции обработки какой-то структуры данных вместе, в рамках одного модуля. На рис.49 изображен пассивный пакет, предназначенный для обработки структуры данных Y. Процедуры «Прочитать», «Записать» и «Скорректировать» – интерфейсные, так как они доступны извне пакета. В пакете могут находиться и внутренние процедуры, доступ к которым извне пакета невозможен.

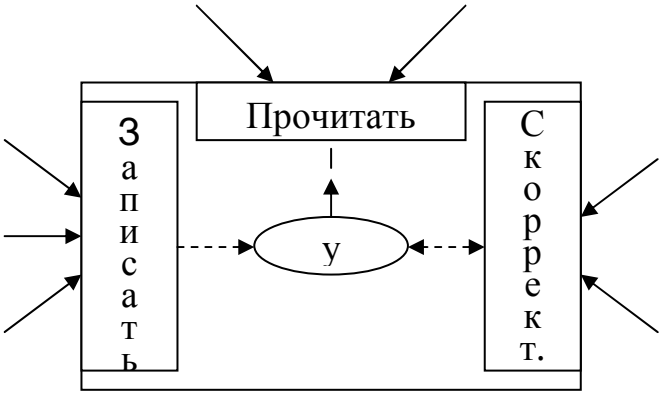


Рис.49. Пример пассивного пакета

Активный пакет – модуль, содержащий, кроме процедур и структур данных, хотя бы один процесс. Пример активного процесса приведен на рис.50. Здесь процедура «Записать» позволяет внешним модулям помещать данные в буфер Y, а процесс A считывает эти данные из Y и обрабатывает.

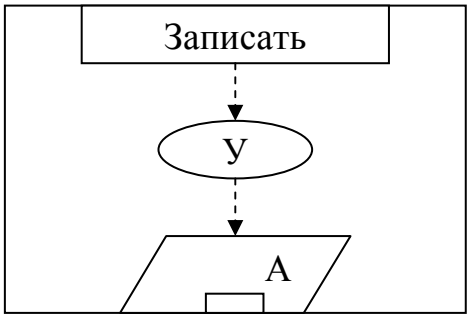


Рис.50. Пример активного пакета