МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Алтайский государственный технический университет
им. И.И. Ползунова»

Л.И. Сучкова

ПРОГРАММНО-АППАРАТНЫЕ АСПЕКТЫ НИЗКОУРОВНЕВОГО ОБМЕНА С ПЕРИФЕРИЙНЫМИ УСТРОЙСТВАМИ

Учебное пособие

Барнаул 2012

УДК 004.42

Сучкова Л.И. Программно-аппаратные аспекты низкоуровневого обмена с периферийными устройствами: Учебное пособие. / Алт. гос. техн. ун-т им. И.И. Ползунова. – Барнаул, 2012. – 187 с., ил.

Данное учебное пособие предназначено для изучения логики функционирования периферийных устройств компьютера и управления ими через регистры и порты средствами языка Ассемблер. Пособие рекомендуется для студентов направлений «Информатика и вычислительная техника» и «Программная инженерия». Подробность изложения и большое количество примеров позволяют использовать пособие для дистанционной формы обучения.

Рекомендовано заседаниями
кафедры ПМ (протокол № 9 от 8.06.12 г.) и
кафедры ВСИБ (протокол № 11 от 30.05.12 г.)

Рецензент: доктор технических наук, профессор, зав. кафедрой вычислительных систем и информационной безопасности А.Г. Якунин

ВВЕДЕНИЕ

Это учебное пособие посвящено одному из самых старых из существующих сегодня языков программирования - Ассемблеру. Когда-то он был основным языком, без знания которого нельзя было работать на компьютере. Однако при появлении более удобных для программирования языков высокого уровня Ассемблер не исчез, да и не мог исчезнуть, так как он является символическим представлением машинного языка, обрабатывающего данные на самом низком, аппаратном уровне. Как известно, программы на Ассемблере отличаются минимальным объемом и высоким быстродействием. Без хорошего знания Ассемблера и логики функционирования периферийных устройств в вычислительной среде персонального компьютера невозможно грамотно решать проблемы, связанные с программированием нестандартных устройств. Не случайно языки высокого уровня поддерживают ассемблерные вставки и средства связи с модулями на Ассемблере. В настоящее время широко практикуется составление программ, в которых основная часть написана на языке высокого уровня, а наиболее критические по времени участки написаны на Ассемблере.

Так как в основе серьезного компьютерного образования лежит знание принципов работы компьютера, его архитектуры, то язык Ассемблер предоставляет уникальную возможность изучения компьютера на уровне аппаратуры, освоения возможностей контроллеров устройств и операционной системы. Эти вопросы в настоящее время представляют значительный практический интерес для разработчиков автоматизированных систем управления технологическими процессами.

В данном учебном пособии рассматриваются вопросы программирования на языке Ассемблер для компьютеров на базе процессоров Intel, а именно организация ввода-вывода информации и хранения данных, а также алгоритмы функционирования контроллеров основных периферийных устройств компьютера с программно-аппаратной точки зрения.

1 АРХИТЕКТУРА ПРОЦЕССОРОВ INTEL

При низкоуровневом программировании необходимо учитывать такие особенности архитектуры процессора, как правила использования регистров, организацию и способы адресации памяти, организацию и разрядность интерфейсов ЭВМ. В связи с этим первая глава настоящего учебного пособия посвящена рассмотрению программно-аппаратных особенностей процессоров фирмы Intel.

1.1 РЕГИСТРЫ ПРОЦЕССОРА

Регистр представляет собой цифровую электронную схему, служащую для временного хранения двоичных чисел. В процессоре Intel имеется значительное количество регистров, большая часть которых используется самим процессором и недоступна программисту. Например, при выборке из памяти очередной команды она помещается в регистр команд. Программист обратиться к этому регистру не может. Имеются также регистры, которые в принципе программно доступны, но обращение к ним осуществляется из программ операционной системы (например, управляющие регистры и теневые регистры дескрипторов сегментов, которые будут рассмотрены ниже). Этими регистрами пользуются в основном разработчики операционных систем.

Рассмотрим сначала регистры, общие для 32 и 64-битной архитектуры. Для использования в прикладных программах программная модель процессоров Intel имеет 16 регистров прикладного программиста или пользовательских регистра.

Регистры прикладного программиста включают в себя 8 регистров общего назначения, 6 16-битных сегментных регистров, регистр флагов (32-битный ***EFLAGS*** или 64-битный ***RFLAGS***) и регистр указателя команд. О разрядности регистров косвенно можно судить по начальной букве его наименования – ***r*** означает разрядность 64, ***e*** –разрядность32.

У регистров общего назначения доступными для самостоятельной адресации являются их младшие части. Так, для 64-разрядных регистров ***rax***, ***rbx***, ***rcx***, ***rdx*** доступны для самостоятельного обращения их младшие 32-битные части, называемые соответственно ***eax***, ***ebx***, ***ecx***, ***edx***. 32-битные части, в свою очередь, также предоставляют для работы свои половинки – младшие разряды. Например, биты с 0 по 15 регистра ***eax*** представляют собой регистр ***ax***, который тоже делится на старшую и младшую части (соответственно 8-битовые регистры ***ah*** и ***al***). Отметим, что

старшие 16 битов 32-разрядных регистров и старшие 32 бита 64-разрядных регистра как самостоятельные объекты недоступны.

У регистров общего назначения имеются свои особенности в использовании. Регистр ***rax (eax , ax***) называется регистром-аккумулятором, регистр ***rdx (edx,dx***) - регистром данных, они оба служат для хранения промежуточных данных вычислений, в некоторых командах их использование происходит неявно. Например, при выполнении команды **div *bx*** делимое задается неявно и должно быть расположено в паре регистров ***dx,ax*** причем в ***ax*** должна находиться младшая часть делимого. После выполнения команды частное помещается в ***ax***, а остаток о деления - в ***dx***. Регистр ***rcx*** выполняет функцию счетчика и применяется в командах, осуществляющих повторяющиеся действия, например, ***rcx*** используется в команде цикла ***LOOP***. Регистр ***rbx*** традиционно применяется для хранения базового адреса некоторого объекта памяти.

Регистры ***rsi*** и ***rdi*** используются для поддержки операций последовательной обработки цепочек элементов, причем регистр ***rsi*** содержит текущий адрес элемента в исходной цепочке-источнике, а регистр ***rdi*** содержит текущий адрес в цепочке-приемнике. Для этих регистров программно доступны младшие 32-битные части – ***edi*** и ***esi,*** а у ***edi*** и ***esi*** – младшие 16-битные части – ***di*** и ***si***.

Для работы со стеком в программной модели процессора существуют регистры ***rsp (esp,sp)*** и ***rbp (ebp,rbp)***. Регистр ***rsp*** содержит указатель вершины стека в сегменте стека, а регистр ***rbp*** предназначен для организации произвольного доступа к данным внутри стека.

Отмеченные выше функциональные особенности регистров не являются жесткими ограничениями на их использование, большинство регистров могут использоваться для хранения операндов в различных сочетаниях. Примеры применения рассмотренных регистров в программах будут рассмотрены во второй главе учебного пособия.

Только в 64-битном режиме работы процессора доступны регистры ***R8-R15*** и их части (***R8D-R15D, R8W-R15W, R8B-R15B***), а также младшие части регистров ***sp, bp, si, di*** (соответственно ***spl, bpl, sil, dil***).

Таким образом, архитектура x86-64 вводит две новые особенности:

1. расширение количества и разрядности регистров :

- 8 регистров общего назначения ***R8-R15*** (general-purpose registers);
- все 16 регистров общего назначения 64-битные;
- 8 дополнительных 128-битных XMM регистров к тем 8, что были в 32-разрядной архитектуре;

- новый командный префикс (REX) для доступа к расширенным регистрам.

2. специальный режим "Long Mode", предусматривающий:

- до 64-бит виртуальных адресов;
- 64-битные указатель команд (RIP);
- плоское (flat) адресное пространство.

Отметим, что восемь 64-битных MMX-регистров и восемь 80-битных регистров математического сопроцессора для вычислений с вещественными числами присутствуют и в 32-х и в 64-х битной архитектурах программной модели процессора.

Кроме того, для организации доступа к памяти процессор имеет 6 16-битовых сегментных регистров – ***ds, cs, es, ss, fs*** и ***gs***. Их роль в формировании физического адреса памяти будет рассмотрена ниже.

Различают следующие группы системных и управляющих регистров:

■ 4 регистра управления - ***cr0,cr1,cr2,cr3***. Они доступны только программам с уровнем привилегий 0. Регистр ***cr0*** содержит системные флаги (разрешение защищенного режима, наличие сопроцессора, флажок переключения задач, разрешение/запрещение кэш-памяти, разрешение/запрещение страничного преобразования). Регистр ***cr2*** служит для организации подкачки в память страниц, отсутствующих в памяти в данный момент. Регистр ***cr3*** содержит адрес каталога страниц текущей задачи. Страничная организация памяти будет рассмотрена в п. 1.2 пособия.

■ 4 регистра системных адресов - ***gdtr, ldtr, idtr, tr.*** Более подробно назначение указанных регистров будет рассмотрено ниже.

■ 8 регистров отладки (***dr0-dr7***).

В программной модели процессора имеются регистры, которые содержат информацию о состоянии как самого процессора, так и программы, выполняющейся в данный момент. Это регистр флагов ***rflags (eflags, flags***) и регистр указателя команды ***rip*** (***eip,ip***).

Рассмотрим структуру регистра флагов ***eflags:***

31	...	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
0	...	0	AC	VM	RF		NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

CF (Carry Flag) - флажок переноса, 1 в этом бите указывает на осуществление переноса из старшего бита результата в арифметических операциях;

PF (Parity Flag) - флажок четности, 1 в бите показывает, что 8 младших разрядов результата операции содержат четное число единиц;

AF - флажок вспомогательного переноса в десятичной арифметике, работающей с числами в формате BCD;

ZF (Zero Flag) - флажок нуля (равен 1, если результат операции равен 0);

SF (Sign Flag)- флажок знака результата, отражает состояние старшего бита результата, равен 1, если этот бит установлен;

TF (Trace Flag) - флажок трассировки, предназначен для организации пошаговой работы микропроцессора, если флажок равен 1, то микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Этот флажок может использоваться отладчиками;

IF (Interrupt Flag) - флажок прерывания. Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний по входу INTR процессора. Если IF равен 1, то прерывания разрешены, иначе - прерывания запрещены. Немаскируемые и внутренние прерывания распознаются всегда. Для установки IF в 1 служит команда STI, для сброса IF в 0 служит команда CLI.

DF (Direction Flag) - флажок направления обработки цепочек. Если DF равен 0, то строки обрабатываются от начала к концу, если DF=1, то направление обработки меняется на противоположное. Для работы с флагом DF служат команды CLD и STD, устанавливающие соответственно значения флага 0 и 1;

OF (Overflow Flag)- флажок переполнения. Используется для фиксирования факта потери значащего бита при арифметических операциях;

IOPL (Input-Output Privilege Level) (2 бита) - действует в защищенном режиме и показывает минимальный уровень привилегий, при котором разрешается выполнение ввода-вывода;

NT (Nested Task) - флажок вложенной задачи, действует только в защищенном режиме, нужен для переключения задач в многозадачном режиме;

RF (Restore Flag) - флажок возобновления, используется в средствах отладки;

VM (Virtual Mode) - флажок виртуального режима 8086. Если VM=0, то процессор находится в R- или P- режимах; если VM=1, то процессор работает в V-режиме;

AC (Alignment Check) - флажок контроля выравнивания данных. Так, Pentium позволяет размещать данные с любого байта, но если требуется выравнивание данных по границе слова, то обращение к нечетному адресу приводит к возникновению исключительной ситуации.

1.2 ВИРТУАЛЬНАЯ И ФИЗИЧЕСКАЯ ПАМЯТЬ

Процессор может выполнять программу только в том случае, если она находится в памяти. Обращения к переменным в программе, переходы по меткам основаны на адресации ячеек памяти. Различают виртуальное и физическое адресные пространства.

Микропроцессор имеет доступ по шине адреса к физической оперативной памяти, при этом диапазон значений физических адресов зависит от разрядности шины адреса. Данные и команды хранятся в адресованных ячейках памяти. Минимально адресуемая порция информации - 1 байт, состоящий из 8 битов. Каждому байту соответствует свой уникальный физический адрес.

Процессор 8086 имел 16-ти разрядную шину данных и 20-ти разрядную шину адреса, что определяло физическое адресное пространство в 2^{20} = 1Мб. В процессоре 80286 - 24-х разрядная шина адреса и 16-ти разрядная шина данных, поэтому физическое адресное пространство составляет 2^{24} = 16Мб. В Pentium 32-х разрядная шина данных и 32-х разрядная шина адреса, поэтому теоретический диапазон значений физических адресов находится в пределах от 0 до 2^{32}-1 (4 Гб), хотя на практике объем физической памяти может быть меньше. Механизм управления памятью полностью аппаратный, а значит, программист не может вмешаться в процесс формирования физического адреса на адресной шине.

Диапазон адресов виртуального адресного пространства у процессоров с одной разрядностью является одним и тем же. Например, если процессор 32-разрядный, то диапазон адресов составляет 0-FFFFFFFF, что дает 4 Гб виртуальной памяти. Виртуальное адресное пространство дает логическое представление о наборе адресов, не обязательно соответствующее структуре физической памяти. Если объем физической памяти меньше объема виртуального адресного пространства, то необходимо разбиение виртуального адресного пространства на части и аппаратная реализация сохранения ненужных в данный момент частей на диске и подкачки в физическую память нужных частей, которые там отсутствуют. В связи с этим необходимо установление взаимосвязи между физическим и виртуальным адресом.

Преобразование виртуальных адресов в физические зависит от того, какой способ структуризации виртуального адресного пространства принят в подсистеме управления памяти конкретной операционной системы. В настоящее время реализация виртуальной памяти представлена тремя классами:

1) Страничная виртуальная память.
2) сегментная виртуальная память
3) сегментно-страничная виртуальная память.

Использование этих классов зависит от режима работы микропроцессора. Если используется устаревший (legacy) режим, то обеспечивается бинарную совместимость не только с 16- и 32-битными приложениями, но и с 16- и 32-битными операционными системами. Legacy режим включает в себя три режима:

- Защищенный (protected) режим. Поддерживаются 16- и 32-битные программы с сегментной организацией памяти, поддержкой привилегий и виртуальной памяти. Адресное пространство - 4Гб.
- Виртуальный (virtual-8086) режим. Поддерживает 16-битные приложения, запускаемые как задачи в защищенном режиме. Адресное пространство - 1Мб.
- Реальный (real) режим. Поддерживает 16-битные программы с простой регистровой адресацией сегментированной памяти. Не поддерживается виртуальная память или привилегии. Доступно 1Мб памяти.

Устаревший (legacy) режим используется только при работе 16- и 32-битных операционных систем.

Архитектура x86-64 поддерживает весь набор инструкций x86 и добавляет некоторые новые инструкции для поддержки long-режима. Команды разбиты на несколько подмножеств:

- Команды общего назначения. Это основные x86 целочисленные команды, используемые во всех программах. Большинство из них предназначены для загрузки, сохранения, обработки данных, расположенных в регистрах общего назначения или памяти. Некоторые из этих инструкций управляют потоком команд, обеспечивая переход к другому месту в программе.
- 128-битные медиа-команды. Это SSE и SSE2 (streaming SIMD extension) команды, предназначенные для загрузки, сохранения, или обработки данных, расположенных в 128-битных XMM регистрах. Они выполняют целочисленные или с плавающей точкой операции над векторными (упакованными) и скалярными типами данных. Поскольку векторные инструкции могут независимо выполнять одну операцию над множеством данных, они называются single-instruction, multiple-data (SIMD) командами. Они используются для медиа- и научных приложений для обработки блоков данных.
- 64-битные медиа-команды. Это multimedia extension (MMX) и 3DNow! команды. Они сохраняют, восстанавливают и обрабатывают данные, расположенные в 64-битных MMX регистрах. Подобно их 128-битным аналогам, описанным выше, они выполняют целочисленные и с плавающей точкой операции надо векторными (упакованными) и скалярными данными.

- x87 команды. Предназначены для работы с плавающей точкой в старых x87 приложениях. Обрабатывают данные в x87 регистрах.

Некоторые из этих команд соединяют два или более подмножества описанных выше команд. Например, это команды пересылки данных между регистрами общего назначения и XMM или MMX регистрами.

Рассмотрим различные концепции использования виртуального адресного пространства более подробно.

1.2.1 Страничная организация памяти

При реализации страничной виртуальной памяти виртуальное адресное пространство делится на части одинакового фиксированного размера, называемые виртуальными страницами. Если виртуальное адресное пространство какого-то процесса не кратно размеру страницы, то последняя страница дополняется фиктивной областью. Физическая оперативная память также делится на части такого же размера, называемые физическими страницами. Размер страницы выбирается кратным степени двойки. Копия всех виртуальных страниц хранится на диске. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. Для каждого процесса ОС создает таблицу страниц, в которой хранятся дескрипторы всех виртуальных страниц. Каждый дескриптор содержит:

- номер физической страницы, в которую загружена данная виртуальная страница;
- признак присутствия, устанавливается в 1, если виртуальная страница находится в оперативной памяти;
- признак модификации страницы, устанавливается в 1, когда по адресу, относящемуся к странице, осуществляется запись;
- признак обращения к странице (бит доступа), устанавливается в 1 при каждом обращении по адресу в странице.

Сама таблица страниц также располагается в памяти. Если не учитывать кэширование, то при каждом обращении к памяти в таблице выполняется поиск номера виртуальной страницы, содержащей требуемый адрес, затем анализируется признак присутствия страницы в памяти, и, если страница в памяти, то виртуальный адрес преобразуется в физический. Иначе проверяется, имеется ли свободная физическая страница в оперативной памяти. Если да, то требующаяся виртуальная страница загружается с диска немедленно, иначе анализируется, какую физическую страницу можно выгрузить на диск. Перед выгрузкой содержимого физической страницы на диск анализируется бит

модификации связанной с ней виртуальной страницы. Если он равен 1, то выталкиваемая из физической памяти страница была модифицирована, и ее необходимо сохранить на диске.

Рассмотрим преобразование виртуального адреса в физический. Виртуальный адрес при страничном распределении может быть представлен в виде пары, включающей порядковый номер виртуальной страницы и смещение относительно начала виртуальной страницы. Если размер страницы выбирается кратным $2^{к}$, то $к$ младших разрядов в записи адреса страницы представляют собой смещение, а оставшаяся часть адреса – это номер страницы. (рисунок 1.1). При этом в таблице виртуальных страниц в качестве номера физической страницы хранятся старшие разряды адреса физической страницы, а номер дескриптора виртуальной страницы равен старшим разрядам ее виртуального адреса. В силу одинаковости размера виртуальной и физической страниц в пределах страницы непрерывная последовательность виртуальных адресов однозначно отображается в непрерывную последовательность физических адресов. Это означает, что смещение для виртуального адреса равно смещению в физическом адресе. В связи с этим при преобразовании виртуального адреса в физический по старшим разрядам адреса виртуальной страницы из таблицы виртуальных страниц выбирается дескриптор, в котором хранятся старшие разряды адреса физической страницы, а младшие разряды адреса берутся из смещения в виртуальной странице в силу их одинаковости.

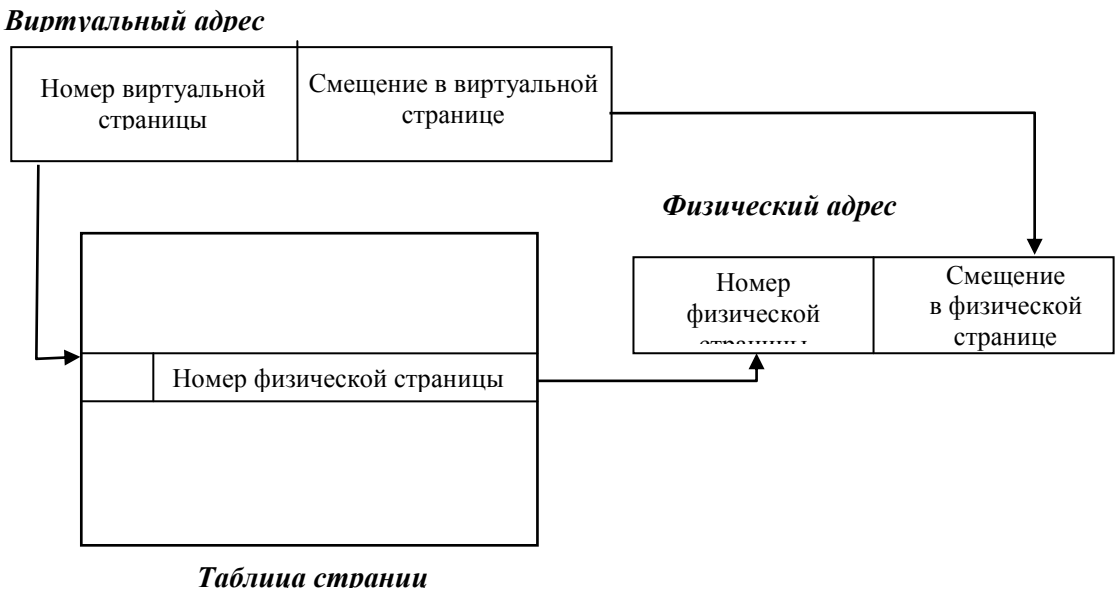


Рисунок 1.1 – Схема преобразования виртуального адреса в физический при страничной организации памяти

Выбор оптимального размера страницы является достаточно сложной задачей, так как при маленьком размере страницы сами таблицы страниц занимают достаточно много места в памяти, а поскольку они хранятся в таких же страницах, то их обработкой также надо

управлять, что усложняет алгоритм управления загрузкой и выгрузкой страниц. Поэтому при страничной организации виртуальной памяти хранят в памяти не всю таблицу страниц, а только ее часть.

Этот метод реализован при двухуровневой структуризации виртуального адресного пространства, когда виртуальное пространство делится на разделы, а разделы, в свою очередь, делятся на страницы. Для каждого раздела строится своя таблица страниц. Количество дескрипторов в таблице выбирается таким образом, чтобы они занимали одну страницу. Каждая таблица страниц описывается дескриптором, структура которого совпадает со структурой дескриптора страницы. Все дескрипторы таблиц сведены в таблицу разделов, которая называется каталогом страниц. Страница, содержащая таблицу разделов, постоянно находится в памяти. Виртуальный адрес при двухуровневой структуризации виртуального адресного пространства состоит из трех групп битов: младшие биты хранят смещение, средние – номер виртуальной страницы в разделе, а старшие – номер раздела. По номеру раздела из таблицы разделов выбирается дескриптор нужной таблицы страниц. Если этой таблицы нет в памяти, она подгружается с диска. В этой таблице по номеру виртуальной страницы выбирается ее дескриптор, из которого извлекается номер физической страницы.

1.2.2 Сегментное распределение памяти

Страничное распределение памяти предусматривает только механическое разбиение на страницы, оно не позволяет различным образом организовать работу со страницами, содержащими команды и данные. При сегментном распределении памяти виртуальное адресное пространство делится на части, называемые сегментами, размер которых определяется смысловым значением содержащейся в них информации. Максимальный размер сегмента определяется разрядностью процессора, для 32-разрядного процессора он равен 4 Гбайт. Сегменты никак не упорядочиваются друг относительно друга. Виртуальный адрес задается парой чисел: номером сегмента и смещением внутри сегмента. При загрузке процесса в оперативную память помещается только часть его сегментов, полный образ виртуального адресного пространства находится на диске. Для каждого загружаемого сегмента подыскивается непрерывный участок свободной физической памяти достаточного размера. Для работы с сегментом имеется дескриптор сегмента, содержащий:

- базовый физический адрес сегмента в оперативной памяти;
- размер сегмента;
- уровень привилегий сегмента (DPL);

- признаки модификации и обращения к сегменту;

- признак присутствия сегмента в памяти;

- способ использования сегмента (чтение, запись, выполнение) и его тип (данные, код, системный).

Дескрипторы сегментов располагаются в специальных таблицах. В процессоре Pentium имеется два типа таблиц для дескрипторов сегментов:

- глобальная таблица дескрипторов (***GDT***);

- локальная таблица дескрипторов (***LDT***).

Глобальная таблица в системе одна, а локальных столько, сколько задач. Сами таблицы дескрипторов тоже хранятся в сегментах, и поэтому в глобальной таблице наряду с дескрипторами сегментов операционной системы хранятся дескрипторы локальных таблиц. Базовый адрес в физической памяти глобальной таблицы и ее размер содержатся в регистре ***gdtr***. Для работы с локальной таблицей дескрипторов сегментов, активной в текущий момент, используется регистр ***ldtr***. Он содержит номер дескриптора в глобальной таблице, соответствующий нужной локальной таблице.

Обращение к физической памяти осуществляется по адресу, описываемому содержимым одного из сегментных регистров и смещением. В сегментном регистре хранится номер дескриптора (селектор) в таблице ***GDT*** или ***LDT***, признак, из какой таблицы осуществляется выборка дескриптора, и требуемый уровень привилегий – RPL. Смещение представляет собой положение нужной ячейки памяти относительно начала сегмента.

При преобразовании в физический адрес виртуального адреса, заданного через номер сегмента и смещение, осуществляется сложение базового физического адреса сегмента, выбранного из нужной таблицы сегментов по его дескриптору, и смещения в сегменте.

Недостатки сегментного распределения:

1. Операцией конкатенации в данном случае пользоваться нельзя, так как сегмент может располагаться в памяти, начиная с любого адреса, у которого в разрядах, соответствующих смещению, записаны не нули, как это было при страничной организации памяти.

2. Осуществляется оперирование целыми сегментами значительной длины, хотя не все они необходимы в памяти в текущий момент.

3. Возникновение фрагментации памяти, обусловленной различной длиной сегментов. В физической памяти образуются неиспользуемые куски, которые не пригодны ни для одного сегмента.

Сегментная модель организации виртуальной памяти для процессоров Pentium может использоваться как в комбинации со страничной организацией, так и без нее.

1.2.3 Сегментно-страничное распределение памяти

При сегментно-страничном подходе к организации памяти все сегменты и физическая память делятся на страницы одинакового размера. При этом все сегменты образуют непрерывное линейное виртуальное адресное пространство, страницы которого имеют сквозную нумерацию. Виртуальный адрес задается содержимым сегментного регистра и смещением.

Рассмотрим, как формируется физический адрес по виртуальному. Сначала по данным из сегментного регистра выбирается дескриптор сегмента из локальной или глобальной таблицы, затем базовый адрес сегмента из дескриптора складывается со смещением. Однако в механизме сегментно-страничного распределения памяти базовый адрес сегмента представляет собой не физический базовый адрес сегмента в оперативной памяти, а его базовый адрес в виртуальном адресном пространстве. Таким образом, в результате сложения базового адреса со смещением вычисляется линейный виртуальный адрес. Этот адрес является начальным для механизма страничного распределения. Так как адрес является 32-разрядным, а страница 4 Кбайт, то при двухуровневой схеме адресации страниц старшие 10 разрядов хранят номер раздела, средние 10 битов – номер страницы в разделе, оставшиеся младшие 12 битов хранят смещение в странице. Размер раздела в Pentium выбран так, чтобы он занимал одну физическую страницу. Виртуальная страница, в которой хранится таблица разделов, всегда находится в физической памяти, и номер физической страницы с таблицей разделов указан в системном регистре ***cr3***.

1.3 АРХИТЕКТУРНЫЕ ОСОБЕННОСТИ РЕАЛЬНОГО, ЗАЩИЩЕННОГО РЕЖИМОВ И LONG-РЕЖИМА

1.3.1 Реальный режим

В реальном режиме под сегментом понимается последовательность байтов длиной от 0 до 64 Кб. Иначе говоря, сегмент - это логическая структура, накладываемая на любые участки физического адресного пространства и позволяющая указывать в командах только часть физического адреса - смещение относительно адреса начала сегмента.

Для работы с сегментами используются 16-ти битовые сегментные регистры:

cs - для сегмента кода программы, содержащего команды;

ds - для сегмента данных, содержащего обрабатываемые программой данные;

ss - для сегмента, содержащего стек - специальную область памяти, выделяемую для временного хранения вспомогательных данных. Запись и чтение данных в стеке осуществляется в соответствии с принципом LIFO (Last In First Out - последний пришел, первый обслуживается). По мере записи в стек данных он растет в сторону младших адресов;

es,fs,gs - для дополнительных сегментов данных. Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адресуемом через ***ds***. Если одного сегмента данных недостаточно, то используются дополнительные сегменты данных, причем при работе с этими данными требуется явное указание сегментного регистра.

В реальном режиме для обращения к физическому адресу оперативной памяти необходимо определить адрес начала сегмента (базу сегмента) и смещение внутри сегмента. При этом учитывается, что в сегментном регистре содержатся только старшие 16 битов физического адреса начала сегмента. Поэтому для вычисления 20-разрядного физического адреса содержимое сегментного регистра сдвигается влево на 4 бита и складывается с 16-битным смещением. Величина смещения может содержаться явно в команде, либо косвенно в одном из регистров. Эти операции реализованы на аппаратном уровне. В реальном режиме максимальный размер сегмента 64 Кб, т.к. максимальное число, которое может содержать 16-разрядный регистр, равно 2^{16}-1.

Запись в сегментные регистры непосредственно не разрешается, поэтому необходимо использовать в качестве промежуточных регистры общего назначения или воспользоваться как промежуточным хранилищем информации стеком.

Отметим следующие недостатки реального режима:

- максимальный размер сегмента - 64 Кб;
- возможно перекрытие сегментов;
- отсутствуют средства контроля правильности использования сегментов;
- программа может обратиться по любому физическому адресу, нет средств для защиты определенных областей памяти от несанкционированного доступа.

1.3.2 Защищенный режим

Защищенный режим свободен от указанных выше недостатков реального режима. В защищенном режиме (Р-режиме) работают многозадачные операционные системы, поэтому механизм распределения памяти построен с учетом защиты различных задач от взаимного влияния и регулирования взаимосвязей между ними. Для этого предложено рассмотреть сегмент памяти как объект, описываемый 8-байтовым дескриптором сегмента

В дескрипторе 20 битов отведено для хранения размера сегмента, 32 бита - для хранения базового адреса сегмента, 1 байт - для определения прав доступа к сегменту. Из оставшихся битов в дескрипторе представляет интерес бит гранулярности G. Если он равен 0, то считается, что число, хранящееся в поле **<размер сегмента>**, соответствует длине сегмента в байтах, а если G равен 1, то единицей измерения длины сегмента является страница размером 4 Кб. Таким образом, размер сегмента в P-режиме может достигать 4 Гб, т.е. занимать все возможное физическое пространство памяти.

Для работы с областью памяти необходимо описать эту память дескриптором и занести его в одну из дескрипторных таблиц - глобальную (***GDT***), локальную (***LDT***) таблицы дескрипторов сегментов. Третья таблица дескрипторов содержит дескрипторы прерываний и будет рассмотрена позже.

Структура сегментных регистров в защищенном режиме определяется тремя полями. Биты 0 и 1 представляют собой запрашиваемый уровень привилегий (RPL, Request Privilege Level), служащий для ограничения доступа к сегменту по привилегиям. Бит 2 указывает, в какой из таблиц дескрипторов (***LDT*** или ***GDT***) искать дескриптор сегмента (если бит равен 0, то дескриптор сегмента находится в ***GDT***, иначе - в ***LDT***). Третье поле в содержимом сегментного регистра называется селектором и является индексом в соответствующей таблице дескрипторов.

К глобальной дескрипторной таблице допускается обращение программ с достаточным уровнем привилегий. ***GDT*** может содержать следующие типы дескрипторов:

- дескрипторы сегментов кодов программ;
- дескрипторы сегментов данных программ;
- дескрипторы сегментов стека программ;
- дескрипторы сегментов состояния задач (***TSS***, Task State Segment);
- дескрипторы для таблиц ***LDT***;
- специальные дескрипторы, называемые шлюзами вызова и шлюзами задач.

В локальной дескрипторной таблице (***LDT***) определены сегменты, которые могут быть использованы текущей конкретной задачей. В ***LDT*** не могут содержаться дескрипторы сегментов состояния задач и дескрипторы других ***LDT***. Дескриптор, содержащий адрес ***LDT*** текущей задачи, хранится в специальном теневом регистре.

Задачей по терминологии процессоров Intel называется программа (или даже ее фрагмент), которая выполняется в системе и обладает закрепленным за ней специальным сегментом состояния задачи ***TSS***. ***TSS*** представляет собой сегмент небольшого размера, в состав которого входит так называемый контекст задачи, который представляет собой содержимое всех регистров процессора на момент переключения задачи. Для того чтобы

задачу сделать текущей и передать ей управление, селектор ее ***TSS*** загружается в специально предназначенный для этого регистр процессора ***tr***. Переключение задач осуществляется процессором на аппаратном уровне. В ходе этого переключения процессор восстанавливает из ***TSS*** контекст задачи и передает ей управление. Поскольку в контекст задачи входит селектор ее локальной таблицы дескрипторов, то переключение задач автоматически приводит к замене активной ***LDT***, которая осуществляется загрузкой селектора ***LDT*** в регистр ***ldtr***. Тем самым реализуется защита задач друг от друга, так как процессору в любой момент доступны только те сегменты, которые описаны в активной в настоящий момент времени ***LDT***, то есть сегменты текущей задачи.

Сегменты неравноправны по правам доступа, каждый сегмент может обладать уровнем привилегий - 0,1,2 или 3. В дескрипторе сегмента имеется специальное поле уровня привилегий - ***DPL*** (Descriptor Privelege Level). Процессор в Р-режиме постоянно контролирует, чтобы текущая программа имела достаточный уровень привилегий для выполнения некоторых команд, обращения к данным других программ, передачи управления внешнему коду. Привилегированные команды разрешены на уровне 0, иначе генерируется прерывание ***GENERAL PROTECTION FAULT***. Программам не разрешена работа с данными на более высоком уровне привилегий. Передача управления между участками кода ограничивается тем же кольцом защиты, за некоторыми исключениями.

В защищенном режиме программа, как и в реальном режиме, по-прежнему состоит из сегментов, адресуемых с помощью 16-разрядных сегментных регистров, и местоположение адресуемого байта определяется заданием смещения в сегменте, однако преобразование виртуального адреса в физический осуществляется путем цепочки следующих преобразований. Сначала процессор преобразует виртуальный адрес в линейный путем извлечения из сегментного регистра селектора, определяющего номер дескриптора сегмента в локальной или глобальной таблице дескрипторов. Из дескриптора сегмента извлекается базовый линейный 32-разрядный адрес сегмента, к нему прибавляется 32-разрядное число, в результате чего получается 32-разрядное число, называемое линейным адресом.

Линейные адреса не связаны с какими-либо физическими объектами. Это фиктивные адреса, которые впоследствии преобразуются в физический адрес оперативной памяти. Физический адрес оперативной памяти зависит от реального объема памяти в системе и может располагаться в любом месте памяти, кроме первого мегабайта.

Для 32-разрядных приложений в Windows формируются сегмент команд и сегмент данных. Сегмент стека объединяется с сегментом данных, и этот объединенный сегмент

используется для размещения как глобальных переменных (в начале сегмента), так и локальных переменных. При загрузке программы в память регистры **ds** и **ss** принимают одно и то же значение. Обращение к локальным переменным осуществляется с использованием **ss** и относительно больших смещений, а при обращении к глобальным переменным используется регистр **ds** и малые смещения. Сегменту команд и сегменту данных для всех приложений назначаются селекторы 0x1B и 0x23. Базы обоих сегментов равны нулю, а размер сегмента равен 4 Гб. Таким образом, каждому запущенному приложению предоставляется все линейное адресное пространство. Принято говорить, что такие приложения работают в плоской модели памяти – модели **flat**. Так как базовые линейные адреса сегментов программы равны нулю, то виртуальные смещения совпадают с абсолютными линейными адресами. При этом получается, что все приложения используют одни и те же диапазоны линейных адресов. Для того чтобы приложения при одинаковых линейных адресах занимали отдельные участки физической памяти, не затирая друг друга, Windows при смене текущего приложения изменяет таблицы страничной трансляции.

1.3.3 Long-режим

Long-режим - расширение устаревшего защищенного (protected) режима. Long-режим состоит из двух подвидов: 64-битный режим и режим совместимости. 64-битный режим поддерживает все новые возможности и регистровые расширения, введенные в x86-64. Режим совместимости поддерживает бинарную совместимость с существующим 16-битным и 32-битным кодом. Long-режим не поддерживает устаревший реальный (real) режим или устаревший виртуальный (virtual-8086) режим, а также не поддерживает аппаратное переключение задач.

Поскольку 64-битный режим поддерживает 64-битное адресное пространство, то для его работы необходимо использовать новую 64-битную операционную систему. Существующие приложения при этом могут запускаться без перекомпиляции в режиме совместимости под операционной системой, работающей в 64-битном режиме. Для 64-битной адресации команд используется 64-битный регистр (RIP) и новый режим адресации с единым плоским (flat) адресным пространством и единым пространством для кода, стека и данных.

64-битный режим реализует поддержку расширенных регистров через новую группу префиксов команд REX. В 64-битном режиме размер адресов по умолчанию 64 бита, однако реализации x86-64 могут иметь меньший размер. Размер операнда по умолчанию 32 бита.

Для большинства инструкций размер операнда по умолчанию может быть перекрыт с использованием префикса команд типа REX.

64-битный режим обеспечивает адресацию данных относительно 64-битного регистра RIP. x86 архитектура обеспечивала адресацию относительно IP регистра только в командах передачи управления. RIP-относительная адресация повышает эффективность позиционно-независимого кода и кода, который адресует глобальные данные.

Несколько кодов операций (opcode) команд были переопределены для поддержки расширенных регистров и 64-битной адресации.

Режим совместимости предназначен для выполнения в 64-битной операционной системе существующих 16-битных и 32-битных программ. Приложения запускаются в режиме совместимости с использованием 32- или 16-битного адресного пространства и могут иметь доступ к 4Гб виртуального адресного пространства. Префиксы команд могут переключать 16- и 32-битные адреса и размеры операндов.

С точки зрения приложения, режим совместимости выглядит как устаревший защищенный режим x86, однако с точки зрения операционной системы (трансляция адресов, обработки прерываний и исключений) используются 64-битные механизмы.

Серьезных недостатков в 32-битную архитектуру новая архитектура AMD x86-64 не внесла. Отметить можно разве лишь чуть большие требования программ к памяти из-за того, что увеличился размер адресов и операндов. Однако это серьезно не скажется ни на размере кода, ни на требованиях к объему доступной оперативной памяти.

Но фактом является и то, что AMD x86-64 не привнесла ничего существенно нового. Нет принципиального повышения производительности. В среднем после перекомпиляции программы можно ожидать прирост производительности в пределах 5-15%.

Практически все современные операционные системы сейчас имеют версии для архитектуры AMD64. Так Microsoft предоставляет Windows XP 64bit, Windows Server 2003 64bit, Windows Vista 64bit. Крупнейшие разработчики UNIX систем также поставляют 64-битные версии, как например Linux Debian 3.1 x86-64. Однако это не означает, что весь код такой системы является полностью 64-битным. Часть кода ОС и многие приложения вполне могут оставаться 32-битными.

64-битная версия Windows, к примеру, использует специальный режим WoW (Windows-on-Windows 64), который транслирует вызовы 32-битных приложений к ресурсам 64-битной операционной системы. Рассмотрим более подробно программную модель Win64. Начнем с адресного пространства. Хотя 64-битный процессор теоретически может адресовать 16 экзабайт памяти (2^{64}), Win64 в настоящий момент поддерживает 16 терабайт (2^{44}). Этому есть несколько причин. Текущие процессоры могут обеспечивать доступ лишь к

1 терабайту (2^{40}) физической памяти. Архитектура (но не аппаратная часть) может расширить это пространство до 4 петабайт (2^{52}). Требуется память для хранения системных таблиц страниц памяти. Как и в Win32, адресуемый диапазон памяти делится на пользовательские адреса и на системные. Каждый процесс получает 8Тб и 8Тб остается системе (в отличие от 2Гб и 2Гб в Win32 соответственно).

Так же, как и в Win32, размер страницы составляет 4Кб. Первые 64Кб адресного пространства никогда не отображаются, т.е. наименьший правильный адрес это 0x10000. В отличие от Win32, системные DLL загружаются выше 4Гб. Все процессоры, реализующие 64-битную архитектуру, имеют поддержку для "CPU No Execution" бита, который Windows использует для реализации аппаратной технологии "Data Execution Protection" (DEP), которая запрещает исполнение пользовательских данных вместо кода. Это позволяет повысить надежность программ, исключая влияние ошибок типа выполнения буфера с данными как кода.

Особенность 64-битных компиляторов в том, что они могут наиболее эффективно использовать регистры для передачи параметров в функции, вместо использования стека. Это позволило разработчикам Win64 архитектуры избавиться от такого понятия как соглашение о вызовах (calling convention). В Win32 можно использовать разные соглашения (способы передачи параметров): __stdcall, __cdecl, __fastcall и т.д. В Win64 есть только одно соглашение о вызовах. Рассмотрим пример, как передаются в регистрах четыре аргумента типа int:

- RCX: первый аргумент
- RDX: второй аргумент
- R8: третий аргумент
- R9: четвертый аргумент

Аргументы после первых четырех int передаются на стеке. Для float-аргументов используются XMM0-XMM3 регистры, а также стек. Разница в соглашениях о вызове приводит к тому, что в одной программе нельзя использовать и 64-битный, и 32-битный код. Другими словами, если приложение скомпилировано для 64-битного режима, то все используемые библиотеки (DLL) также должны быть 64-битными.

1.4 ПРОСТРАНСТВО ВВОДА-ВЫВОДА И ПОРТЫ

Управление периферийным оборудованием может осуществляться различными способами:

■ с использованием вызовов функций ОС (прерывания DOS, API Windows);

- с использованием вызовов функций базовой системы ввода-вывода (BIOS);
- непосредственно взаимодействуя с регистрами периферийных устройств или контроллеров интерфейсов.

Управление периферийным оборудованием с использованием функций API Windows и программирование на Ассемблере в ОС Windows рассмотрено в пособии автора «Win32: основы программирования». Однако для этого необходимо хорошо представлять архитектуру и логику работу периферийных устройств компьютера.

Все устройства компьютера можно по способу обращения к ним разбить на две категории. В первую входят оперативная и видеопамять, ПЗУ, которые отличаются большим объемом хранимой информации и соответственно большим диапазоном используемых адресов. Ко второй категории относятся все периферийные и многие внутренние устройства компьютера, например, контроллеры подключаемой аппаратуры, которые не содержат большого объема памяти и требуют для работы небольшого числа адресов. Так как каждое устройство ввода-вывода, каждое системное устройство обычно имеет 1 или несколько 8, 16, или 32-битовых регистра, то обращение к регистрам устройства осуществляется не через пространство памяти, а через так называемое пространство ввода-вывода. Это адресное пространство физически независимо от оперативной памяти, что обеспечивается наличием специального выходного сигнала процессора, разрешающего работу либо с устройствами, подключенными к системной шине через пространство памяти, либо с устройствами, подключенными через пространство ввода-вывода. Адресное пространство ввода-вывода разбито на участки, называемые портами ввода-вывода. Сведения о номерах портов, их разрядности, формате управляющей информации, о логике работы устройств приводятся, как правило, в соответствующей документации, в настоящем пособии эта информация будет приводиться по мере изучения курса и рассмотрении программно-управляемых устройств и подсистем компьютера.

Программное разделение пространства памяти и пространства ввода-вывода реализуется с помощью двух наборов команд процессора – для памяти и для устройств. В первую группу команд входят практически все команды процессора, с помощью которых можно обратиться по адресу памяти – команды пересылки, команды арифметических операций, команды сдвигов и т.д. Вторую группу команд образуют команды чтения из порта ***in*** и записи в порт ***out***, а также модификации этих команд ***ins*** и ***outs,*** обеспечивающие обмен с устройством последовательностью байтов данных.

Команды ***in*** и ***out*** имеют следующий формат:

Команда чтения из порта:

in регистр-аккумулятор,№_порта

Команда записи в порт:

out №_порта,регистр-аккумулятор

Отметим, что ***регистр-аккумулятор*** – это ***al***, ***ax*** или ***eax***.

1.5 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие режимы работы процессора Вы знаете ?
2. Какие регистры характерны для long-режима?
3. Поясните понятие сегментации памяти.
4. Что содержат сегментные регистры в реальном и защищенном режимах ?
4. Какие системные регистры Вы знаете?
5. Чем различаются локальная и глобальная таблицы дескрипторов?
6. Отличие сегментной организации памяти от сегментно-страничной.
7. Методы обращения к внешним устройствам.
8. Что такое пространство ввода-вывода?
9. Что такое порт?
10. Команды работы с портами.

1.6 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Минимально адресуемой единицей информации является:

   а) бит

   б) байт

   в) слово

   г) двойное слово

   д) сегмент

2. Имеется несколько утверждений:

   1) в реальном режиме доступ к сегментам может иметь любая программа;

   2) любая программа, написанная для функционирования в R-режиме, может быть выполнена в защищенном режиме;

   3) сегментные регистры в R-режиме содержат либо часть адреса начала сегмента, либо дескриптор этого сегмента;

   4) в P-режиме в сегментных регистрах содержится информация, по которой осуществляется доступ к дескриптору сегмента в соответствующих таблицах дескрипторов;

5) локальная таблица дескрипторов может содержать дескрипторы тех же типов, что и глобальная.

Из этих утверждений верно: а - 1; б - 1,2; в - 1,4; г - 2,3; д - 3,5.

3. Укажите правильную команду чтения из порта:

а) ***in al,60h***

б) ***in bx,60h***

в) ***out al,60h***

г) ***out 60h,al***

Ответы: 1 - б; 2 - в; 3 – а.

2 ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

2.1 ОПИСАНИЕ ДАННЫХ И СПОСОБЫ АДРЕСАЦИИ

Как было отмечено выше, минимальной порцией адресуемой информации является байт. Кроме байтов, Ассемблер работает с более сложными данными. Например, со словами (WORD - 2 байта), двойными словами (DWORD - 4 байта), а также с 6, 8 и 10-байтными данные (FWORD, QWORD и TBYTE). При этом нужно помнить, что младшие байты хранятся по меньшим адресам. Например, число 0125h хранится в памяти так

	A	**A+1**	
	25	01	

Это связано с тем, что в младших моделях ЭВМ арифметические операции начинаются с действий над младшим байтом, в связи с чем сохраняется преемственность. В регистрах же числа хранятся в нормальном виде.

Описываются данные с помощью специальных директив, в которых указывается имя описываемой структуры данных, символическое имя директивы и значение структуры данных при инициализации. Если структура данных не инициализируется, то в качестве значения для инициализации используется ***?***. Символическое имя директивы указывает размер описываемых данных – байт (***db***), слово (***dw***), двойное слово (***dd***), шесть байтов (***df***), восемь байтов (***dq***), десять байтов (***dt***). Видно, что мнемоника символического имени связана с описываемыми данными, в частности ***db*** означает ***d***efine ***b***yte - определить байт. При необходимости описания нескольких полей данных одного типа после символического имени указывается количество повторений, за которым следует конструкция ***dup(значение_повторяемого_элемента_при_инициализации)***. В качестве значения повторяемого элемента при инициализации может использоваться константа, выражение, или аналогичная конструкция ***dup*** для резервирования памяти для многомерных массивов. Символьные данные хранятся как набор байтов, т.е. 1 символ соответствует 1 байту.

Пример 2.1:

```
PROG db `my_program`;  строка символов
A dd 012h;  двойное слово инициализируется шестнадцатеричным значением 012h
AL1 db 00100011b ;  байт инициализируется двоичным числом
B db 0,0,0,0,0,0,0,0,0,0;  массив из 10 элементов с инициализацией нулями
B1 db 10 DUP(8);  массив из 10 элементов с инициализацией числом 8
```

C db 8 DUP(8 DUP(?)); массив из 64 элементов без инициализации

В языке Ассемблер имеются более сложные типы данных, такие как структуры, объединения, указатели, битовые последовательности, упакованные и неупакованные двоично-десятичные числа.

При программировании на Ассемблере в среде Windows широко используются структуры, поэтому рассмотрим их более подробно. Как известно, структура состоит из фиксированного числа элементов различных типов. Для использования структуры в программе сначала необходимо описать элементы структуры. Это делается с помощью конструкции **STRUC**, имеющей следующий синтаксис:

```
имя_структуры STRUC
описание элементов
имя_структуры ENDS
```

Описание элементов представляет собой последовательность директив описания данных **db, dw, dd** и т.п., где определяются размеры элементов структуры и, возможно, их начальные значения.

Пример 2.2. Описание структуры.

```
group STRUC ; структура с данными о студенческой группе
name db 15 dup(' ') ; название группы
kurs db ?           ; курс
kol_stud db 0       ; количество студентов
group ENDS
```

Появление конструкции **STRUC** в программе не приводит к выделению памяти под структуру. Для того чтобы работать со структурой, программист должен описать конкретную переменной или массив с типом определенной ранее структуры. Синтаксис такого описания имеет вид:

имя_переменной имя_структуры < [список_значений] >

Заключенный в угловые скобки **список_значений** представляет собой список начальных значений элементов структуры, разделенных запятыми. Задание списка является необязательным, кроме того, допускается инициализация только отдельных элементов, но в этом случае обязательно указание запятых. Пропущенные поля инициализируются значениями из описания структуры. Если программиста устраивают значения элементов структуры в **STRUC,** то список значений можно не указывать.

Пример 2.3. Выделение памяти под переменные типа «структура»

```
gr1 group <'IVT41',3,25>
```

```
gr2 group <'PI31',2,20>
gr3 group <>
```

Рассмотрим организацию работы с элементами структуры. К ее элементам можно обращаться через уточненное имя, представляющее разделенные точкой имя структуры и имя элемента. Альтернативным способом обращения к элементу структуры является указание в уточненном имени вместо имени структуры ее адреса, хранящегося в регистре. Например, запись в регистр ***al*** элемента ***kurs*** реализуется командой:

```
mov al, byte ptr gr1.kurs
```

или командами

```
lea bx,gr1      ; в bx запишем адрес gr1
mov al,byte ptr [bx].kurs
```

Напомним, что команда **mov приемник,источник** служит для пересылки данных из источника в приемник. Ограничения в использовании этой команды заключаются в следующем. Во-первых, командой **mov** нельзя осуществлять пересылку из одной области памяти в другую, а если такая операция необходима, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения. Во-вторых, нельзя загружать в сегментный регистр значение непосредственно из памяти. Для загрузки данных в сегментный регистр обычно используют регистр общего назначения или стек. В-третьих, нельзя пересылать содержимое между сегментными регистрами и нельзя записывать данные в регистр ***cx***.

Поясним также использование **byte ptr** в приведенных выше командах пересылки. При обращении к памяти (**gr1.kurs**) или при использовании для хранения адреса памяти регистра транслятор не всегда однозначно может определить, сколько именно байтов необходимо пересылать – один, два или больше. Иногда транслятор анализирует второй операнд и по его размеру принимает решение о количестве пересылаемых байтов, но это бывает не всегда. Поэтому в командах пересылки программисту предоставляется возможность явного указания типа пересылаемых данных, определяющего количество байтов. Для этой цели перед операндом пишется конструкция вида **тип_данных ptr,** где **тип_данных** принимает значения **byte,word,dword** и т.п.

В рассмотренных выше командах использованы различные методы адресации данных. Так как впоследствии при написании программ потребуется практическое использование большинства существующих методов адресации, то приведем их краткую характеристику.

Регистровая адресация

Операнд (байт или слово) находится в регистре. Например:

```
mov ax,bx     ; пересылка содержимого bx в ax
mov dl,ah     ; пересылка содержимого ah в dl
```

Непосредственная адресация

Операнд (байт или слово) может быть представлен в виде числа, адреса, кода ASCII или символа. Например:

```
mov ax,4C00h       ; операнд – 16-ричное число
mov dx,offset mas  ; операнд – адрес начала массива mas
mov dh,'&'         ; операнд – ASCII-код символа '&'
```

Прямая адресация памяти

В команде указывается символическое обозначение ячейки памяти, над содержимым которой требуется выполнить операцию, или если операндом в команде является содержимое ячейки с известным абсолютным адресом. Например:

```
mov bl,mem    ; содержимое байта с именем mem пишем в bl
mov ax,es:[0];содержимое слова по адресу в es пишем в ax
```

Все остальные способы адресации относятся к косвенной адресации памяти.

Базовая и индексная адресация памяти

Относительный адрес ячейки памяти находится в регистре, обозначение которого заключается в квадратные скобки. При использовании регистров ***bx*** и ***bp*** адресацию называют базовой, при использовании регистров ***si*** и ***di*** – индексной. При адресации через регистры ***bx***, ***si*** или ***di*** в качестве сегментного регистра подразумевается ***ds***, при адресации через ***bp*** – регистр ***ss***. Таким образом, косвенная адресация через регистр ***bp*** предназначена для работы со стеком. Однако при необходимости можно явно указать требуемый сегментный регистр. Во всех базовых и индексных способах адресации операндом является содержимое ячейки памяти, адрес которой находится в одном из указанных выше регистров или вычисляется как сумма содержимого двух регистров. Например:

```
mov al,[bx] ; байт по адресу в bx пишем в al, сегмент в ds
mov dx,[bp] ; слово по адресу в bp пишем в dx, сегмент в ss
```

Базовая и индексная адресация со смещением

Относительный адрес операнда определяется суммой содержимого регистра (***bx,bp,si*** или ***di***) и указанной в команде константы или адреса, который может быть задан с помощью указания имени ячейки памяти. Например, для загрузки в регистр dl значения третьего элемента массива из 5 байтов можно использовать следующие команды:

```
Mas db 1,3,7,8,9     ; объявление и инициализация массива
```

```
mov bx,2          ; в bx заносим индекс элемента
mov dl,mas[bx]    ; в dl заносим третий элемент массива
```

Тот же результат будет получен с применением команд:

```
mov bx,offset mas ;в bx запишем относительный адрес mas
mov dl,2[bx]
```

Последняя команда может быть записана в виде

```
mov dl,[bx+2]
```

или в виде

```
mov dl,[bx]+2.
```

Базово-индексная адресация памяти

Относительный адрес операнда определяется суммой содержимого базового и индексного регистров. В качестве базового допускается использование ***bx*** или ***bp,*** а в качестве индексного - ***si*** или ***di***. Если в качестве базового используется регистр ***bx***, то по умолчанию сегментным регистром считается ***ds***, а если базовый регистр ***bp***, то сегментный по умолчанию – ***ss***. Можно явно указывать нужный сегментный регистр. Например:

```
mov bx,[bp][si]
;в bx пишем слово по адресу, равному сумме содержимого bp и si.
;Сегментный адрес в ss.
mov [bx][di],ax
;в слово по адресу, равному сумме содержимого bx и di, пишем
;число из ax. Сегментный адрес в ds.
```

Базово-индексная адресация со смещением

Отличается от базово-индексной адресации тем, что относительный адрес операнда определяется суммой трех величин: содержимого базового и индексного регистров, а также дополнительного смещения. Например:

```
mov dl,[bx+si+2]
mov dl,2[bx][si]
mov dl,matr[bp][di]
```

2.2 ОСНОВНЫЕ КОМАНДЫ ОБМЕНА ДАННЫМИ, АРИФМЕТИЧЕСКИЕ И ЛОГИЧЕСКИЕ КОМАНДЫ

К основным командам обмена данными относятся **mov,xchg,lea**,а также группа команд для работы со стеком - **push**, **pop,pusha,popa,pushf, popf**. Синтаксис

команды **mov** рассмотрен выше. Команда **xchg** служит для обмена двух операндов, причем не допускается обмен «память-память». Например:

```
xchg dx,cx  ; обмен содержимым dx и cx
xchg ax,word ptr[di];обмен между ax и словом по адресу в di
```

Команда **lea** служит для загрузки адреса памяти в регистр, например:

```
lea bx,mas ; загрузка адреса массива mas в bx
```

Рассмотрим команды для работы со стеком. Стек – это область памяти, выделяемая для временного хранения данных программы. Для работы со стеком предназначены сегментный регистр ***ss***, регистр указателя стека ***esp***(***sp***) и регистр базы стека ***ebp***(***bp***). Запись и чтение в стеке осуществляется в соответствии с принципом ***Last In First Out***, и по мере записи данных в стек он растет в сторону младших адресов. Регистр ***esp***(***sp***) всегда указывает на последний записанный в стек элемент. Если стек пуст, то значение этого регистра равно адресу последнего байта сегмента, выделенного под стек. При занесении элемента в стек сначала уменьшается значение ***esp***(***sp***), а затем по адресу в этом регистре записывается элемент. Для записи в стек используется команда

push *элемент*

При извлечении данных из стека сначала копируется элемент по адресу из ***esp***(***sp***), а затем значение этого регистра увеличивается. Для чтения из стека используется команда

pop *элемент*

Например, для записи в регистр ***es*** значения из ***ds*** можно использовать команды:

```
push ds
pop es
```

Если необходимо получить доступ к элементам не на вершине стека, а внутри, то применяют регистр ***ebp***(***bp***), в который записывается содержимое регистра ***esp***(***sp***), а уже для регистра ***ebp***(***bp***) можно использовать адресацию со смещением.

Команды **pusha** и **popa** служат для сохранения в стеке группы регистров общего назначения и не имеют операндов. Команда **pushf** также не имеет операндов и сохраняет в стеке регистр флагов. Команда **popf** восстанавливает регистр флагов из стека.

Теперь перейдем к рассмотрению основных арифметических и логических команд Ассемблера, использующихся для выполнения целочисленных операций. Команды, обрабатывающие данные с плавающей точкой, являются нехарактерными для программирования устройств компьютера через порты, поэтому в настоящем учебном пособии не рассматриваются.

К наиболее часто используемым целочисленным арифметическим командам относятся команды двоичной арифметики (сложение - **add,inc**; вычитание - **sub,dec**; умножение – **imul,mul**; деление – **idiv,div**; изменение знака – **neg**) и команды преобразования типов (**cbw,cwd** и т.п.). Команды **add** и **sub** имеют два операнда, над которыми выполняется арифметическая операция (соответственно сложение или вычитание), результат которой записывается в первый операнд. Команды **inc** и **dec** имеют один операнд, значение которого соответственно увеличивается или уменьшается на единицу.

При выполнении умножения и деления могут быть использованы команды, учитывающие или не учитывающие знак операндов. Команды **mul** и **div** работают с беззнаковыми операндами, а команды **imul** и **idiv** учитывают знак. Рассмотрим выполнение умножения чисел со знаком. Команда **imul** по синтаксису может иметь один, два или три операнда. Форма команды с одним операндом требует явного указания расположения только первого сомножителя, который может быть расположен в ячейке памяти или регистре. Расположение второго сомножителя фиксировано и зависит от размера первого сомножителя:

- если операнд в команде - байт, то второй сомножитель должен располагаться в ***al***;

- если операнд - слово, то второй сомножитель должен располагаться в ***ax***;

- если операнд в команде – двойное слово, то второй сомножитель располагается в регистре ***eax***.

Результат умножения для команды с одним операндом помещается также в строго определенное место, определяемое размером сомножителей:

- при умножении байтов произведение помещается в ***ax***;

- при умножении слов результат помещается в пару регистров ***dx:ax*** (в ***dx*** - старшая часть произведения);

- при умножении двойных слов результат помещается в пару ***edx:eax***.

В команде с двумя операндами первый и второй операнды определяют местоположение первого и второго сомножителей, результат будет записан на место первого сомножителя. В команде с тремя операндами первый операнд определяет местоположение результата, второй операнд – местоположение первого сомножителя, третий операнд может быть непосредственно заданным значением размером в байт, слово или двойное слово. Команды умножения влияют на флаги переноса и переполнения в регистре флагов (CF и OF). Если результат мал и помещается в младшую часть, отведенную под хранение произведения, то CF=OF=0, и содержимое старшей части является расширением знака. В противном случае

факт переполнения или переноса должен быть обработан программно при дальнейшей обработке результата умножения.

При делении чисел, так же, как и при умножении, может учитываться или не учитываться знак. Рассмотрим команду знакового деления **idiv**, имеющую один операнд, который является делителем и может находиться в памяти или в регистре и иметь размер в байт, слово или двойное слово. Местонахождение делимого задается неявно по следующим правилам:

- если делитель размером в байт, то делимое должно быть расположено в регистре ***ax***. После операции частное помещается в ***al***, а остаток – в ***ah***;

- если делитель размером в слово, то делимое должно быть расположено в паре регистров ***dx:ax***, причем младшая часть делимого должна находиться в ***ax***. После деления частное помещается в ***ax***, а остаток – в ***dx***.

- если делитель размером с двойное слово, то делимое должно быть расположено в паре регистров ***edx:eax***. После деления частное помещается в ***eax***, а остаток в ***edx***.

Ошибка деления возникает при делении на ноль или если частное слишком велико для размещения в регистре ***eax/ax/al***.

Для того чтобы делимое размещать в паре регистров или чтобы расширять байт до слова, используются команды расширения знака **cbw** (расширение ***al*** до ***ax***) и **cwd** (расширение ***ax*** до ***dx:ax***).

К логическим командам, выполняющимися над битами операндов, относятся **and**, **or**, **xor**, **not** и **test**. Команды **and**, **or**, **xor** имеют два операнда, над которыми выполняются соответственно операции «логическое и», «логическое или», «логическое исключающее сложение». Результат операции записывается в первый операнд. Команда **not** имеет один операнд, над которым выполняется инвертирование каждого бита. Команда **test** имеет два операнда, служит для проверки их отдельных битов и выполняет поразрядно логическую операцию «И» над битами операндов. Удобство этой команды заключается в том, что сами операнды при этом не изменяются, а в результате работы команды устанавливаются флаги ZF, PF и SF в регистре флагов, которые в дальнейшем могут быть проанализированы.

С помощью логических команд возможно выделение отдельных битов в операнде, их установка, сброс, инвертирование, проверка на равенство определенному значению. Например, для установки в 1 нулевого и второго битов в регистре ***al*** используется команда **xor al,00000101b**, для сброса в ноль этих же битов можно использовать команду **and al,11111010b.** Для проверки второго бита ***al*** применяется команда

test al,00000100b,

в результате которой флаг ZF=0, если бит равен нулю.

Наряду с логическими командами отметим наличие в Ассемблере большой группы команд сдвига, так же, как и логические команды, обеспечивающих операции на уровне битов. Простейшими командами нециклического сдвига являются команды **shr** (сдвиг вправо) и **shl** (сдвиг влево), имеющие два операнда, первый из которых содержит сдвигаемое вправо или влево значение, а второй указывает на количество сдвигаемых битов. При нециклическом сдвиге освободившиеся биты заполняются нулями. По мере сдвига первого операнда сдвигаемые биты попадают во флаг переноса CF, который также может применяться для анализа значения сдвинутого бита. Команды циклического сдвига (**rol**, **ror**) сохраняют сдвигаемые биты в первом операнде, они не теряются. Примером практического применения команд сдвига является организация умножения или деления операнда на степени двойки путем его сдвига вправо (деление) или влево (умножение) на столько битов, какова степень двойки у множителя или делителя. Например:

```
mov ax,48 ; делим число в ax
shr ax,2  ; на 4
```

2.3 ПРОГРАММИРОВАНИЕ НЕЛИНЕЙНЫХ АЛГОРИТМОВ

В системе команд Ассемблера для организации нелинейных алгоритмов используются команды безусловного и условного переходов. Команда безусловного перехода

```
jmp метка
```

позволяет осуществить переход по адресу, где расположена метка, вне зависимости от каких-либо условий. Мнемоническое имя команды условного перехода начинается с символа **j**, после которого указываются символы, служащие, как правило, начальными буквами английского текста, поясняющего условие перехода на метку. Мнемонические обозначения и смысл основных команд условного перехода приведены в таблице 2.1.

Все условные переходы основаны на анализе флагов в регистре ***eflags***(***flags***). Эти флаги могут быть изменены с помощью арифметических и логических команд, а также с помощью специальных команд проверки условия **cmp** и организации цикла **loop.**

Команда **cmp** сравнивает два операнда путем выполнения вычитания из первого операнда второго, устанавливая при этом флаги, но не изменяя операндов. Например, пусть необходимо при равенстве содержимого регистра ax числу 29 записать в регистр ***bx*** число 9,

а в противном случае записать в ***bx*** 16. Значение в ***bx*** в дальнейшем должно складываться со значением в ***dx***.

Пример 2.4.

```
      cmp ax,29 ; вычтем из ax 29
      jnz zap16 ; если результат не ноль, то число в ax не
; равно 29, и переходим на метку zap16
      mov bx,9 ; если не ушли, то в ax 29, и в bx пишем 9
      jmp next  ; нужно пропустить оператор, заносящий в bx 16
zap16: mov bx,16  ; в bx пишем 16
next:  add dx,bx  ; сложение dx с нужным значением в bx
```

Таблица 2.1 – Основные знаковые и беззнаковые команды условного перехода

Мнемоника команды	Вид перехода
jg	**G**reater – переход, если больше
jge	**G**reater or **E**qual – переход, если больше или равно
jl	**L**ess - переход, если меньше
jle	**L**ess or **E**qual - переход, если меньше или равно
jz	**Z**ero – переход, если ноль
jnz	**N**ot **Z**ero- переход, если не ноль
je	**E**qual – переход, если равно
jne	**N**ot **E**qual – переход, если не равно
ja	**A**bove – переход, если больше (для беззнаковых команд)
jae	**A**bove or **E**qual – переход, если больше или равно (для беззнаковых команд)
jb	**B**elow – переход, если меньше (для беззнаковых команд)
jbe	**B**elow or **E**qual – переход, если меньше или равно (для беззнаковых команд)
jc	**C**arry – переход, если установлен флаг переноса CF

Как известно, циклы можно организовывать с использованием явной проверки условий завершения или продолжения циклов, а также с использованием счетчика. Для организации цикла с автоматическим уменьшением счетчика в Ассемблере используется команда

loop *метка*

которая предполагает занесение перед циклом в регистр ***cx***(***ecx***) начального значения счетчика, автоматический декремент счетчика перед командой **loop** и переход на метку, указанную в команде, если счетчик не равен нулю. Например, пусть необходимо

сформировать массив слов, состоящий из 10 последовательно увеличивающихся значений, начиная с 8. Это может быть осуществлено командами, приведенными в примере 2.5.

Пример 2.5.

```
    mas dw 10 dup(?)      ; резервирование 10 слов под элементы
        lea bx,mas        ; в bx запишем адрес массива mas
        mov ax,8          ; в ax – начальное значение элемента
        mov si,0          ; в si будем хранить смещение текущего
; элемента массива относительно начала массива, для первого
; элемента в si запишем ноль
        mov cx,10         ; в cx поместим счетчик цикла
cycl:   mov [bx][si],ax   ; формирование очередного элемента
        inc ax            ; увеличим текущее значение
        add si,2          ; переход к следующему элементу
        loop cycl         ; если счетчик не ноль, то на cycl
```

2.4 ПОДПРОГРАММЫ

Подпрограммы являются важнейшим средством любого языка программирования. Как правило, программа по ходу своего выполнения неоднократно обращается к определенным алгоритмам, например, преобразования чисел или строк. Такие алгоритмы оформляются в виде подпрограмм и вызываются из основной программы по мере необходимости. Для оформления подпрограммы может быть использован шаблон следующего вида:

```
имя_процедуры proc
  команды процедуры
  ret [количество_байтов]
имя_процедуры endp
```

Ключевые слова **proc** и **endp** служат, вообще говоря, для улучшения читабельности исходного кода, выделяя из общего текста тело процедуры. Оператор **ret** обеспечивает возврат в вызывающую программу, снимая со стека и загружая в регистр **eip(ip)** (или в **cs**:**ip** в случае межсегментного вызова в реальном режиме) адрес возврата, который помещается в стек при вызове подпрограммы оператором **call.** Напомним, что при выполнении команды **call** процессор помещает в стек адрес следующей команды, называемый адресом возврата, а в **eip** (**ip**) заносит адрес вызываемой процедуры.

Если после оператора **ret** указано число, то после снятия со стека адреса возврата содержимое регистра **esp**(**sp**) увеличится еще на указанное в **ret** число байтов. Это может потребоваться для автоматического снятия со стека параметров подпрограммы, если они передавались через стек.

Методы обмена данными основной программы и подпрограммы рассматривались ранее на примере МОДЕЛИ-2 Ассемблера. Так как передача аргументов в подпрограмму и возврат результатов ее работы в вызывающую программу являются очень важным аспектом взаимосвязи программ на Ассемблере и программ, написанных на языке высокого уровня, а также применяются при организации вызовов API-функций WINDOWS, рассмотрим достоинства и недостатки информационных связей между модулями в Ассемблере более подробно.

Основными методами передачи аргументов в процедуру являются передача данных через регистры, через общую область памяти, через стек.

Передача аргументов через регистры является наиболее простым в реализации способом передачи данных. Его достоинствами являются немедленная доступность данных после передачи управления процедуре. Однако число доступных для пользователя регистров является сравнительно небольшим, и размер передаваемых данных ограничен размерами регистра. Если размер данных велик, то приходится передавать через регистры не сами данные, а их адреса. В процедуре нужно извлечь эти адреса, загрузить их в регистры и только потом через адреса в регистрах получить доступ к данным.

Передача данных через общую область предполагает существование области данных, доступной как вызывающей, так и вызываемой подпрограмме. Недостатком использования этого способа в реальном режиме является отсутствие средств защиты данных от разрушения.

Наиболее часто для передачи аргументов при вызове процедуры используется стек. Пусть вызывающая программа заносит в стек передаваемые данные, после чего вызывает подпрограмму командой **call**. В этом случае аргументы будут размещены в стеке под адресом возврата. Возможность доступа подпрограммы к аргументам без разрушения стека реализуется с помощью регистра **ebp** (**bp**). Перед использованием этого регистра для доступа к данным стека его необходимо правильно инициализировать, для чего в начало процедуры включаются команды:

```
push bp
mov bp,sp
```

Первая из них сохраняет содержимое **bp** в стеке, чтобы исключить порчу его значения в процедуре. Вторая команда настраивает **bp** на вершину стека. При этом надо помнить, что

если в стек сохранен регистр **ip**, то самый близкий к верхушке стека аргумент расположен по адресу **bp+4** за счет **ip** и самого только что сохраненного **bp**. Для извлечения из стека аргументов используются обычные команды пересылки данных:

```
mov ax,[bp+4]   ; извлекаем аргумент N
mov bx,[bp+6]   ; извлекаем аргумент N-1
```

После обработки аргументов нужно не забыть восстановить ***sp*** и старое значение ***bp*** до входа в процедуру командами:

```
mov sp,bp
pop bp
```

В конце процедуры стек освобождается от аргументов командой **ret N,** где **N** – количество_байтов, занимаемых аргументами. Отметим, что освободить стек можно в вызывающей программе применением команд **pop**, либо прямой корректировкой ***sp*** путем сложения его с количеством байтов, занимаемых аргументами.

Организация возврата из процедуры в Ассемблере полностью ложится на программиста. В общем случае программист располагает тремя способами возврата значений из процедуры:

- С использованием регистров. Этот способ является самым быстрым и имеет те же недостатки, что и при передаче аргументов.
- С использованием общей области памяти. Этот способ удобен при возврате большого количества данных.
- С использованием стека. При этом возможно применение для возвращаемых аргументов тех же ячеек в стеке, которые использовались для аргументов. Альтернативным способом использования стека для хранения результатов работы процедуры является предварительное помещение в стек наряду с передаваемыми аргументами фиктивных значений с целью резервирования места для возвращаемых значений. Очевидно, что при использовании передачи результатов через стек его нельзя очищать в процедуре.

Пример практического использования подпрограмм приведен ниже.

2.5 ОПИСАНИЕ СЕГМЕНТОВ. СТРУКТУРА ПРОГРАММЫ НА АССЕМБЛЕРЕ В ФОРМАТАХ **.EXE* И **.COM*

Сегменты описываются следующим образом:

```
<имя сегмента> SEGMENT <выравнивание> <совмещение> <класс>
                       <тип размера>
```

```
        [<описание данных>]
        [<команды>]
<имя сегмента> ENDS
```

Выравнивание определяет размещение начального адреса описываемого сегмента при его возможной компоновке в один сегмент с другими сегментами. Допустимые значения атрибута выравнивания следующие:

BYTE - размещение с адреса, кратного байту, т.е. произвольно;

WORD - сегмент размещается с адреса, кратного слову (2 байта);

DWORD - сегмент начинается с адреса, кратного 4 ;

PARA - размещение с адреса, кратного параграфу (16 байтов), т.е. последняя шестнадцатеричная цифра равна 0;

PAGE - сегмент начинается по адресу, кратному 256, т.е. происходит выравнивание на границу страницы размером 256 байтов;

MEMPAGE - размещение сегмента происходит с адреса, кратному 4 Кбайт.

По умолчанию принят тип выравнивания PARA.

Совмещение указывает, как нужно комбинировать сегменты различных модулей, имеющих одинаковое имя. Атрибут совмещения может принимать следующие значения:

PRIVATE - значение атрибута по умолчанию, сегмент объединению с другими не подлежит;

PUBLIC - сегменты с одинаковыми именами при компоновке можно объединить. Новый сегмент будет непрерывным, адреса объектов будут вычисляться относительно начала нового созданного сегмента;

COMMON - все сегменты с одним и тем же именем располагаются по одному адресу. Сегменты с одинаковым именем будут перекрываться и совместно использовать память. Размер результирующего сегмента будет равен размеру самого большого сегмента;

AT xxxx - сегмент располагается по абсолютному адресу параграфа xxxx. Все метки и адреса отсчитываются относительно заданного абсолютного адреса;

STACK - все стековые сегменты можно объединить в один, они последовательно разместятся в памяти.

Класс - это дополнительное имя в кавычках, например ‘CODE’, ‘STACK’ или ‘DATA’. Сегменты, имеющие одинаковые дополнительные имена, компонуются в один физический сегмент.

Тип размера сегмента - USE16 или USE32 - указывает, какую адресацию допускает сегмент (16-ти или 32-разрядную). Сегмент с атрибутом USE16 может содержать до 64 Кбайт кода или данных, а сегмент с атрибутом USE32 может содержать до 4 Гбайт кода или данных.

Описание сегмента не содержит информации о его функциональном назначении, поэтому для использования сегмента необходимо сообщить транслятору, для чего предназначен сегментов содержащих команды, данные или стек. Для сопоставления сегментных регистров и сегментов используется директива **assume**:

```
assume cs: cseg, ds: data, ss: sseg
```

Однако надо помнить, что это закрепление носит формальный характер и директива **assume** сообщает только намерение адресовать **cseg** через ***cs***, **data** через ***ds***. Непосредственно инициализация ***ds*** осуществляется по-разному в зависимости от формата программы - ****.exe*** или ****.com***.

Файл типа *.com содержит образ программы в памяти, поэтому после установки среды и распределения блока памяти для программы операционная система заполняет содержимое 256 байтов в начале этого блока служебной информацией и в память считывается файл ****.com*** по смещению ***100h*** от начала блока без настройки. Происходит инициализация сегментных регистров, а также регистров ***eip*** и ***esp***, затем программа ****.com*** начинает выполнение.

Файл типа ****.exe*** состоит из нескольких разделов, включающих заголовок файла ****.exe***, таблицы настройки и образ программы. В связи с тем, что в реальном режиме адрес сегмента в программе будет зависеть от того, где он загружен в памяти, необходимо иметь возможность обновить ячейки в программе, где делается ссылка на данный сегмент. Этот процесс называется настройкой. Таблицы настройки включают списки, где в программе делаются явные ссылки на программу или кодовый сегмент по его адресу. Во время настройки происходит обновление образа загрузки, в программу включаются действительные значения сегмента. Как и для файлов типа ****.com***, после распределения блока памяти создается специальная область из 256 байтов - ***PSP*** (префикс программного сегмента). Затем образ программы читается в память выше ***PSP***, считывается таблица настройки и начинается настройка образа программы. Первым шагом при настройке является вычисление адреса начала сегмента, который является адресом реальной памяти. Когда программа-компоновщик строит образ программы, она использует предполагаемый базовый сегмент 0000. Если на самом деле программа загружается в сегмент с адресом A, то к каждой ссылке на адрес сегмента необходимо добавить A.

После завершения настройки регистры ***es*** и ***ds*** процесса устанавливаются на адрес сегмента ***PSP***, а регистры ***cs:ip*** и ***ss:sp*** инициализируются значениями, данными в заголовке программного файла типа ****.exe***. Оба регистра ***cs*** и **ss** увеличиваются на адрес начала сегмента образа программы.

Рассмотрим пример описания сегментов данных, кода и стека для программы в формате ****exe***, выводящей на экран слово “HELLO !”. Собственно вывод строки реализуем с

помощью функции 09h прерывания 21h, требующей, чтобы пара регистров ***ds*** и ***dx*** указывала на сегмент и смещение выводимой строки.

После завершения работы программы необходимо организовать возврат управления операционной системе. Наиболее просто это может быть осуществлено следующими способами:

1. Через функцию 4Ch прерывания 21h:

```
mov   ax,4C00h
int   21h
```

2. Через прерывание 20h

```
int   20h
```

Пример 2.7. *Описание сегментов программы в формате *.exe*

```
data segment para public 'data'      ; заголовок сегмента данных
          soob db   'hello !','$'    ; строка заканчивается символом $,
                                     ; используемым как ограничитель
data ends                            ; при выводе
cseg segment para  public 'code'     ;заголовок сегмента кода
assume    cs:cseg, ds:data,ss:sseg   ; сопоставление сегментных регистров с
                                     ;сегментами для трансляции
main:     mov ax,data                ; в ax занесем адрес сегмента данных
          mov ds,ax                  ; загрузим адрес сегмента данных в ds
          mov dx, offset soob        ; в dx поместим адрес строки soob
          mov ah,09h                 ; для вывода строки нужна функция 09h
          int 21h                    ; прерывания 21h
          mov ax,4c00h               ;используем функцию 4Ch и подфункцию
          int 21h                    ; 00h прерывания 21h для выхода
cseg ends                            ;конец описания сегмента кода
sseg segment para stack 'stack'      ;заголовок сегмента стека
          db 100h dup(0)             ;резервирование под стек 256 байтов
sseg ends                            ;конец описания сегмента стека
END main                             ;директива для окончания трансляции
```

Директива **END** служит для завершения трансляции. Имя после **END** - это имя точки входа в программу. Если ее нет, то управление при выполнении программы передается на начало кодового сегмента.

Возможно сокращенное описание сегментов с помощью упрощенных специальных директив сегментации и директивы указания модели памяти ***model***, которая частично управляет размещением сегментов и выполняет функции директивы assume. Обязательным аргументом директивы ***model*** является имя модели памяти. В модели **tiny** код и данные объединены в один сегмент, ссылки на данные и код имеют тип **near** (то есть для доступа к коду и данным достаточно изменить только смещение внутри сегмента, без смены содержимого сегментного регистра), эта модель используется для создания программ в формате ****.com***. Модель **small** предполагает использование 1 сегмента кода и 1 сегмента данных, ссылки на данные и код имеют тип **near**. В модели **medium** данные занимают один, а код - несколько сегментов, все ссылки на передачу управления имеют тип **far** (при доступе к коду изменяется не только смещение, но и адрес сегмента, т.е. происходит смена содержимого сегментного регистра). Модель **compact** подразумевает использование 1 сегмента кода и несколько сегментов данных типа **far**. Модели **large**, **huge** служат для работы с несколькими сегментами кода и несколькими сегментами данных, причем данные и код имеют тип **far**. В настоящее время при программировании под WINDOWS используется плоская модель памяти **flat**.

Пакет Turbo Assembler, с которым мы будем работать при изучении материала, допускает описание сегментов программы с применением упрощенных директив, что отражено в примере 2.8.

Пример 2.8. *Описание сегментов программы в формате *.exe с использованием упрощенных директив*

```
.model small  ; будем использовать модель с 1 сегментом данных и 1 сегментом кода
.data
     soob db 'hello !',’$’      ; выводимая строка
.code
main: mov ax,@data              ; в ax занесем адрес сегмента данных
     mov ds,ax                  ; загрузим адрес сегмента данных в ds
     mov dx,offset soob         ; в dx поместим адрес строки soob
     mov ah,09h                 ; вызов функции
     int 21h                    ; вывода строки
     int 20h                    ; выход
.stack 100h
end main
```

Как показано в примерах 2.7 и 2.8, сегментный регистр ***ds*** инициализирован явным способом. Для того чтобы объяснить необходимость этого действия, рассмотрим образ программы формата ****.exe*** в памяти после загрузки. Образ программы ****.exe*** в памяти имеет структуру, приведенную на рисунке 2.1. В регистр ***ip*** загружается адрес точки входа в программу, вычисляемый по метке в операторе ***END*** или адрес начала кодового сегмента, в ***sp*** - смещение конца сегмента стека. ***cs*** указывает на сегмент кода, ***ss*** - на сегмент стека, ***ds*** и ***es*** указывают на ***PSP***. Таким образом, сегмент данных оказывается не адресованным. Поэтому для получения доступа к данным надо инициализировать ***ds*** следующим образом:

```
mov ax,data
mov ds,ax
```

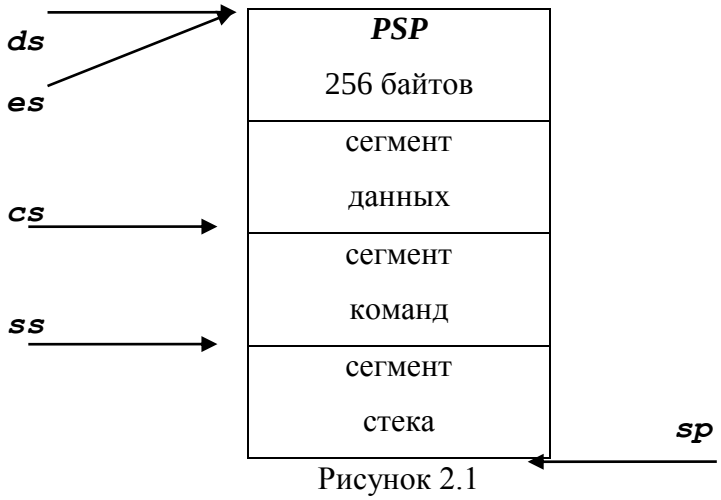


Рисунок 2.1

В начале программы располагается область префикса программного сегмента ***PSP*** (*Program Segment Prefics*) размером 256 байтов, заполняемая операционной системой при загрузке программы в память. ***PSP*** содержит специальные таблицы и данные, нужные в процессе выполнения программы. В программах формата ****.exe*** место под PSP не резервируется программистом, префикс пристраивается к программе в процессе ее загрузки

Программа типа ****.com*** находится в одном сегменте, включающем программу, данные и стек. При написании программ типа ****.com*** нужно помнить о необходимости резервирования места под ***PSP***. Это осуществляется оператором **`org 100h`**. Образ программы в формате ***.com** имеет структуру, отраженную на рисунке 2.2.

Все сегментные регистры указывают на ***PSP***. Следует иметь в виду, что регистр ***ip*** всегда инициализируется числом ***100h***, поэтому после оператора **`org 100h`** нужно ставить первую выполнимую строку программы. Программы формата **`*.com`** желательно писать так, чтобы данные определялись раньше кодов. Поэтому, если в начале программы размещены

данные, то необходимо их обойти, поставив переход на новую точку входа вида **jmp <имя точки входа>.**

Рисунок 2.2

Пример 2.9.

```
.model tiny
myseg segment para public 'code'
      assume cs:myseg,ds:myseg,ss:myseg,es:myseg
      org 100h                    ;резерв под PSP
start:jmp main                    ;обойдем данные, передав управление
      soob db 'hello!','$'        ; выводимая строка
main: mov dx,offset soob          ; в dx поместим адрес строки soob
      mov ah,09h                  ; вывод
      int 21h                     ; строки
      int 20h                     ; выход
myseg ends
end start
```

Так как в программах типа ***.com** стек находится в том же сегменте, что и код и данные, и растет при этом вверх, в область младших адресов, то существует опасность затирания стеком кода программы.

2.6 ЭТАПЫ РАЗРАБОТКИ ПРОГРАММ НА АССЕМБЛЕРЕ

Для учебных целей будем использовать простейший 16-разрядный компилятор и редактор связей. В TASM нет интегрированной среды разработки программ, поэтому для выполнения всех действий по вводу кода программы, ее трансляции, редактированию и отладке необходимо использовать отдельные служебные программы.

Для ввода кода программы можно использовать любой текстовый редактор, удовлетворяющий требованию, чтобы при наборе не вставлялись специальные символы. Создаваемый файл должен иметь расширение **.asm**. Процесс разработки программы на Ассемблере отражен на рисунке 2.3.

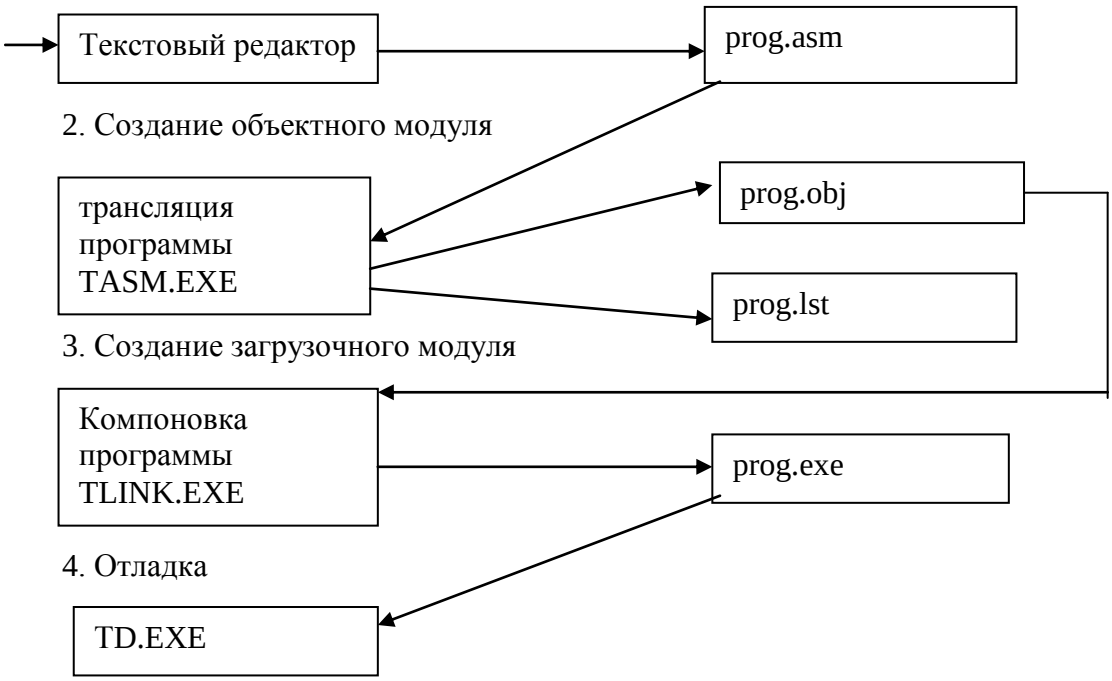


Рисунок 2.3 - Процесс разработки программ на ассемблере

Следующим шагом после подготовки текста программы является его трансляция, результатом которой служит формирование файла с расширением .obj - объектного модуля. Объектный модуль включает в себя представление исходной программы в машинных кодах, а также информацию, необходимую для компоновки его с другими модулями и отладки. Для создания объектного файла нужно из командной строки запустить транслятор tasm.exe командой

tasm.exe [опции] имя_исходного_файла [, имя_объектного файла]
[, имя_файла_листинга] [, имя_файла перекрестных ссылок]

Как видно из формата вызова транслятора, обязательным является только указание файла с исходным модулем. За именем исходного файла через запятую могут следовать имена объектного файла, файла листинга и файла перекрестных ссылок. Если эти имена не указать, то соответствующие файлы не будут созданы. Особенно полезным является создание файла листинга, имеющего расширение lst. Файл листинга содержит: код ассемблера исходной программы, машинный код каждой команды и ее смещение в кодовом сегменте, а также информацию о метках и сегментах, используемых в программе. При этом

сообщения об ошибках включаются в листинг непосредственно после ошибочной строки, что очень удобно для программиста.

В результате работы транслятора выдаются ошибки в программе (ERROR) и предупреждения (WARNINGS). В строке с ERROR указывается номер ошибочной строки с точки зрения синтаксиса ассемблера, а наличие строки с предупреждением означает, что конструкция синтаксически правильна, но не соответствует некоторым соглашением языка и может послужить источником последующих ошибок. Основные сообщения, выдаваемые транслятором TASM, приведены в приложении А. Запомнив номера строк с ошибками и предупреждениями, можно посмотреть эти строки в файле с расширением asm. О нормальном окончании процесса трансляции можно судить по отсутствию строк с ошибками и созданию объектного файла.

После получения объектного модуля следующим шагом является создание исполняемого (загрузочного) модуля. Этот шаг называется компоновкой программы, главной целью которой - преобразование кода и данных объектного файла в перемещаемое выполняемое отображение. Формат объектного файла позволяет объединить несколько отдельно оттранслированных исходных модулей в один модуль. При этом программа-компоновщик разрешает внешние ссылки на процедуры и переменные в этих внешних модулях. Результатом работы компоновщика является создание загрузочного файла с расширением .exe. Для создания файла .exe можно воспользоваться компоновщиком TLINK:

tlink <список имен объектных модулей с расширителями obj>

Имена компонуемых объектных модулей должны быть разделены пробелами или знаками “+”. Для создания исполняемого файла в формате .com необходимо указать компоновщику опцию t:

tlink /t <список имен объектных модулей с расширителями obj>

Обязательным этапом процесса разработки программы является ее отладка. Специфика программы на ассемблере состоит в том, что она интенсивно работает с аппаратными ресурсами компьютера, поэтому необходимо постоянно отслеживать содержимое определенных регистров и областей памяти. Для тестирования программ на ассемблере используют специальные отладчики, например, Turbo Debugger (td.exe), позволяющий выполнять трассирование программы и просмотр состояния регистров и областей памяти. Недостатком td.exe является то, что он не позволяет вносить исправления в исходный текст программы. Запуск программы-отладчика осуществляется по команде

td.exe <имя исполняемого файла>

2.7 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как описываются данные в языке Ассемблер?
2. Обработка элементов структуры.
3. Перечислите ограничения на операнды в команде **mov.**
4. Как выполняется команда знакового умножения?
5. Типы условных переходов и примеры их применения.
6. Как организуется цикл со счетчиком?
7. Организация передачи управления и параметров в подпрограмму.
8. Как описываются сегменты ?
9. Перечислите этапы разработки программы на Ассемблере.

2.8 УПРАЖНЕНИЯ

Задание. *Дана квадратная матрица A. Сформировать массив B из элементов ее главной диагонали.*

```
prg segment para public 'code'; заголовок сегмента
assume cs:prg,ds:prg,es:prg,ss:prg
;сообщим транслятору о намерении установить сегментные
; регистры  cs, ds, es на наш сегмент
org 100h                        ; место под PSP
beg: jmp start                  ; переход на start
mess1 db 0ah,0dh,'Введите размерность : $'   ; приглашение для
; ввода размерности. Коды 0ah и 0dh означают переход на начало
; следующей строки
mess2 db 0ah,0dh,'Введите элементы A: $'     ;приглашение для
; ввода элементов матрицы
mess3 db 0ah,0dh,'Элементы B: ',0ah,0dh,'$'
;заголовок для вывода элементов массива
answer db 7 dup(?),'$'          ; буфер под ответ
bufer db 0ah,0,11 dup(?)        ; буфер для ввода
n dw ?                          ; размерность матрицы
a dw 25 dup(?)                  ; память под матрицу
b dw 5 dup(?)                   ; память под массив
newline db 0ah,0dh,'$'          ; для перевода строки
include bin2str.asm
; подключение процедуры bin2str, находящейся в файле
; bin2str.asm и осуществляющей преобразование числа в строку
; вход bin2str – ax – число;
```

```
; выход - строка по адресу ds:bx
include str2bin.asm
; подключение процедуры str2bin, находящейся в файле
;str2bin.asm и осуществляющей преобразование из строки в число
; вход str2bin - ds:bx   - адрес строки
; выход - ax - число со знаком

; input – подпрограмма для ввода одного числа
; Вход: в dx - адрес буфера,
; Выход: в ax возвращается введенное число
   input proc
         push bx              ;сохраним используемые регистры
         push dx
         push dx              ;сохраним адрес буфера, т.к. он
; будет запорчен при организации вывода строки newline
         lea dx,newline
         mov ah,09h
         int 21h              ;вывод строки
         pop dx               ; восстановим адрес буфера ввода
         mov ah,0ah           ;помещаем в ah номер прерывания
         int 21h              ;для ввода строки
         mov bx,dx            ;строка по адресу ds:dx
         inc bx               ;увеличиваем на 1 ее адрес
; и передаем его в процедуру str2bin
         call str2bin         ;преобразуем строку в число
         pop dx               ;число будет в ax
         pop bx
         ret
   input endp
; Процедура вывода числа, находящегося в ax
   output proc
          push bx           ;сохраним используемый регистр
          lea bx,answer     ;адрес буфера для преобразования
          call bin2str      ;преобразуем число в строку
          lea dx,answer+1   ;помещаем в dx адрес начала строки
          mov ah,09h        ;помещаем в ah номер прерывания
          int 21h             ;вывода строки
          lea dx,newline      ;переход на новую строку
          mov ah,09h
          int 21h
          pop bx              ;восстановление bx
```

```
        ret
 output endp
 start: lea dx,mess1;Загружаем в dx адрес строки для ввода N
        mov ah,09h   ;функция 09h – реализует вывод строки
        int 21h      ;вызываем прерывание int 21h
        lea dx,bufer ;загружаем в dx адрес буфера для ввода
        call input   ;вызываем input
        mov n,ax     ;сохраним введенное число в N
        lea dx,mess2;Загружаем в dx адрес строки для ввода A
        mov ah,09h
        int 21h      ;и выводим сообщение
        mov cx,n     ;в cx запишем счетчик цикла по строкам
        lea bx,a    ;в bx запишем адрес начала матрицы A
        mov si,0     ;смещение от начала матрицы равно нулю
  v1:   push cx     ;перед организацией цикла сохраним
; счетчик по строкам
        mov cx,n    ; в cx запишем счетчик цикла по столбцам
  v2:   push cx      ; сохраним счетчик внутреннего цикла
        lea dx,bufer; в dx адрес буфера для ввода
        call input  ; введем одно число (вернется в ax)
        mov [bx][si],ax  ;и запишем его в матрицу A по
; адресу (bx)+(si) с помощью базово-индексной адресации
        add si,2         ;переход к следующему элементу
        pop cx           ;восстановим счетчик цикла
        loop v2 ;продолжим цикл ввода элементов одной строки
        pop cx           ;восстановим счетчик внешнего цикла
        loop v1          ;перейдем к следующей строке
        lea bx,a         ;в bx адрес начала матрицы A
        lea bp,b         ;в bp адрес начала массива B
        mov di,0         ;смещение от начала массива 0
        mov si,0         ;смещение от начала матрицы 0
        mov cx,n       ;организуем цикл по строкам матрицы A
  c1:   mov ax,[bx][si]   ;в ax запишем элемент из матрицы
; по адресу (bx)+(si)
        mov [bp][di],ax   ;и переместим его в массив B
        add di,2   ;переход к следующему элементу массива B
        add si,2          ;переход к следующему элементу
        add si,n          ;главной диагонали
        add si,n          ;матрицы A
        loop c1
        lea dx,mess3      ;Загружаем в dx адрес сообщения
```

```
          mov ah,09h          ; для вывода B
          int 21h
          mov cx,n             ;в cx счетчик цикла по выводу
; элементов массива
          lea bx,b            ;адрес начала массива в bx
          mov si,0            ;смещение от начала массива = нулю
  v3:     push cx             ;сохраним счетчик цикла
          mov ax,[bx][si]     ;очередное число в ax
          call output          ;вызов подпрограммы вывода
          add si,2            ;переход к следующему элементу
          pop cx              ;восстановим счетчик цикла
          loop v3             ;к началу цикла
          int 20h             ;завершить работу
   prg    ends                ;конец сегмента
          end beg              ;точка входа - beg
```

Варианты:

1. В массиве из N элементов найти сумму элементов, больших заданного числа K.

2. Дана матрица размерности NxN, состоящая из целых чисел. Найти сумму элементов главной диагонали этой матрицы.

3. Дана матрица размерности NxN, состоящая из целых чисел. Найти количество элементов в матрице, значения которых принадлежат заданному отрезку [a, b].

4. Дана матрица размерности NxM, состоящая из целых чисел. Вычислить сумму элементов в четных строках.

5. Дан массив из N элементов. Проверить, является ли он возрастающей последовательностью.

6. Дана матрица размерности NxM, состоящая из целых чисел. Проверить, есть ли в ней элементы, равные 0.

7. Дан массив из N элементов. Посчитать, сколько элементов массива могут храниться в байте.

8. Дана матрица размерности NxN, состоящая из целых чисел. Выдать на экран номера строк, где элементы образуют убывающую последовательность.

9. Сформировать матрицу размерности NxM, записав в элемент a_{ij} число, равное i, если i>j, и число j, если это условие неверно.

10. Дана матрица размерности NxM, состоящая из целых чисел. Найти сумму элементов первого и последнего столбцов.

11. Даны два массива A и B размерности N. Сформировать третий массив C, каждый элемент которого $C_{ij} = \max(A_{ij}, B_{ij})$.

12. Дана матрица размерности NxN, состоящая из целых чисел. Для каждой строки этой матрицы выдать на экран минимальный элемент.

13. Дана матрица размерности NxM, состоящая из целых чисел. Выдать номер строки и номер столбца для элементов, равных заданному числу K.

14. Дан массив A из N элементов. Сформировать массив B, записав в него элементы массива A в обратном порядке.

15. Дана матрица размерности NxM, состоящая из целых чисел. Для каждого столбца матрицы выдать на экран максимальный элемент.

2.9 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Обращение к элементу структуры на Ассемблере осуществляется:
   а) по имени переменной типа «структура» и порядковому номеру поля;
   б) по уточненному имени, включающему имя переменной типа «структура» и имя элемента структуры из ее описания, разделенные точкой;
   в) по уникальному имени поля структуры.
2. В команде **`mov ax,[bx+6]`** используется один из следующих видов адресации:
   а) прямая;
   б) базово-индексная;
   в) базовая со смещением;
   г) базово-индексная со смещением.
3. Доступ к аргументам в подпрограмме через стек предусматривает:
   а) перемещение аргументов из стека во вспомогательную область памяти;
   б) использование для стека базовой адресации со смещением через регистр bp как указателя на базу стека;
   в) загрузку аргументов из стека в специальный сегмент памяти, доступный вызывающему и вызываемому модулям.
4. Сегментные регистры после загрузки программы в формате *.exe в память указывают на следующие объекты:
   а) ***`cs`*** - на сегмент кода, ***`ds`*** и ***`ss`*** - на ***PSP***, ***`es`*** - на сегмент данных
   б) ***`cs, es, ds, ss`*** - указывают на ***PSP***
   в) ***`cs`*** - на сегмент кода, ***`ss`*** - на сегмент стека, ***`ds`*** и ***`es`*** - на ***PSP***
   г) ***`cs, ds, es`*** - на ***PSP***, ***`ss`*** на сегмент стека
   д) ***`ss`*** - на сегмент стека, ***`ds`*** и ***`cs`*** - на ***PSP***, ***`es`*** - на сегмент данных

Ответы: 1 - б; 2 - в; 3 - б; 4 - в.

3 МАКРОСРЕДСТВА

Иногда при написании программы на Ассемблере оказывается, что в ней встречаются одинаковые или похожие куски, различающиеся мнемониками регистров, портов, выражениями в операторах и т.п. Также часто бывает необходимо при отладке программы включать или не включать в программу некоторые части текста исходного модуля для проверки правильности их работы.

Для помощи программисту в подобных ситуациях в Ассемблере реализованы макросредства.

3.1 ОСНОВНЫЕ ПОНЯТИЯ О МАКРОГЕНЕРАЦИИ

При использовании макросредств в программе на Ассемблере ее трансляция осуществляется в 2 фазы (рисунок 3.1). В первой фазе происходит специальная обработка текста программы, заключающаяся в выполнении подстановки вместо одних частей текста, описанных специальным образом, других его частей.

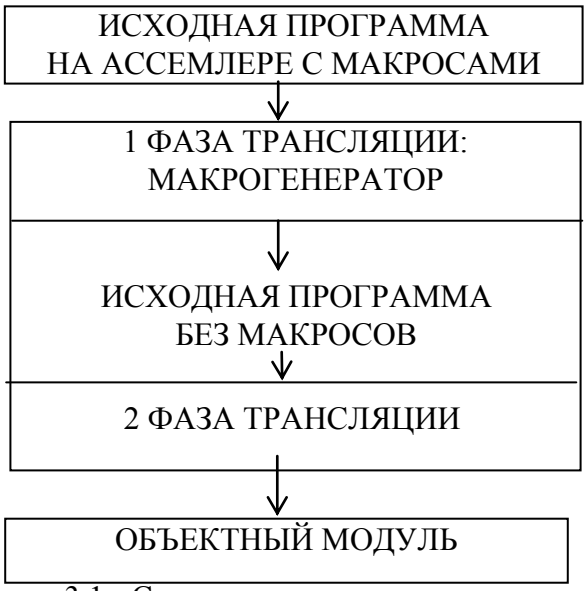


Рисунок 3.1 - Схема трансляции программы на Ассемблере

Например, если в программе несколько раз повторяется неизменяемая часть текста, то эту часть программы можно оформить специальным образом и дать ей имя, не совпадающее ни с какой мнемоникой команды на Ассемблере, после чего в программе вместо повторяющейся части текста можно указать присвоенное ей имя. Понятно, что транслировать такую программу нельзя до тех пор, пока в ней вместо выбранного нами

имени части программы не появится сама эта часть. Для того, чтобы в программе не было строк текста, которые не понимает транслятор, перед передачей управления собственно транслятору необходимо выполнить подстановку вместо имени части программы соответствующий ей текст на Ассемблере. Этой процесс называется макрогенерацией.

Во второй фазе трансляции, после макрогенерации, происходит собственно формирование объектного кода, содержащего оттранслированный текст исходной программы в машинные коды.

К простейшим имеющимся в Ассемблере макросредствам относятся псевдооператоры **equ** и **=**. Они предназначены для присвоения некоторому идентификатору строки или числового выражения.

Псевдооператор **equ** имеет следующий синтаксис:

идентификатор equ выражение

Синтаксис псевдооператора "**=**" практически не отличается от **equ**, однако в качестве присваиваемого выражения может быть использовано только числовое выражение. Другое отличие рассматриваемых псевдооператоров заключается в том, что идентификаторы, определенные с помощью оператора "**=**", можно заново переопределить в тексте программы, а идентификаторы, определенные с использованием ***equ***, переопределять нельзя.

Псевдооператор "**=**" удобно использовать для определения выражений, которые не зависят от места загрузки программы в память и могут быть вычислены.

Примеры:

```
adres1    dw 0
adres2    dw ?
b    egu 40h
len1=15
len2=len+1
len3=adr2-adr1
```

Как видно из примеров, в правой части псевдооператора "**=**" можно использовать метки и ссылки на адреса.

Рассмотрим более сложные макросредства Ассемблера. Если в программе есть повторяющийся с различными вариациями фрагмент текста, то его обычно оформляют в виде макроопределения. Описание макроопределения имеет вид:

```
<имя макроопределения> MACRO [<список формальных параметров>]
        <тело макроопределения>
        ENDM
```

В списке формальных параметров через запятую перечисляются имена формальных параметров. Имя формального параметра представляет собой изменяемое при макрорасширении имя.

Рассмотрим простейший вариант описания макроопределения. Пусть надо вычислить и поместить в регистр ***cx*** число, равное ***max(A,B)+max(A,D)***, причем ***A,B,D*** - переменные. Будем считать, что вычисление максимума из двух чисел оформлено процедурой ***max***, причем для процедуры ***max*** первый параметр передается через регистр ***ax***, второй - через ***bx***, результат вычислений помещается в ***ax***.

Тогда фрагмент программы, выполняющий требуемые вычисления может выглядеть следующим образом:

```
xor   cx,cx        ; занулим регистр cx
mov   ax,A         ; поместим в регистр ax число A
mov   bx,B         ; поместим в регистр bx число B
call  max          ; обратимся к процедуре вычисления
add   cx,ax        ; добавим вычисленный максимум к cx
mov   ax,A         ; поместим в регистр ax число A
mov   bx,D         ; поместим в регистр bx число D
call  max          ; обратимся к процедуре вычисления
add   cx,ax        ; добавим вычисленный максимум к cx
```

Опишем макроопределение с именем ***V_MAX***:

```
V_MAX MACRO X, Y
    mov ax, X
    mov bx, Y
    call max
    add cx,ax
endm
```

Для того чтобы воспользоваться этим макроопределением, нужно указать макрогенератору на необходимость подстановки тела макроопределения в текст программы с помощью *макровызова*.

Синтаксис макровызова имеет вид:

<имя макроопределения> <список фактических параметров>

Для нашего примера :

```
xor    cx,cx
V_MAX  A,B
V_MAX  A,D
```

Когда макрогенератор встречает макровызов, он находит макроопределение с указанным именем, в тело макроопределения вместо формальных параметров подставляет фактические в том же порядке и полученный текст (так называемое макрорасширение)

вставляет в текст вместо макровызова. Каждый фактический параметр представляет собой строку символов.

Возможна ситуация, когда в макроопределении формальный параметр трудно выделить из текста, например, если он представляет собой часть некоторого идентификатора. В этом случае имя формального параметра в теле макроопределения должно быть отделено от остальных символов идентификатора знаком **&**. Назначение знака **&** - указать границу формального параметра, выделить его из окружающего текста, поэтому формальный параметр может быть выделен знаком **&** с двух сторон. Этот прием часто используется для модификации идентификаторов и мнемоник команд. Например, рассмотрим макроопределение, позволяющее создавать в программе описание матриц различного типа и размерности:

```
dmatr   macro     name,typ,len
   name d&typ     len DUP (len DUP(0))
endm
```

Если в тексте программы указать макровызовы

```
dmatr a,w,6
dmatr b,b,10
dmatr c,q,3
```

то макрорасширение будет иметь вид:

```
a dw 6 DUP(6 DUP(0))
b db 10 DUP(10 DUP(0))
c dq 3 DUP(3 DUP(0))
```

Еще одна особенность использования знака **&** в теле макроопределения заключается в том, что он может быть использован при вложенной макрогенерации. Если рядом стоят два знака **&,** то при первом проходе макрогенератора из текста удаляется только один из них, а при втором проходе удаляется оставшийся знак **&.**

Макроопределения можно размещать либо в начале текста программы, либо в отдельном файле. В первом случае описанные макроопределения можно использовать только в той программе, где они находятся. Во втором случае макроопределения можно использовать во многих программах, но для этого необходимо в начале текста программы, где есть вызовы соответствующих макроопределений, написать директиву

```
include <имя файла>
```

где файл с именем <имя файла> содержит тексты макроопределений. Содержимое этого файла вставляется в исходный текст программы перед началом компиляции.

Функционально макроопределения похожи на процедуры. Сходство заключается в том, что макроопределения и процедуры сначала описываются один раз, а затем используются. Однако существуют следующие отличия:

1) Текст процедуры неизменен, а текст макроопределения меняется в зависимости от фактических параметров, причем могут меняться не только операнды, но и команды.

2) При использовании макроопределений скорость работы выше, а объем программы больше за счет многократной вставки тел макроопределений. Для процедур скорость работы меньше за счет времени, затрачиваемого на организацию передачи управления.

В том случае, когда в теле макроопределения встречаются метки или описываются данные, в процессе макрогенерации может возникнуть ситуация, когда в программе один идентификатор будет определен несколько раз, что будет распознано транслятором как ошибка. Для исключения дублирования меток при макрогенерации используется директива ***LOCAL***, имеющая синтаксис:

```
LOCAL <список идентификаторов>
```

Директива ***LOCAL*** пишется сразу после заголовка макроопределения. При использовании этой директивы в каждом экземпляре макрорасширения для всех идентификаторов, перечисленных в списке, генерируются уникальные имена.

3.2 ЦИКЛИЧЕСКАЯ И УСЛОВНАЯ МАКРОГЕНЕРАЦИЯ

С помощью макросредств ассемблера можно модифицировать набор строк текста программы, входящий в макрорасширение. Для этого можно использовать циклическую и условную макрогенерацию.

Циклическая макрогенерация реализуется с помощью директив ***WHILE, REPT, IRP, IRPC.***

Директивы ***WHILE*** и ***REPT*** применяют для повторения определенного количества раз в тексте программы некоторой последовательности строк. Синтаксис директив ***WHILE*** и ***REPT*** похож и имеет вид:

```
<имя директивы> выражение
последовательность строк
ENDM
```

где <имя директивы> - это ***WHILE*** или ***REPT***, а выражение должно быть вычислимо на этапе трансляции.

При использовании директивы ***WHILE*** макрогенератор будет повторять указанную в макроопределении последовательность строк до тех пор, пока значение выражения не станет равно нулю. Из этого следует, что последовательность строк должна изменять значение

выражения. Значение выражения вычисляется и проверяется на равенство нулю каждый раз перед очередной итерацией процесса макрогенерации.

В директиве ***REPT*** последовательность операторов повторяется столько раз, сколько указано в выражении, при этом значение выражения автоматически уменьшается на единицу после каждой итерации макрогенерации.

Например:

```
rept 3
shr  ax,1
endm
```

В блоках повторения может использоваться директива присваивания. Например, рассмотрим получение таблицы квадратов чисел от 1 до 10:

```
X=1
rept 10
dw   X*X
X=X+1
endm
```

В результате макрогенерации получим строки текста программы

```
dw   1
dw   4
dw   9
...
dw   100
```

Таким образом, директивы ***WHILE*** и ***REPT*** дают возможность многократного копирования некоторой последовательности строк. Для выполнения модификации дублируемых строк используются директивы ***IRP*** и ***IRPC***. Директива ***IRP*** имеет следующий синтаксис:

```
IRP формальный_параметр, <строка_1, . . . ,строка_n>
<тело>
ENDM
```

Директива ***IRP*** создает ***n*** копий тела макроопределения, причем в первой копии формальный_параметр заменяется на ***строку_1***, во второй копии - на ***строку_2*** и т.д. Например, рассмотрим построение таблицы квадратов простых чисел, меньших 10.

```
irp X, <2, 3, 5, 7>
  dw x*x
endm
```

Директива ***IRPC*** имеет синтаксис:

```
IRPC формальный_параметр, строка_символов
<тело>
```

```
ENDM
```

Действие директивы подобно ***IRP***, отличие заключается в выполнении на каждой итерации подстановки вместо формального параметра очередного символа из указанной в директиве строки символов. Количество повторений в данном случае будет определяться количеством символов в ***строке_символов***. Например:

```
IRPC    r,<abcd>
push    r&x
ENDM
```

Результатом макрогенерации будут операторы, последовательно помещающие в стек содержимое регистров ***ax***, ***bx***, ***cx***, ***dx***.

Фактические параметры директивы ***IRP*** не должны содержать запятые, точки с запятой и угловые скобки, а в ***строке_символов*** директивы ***IRPC*** нельзя указывать пробелы и точки с запятой. Однако эти правила можно нарушить. Для этого необходимо заключить фактический параметр или ***строку_символов*** в угловые скобки:

```
IRPnum,<<4,5>,7>
db num
ENDM
IRPC    T,<a;d>
db '&T'
ENDM
```

При этом будет сгенерировано следующее макрорасширение:

```
db 4,5
db 7
db 'a'
db ';'
db 'd'
```

Условная макрогенерация позволяет выполнить анализ некоторых условий в процессе генерации макрорасширения и, в зависимости от этих условий, изменить ход макрогенерации. Например, возможно включить в макрорасширение не все строки тела макроопределения, а выбрать один вариант участка текста программы из нескольких имеющихся в зависимости от некоторых условий. Какие конкретно условия должны быть проверены, определяется типом условной директивы, объединяемых в зависимости от назначения в 4 группы.

К директивам первой группы относятся ***IF*** и ***IFE***, имеющих синтаксис:

```
IF(E)   логическое выражение
   текст_1
ELSE
   текст_2
```

```
ENDIF
```

Работают директивы ***IF*** и ***IFE*** следующим образом. Сначала вычисляется значение логического выражения, а это значит, что в нем нельзя сослаться на величины, которые станут известны только на этапе выполнения. Выражение считается истинным, если его значение не равно нулю, и ложным, если значение выражение - ноль. Если выражение истинно, то директива ***IF*** помещает в программу ***текст_1***, а директива ***IFE*** - ***текст_2***. Если выражение ложно, то директива ***IF*** помещает в программу ***текст_2***, а директива ***IFE*** - ***текст_1***.

Ключевое слово ***ELSE*** и ***текст_2*** в директивах могут отсутствовать. В этом случае для директивы ***IF*** в случае ложности логического выражения ***текст_1*** игнорируется и в программу ничего не помещается.

Рассмотрим пример применения условной макрогенерации для вставки в программу в зависимости от значения переменной ***FLAG*** различных директив описания данных:

```
FLAG EQU 1
.....
IF FLAG
Tabl Db 100 DUP(0)
ELSE
Tabl DW 100 DUP(0)
UPACK DB 150 DUP(0)
ENDIF
```

Директивы ***IF*** и ***IFE*** целесообразно использовать при отладке программ для выполнения вывода на экран диагностической информации. Можно определить в программе некоторую переменную, например, ***DEBUG***, и использовать ее с директивами условной макрогенерации по результату вычисления логического выражения:

```
DEBUG   equ 1
   ........
IF DEBUG
<команды вывода на экран отладочной информации>
ENDIF
```

Директивы 2, 3, 4 групп позволяют отслеживать наличие в программе определения символического имени (***IFDEF,IFNDEF***), наличие фактического параметра в макровызове (***IFB,IFNB***), для проверки совпадения фактического параметра с заданной строкой (***IFIDN,IFIDNI,IFDIF,IFDIFI***). Подробное описание этих директив можно найти в [11].

С директивами условной макрогенерации могут быть использованы директивы управления макрогенерацией:

- ***EXITM*** - прекращает процесс генерации макрорасширения;

■ ***GOTO метка*** - служит для прекращения генерации макрорасширения и перехода на другой участок программы. При этом перед именем метки в программе ставится двоеточие.

3.3 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. На каком этапе получения готовой для выполнения программы из файла, содержащего исходный модуль, применяются макросредства ?
2. Что такое макроопределение и макровызов ?
3. Как описываются макроопределения ?
4. Поясните отличия макроопределений от процедур.
5. Как формируется макрорасширение ?
6. Поясните назначение директивы EQU.
7. Как исключить дублирование меток при макрогенерации ?
8. Поясните понятие циклической макрогенерации. Какие директивы циклической макрогенерации Вам известны ?
9. Поясните назначение и синтаксис директив IF и IFE.
10. Для каких целей удобно использовать условную макрогенерацию?

3.4 УПРАЖНЕНИЯ

Задание. Даны 5 массивов. Для каждого из них проверить наличие двух идущих подряд нулевых элементов и выдать соответствующее сообщение.

```
OutputFlag equ 1    ; Флаг ввода массивов

ITR equ 5           ; Количество элементов
code segment   para public 'code'  ; Начало сегмента кода
     assume cs:code, ds:code, ss:code
org 100h             ; резервирование места под PSP
prg: jmp start
     mas1 dw 1,2,3,4,5    ;Резервируем память под массивы
     mas2 dw 5,0,0,5,0
     mas3 dw 5,5,5,5,0
     mas4 dw 0,0,0,0,0
     mas5 dw 1,0,1,0,1
     input_buf db 06,00,5 dup(?)  ;Память под буфер ввода
     messin db 0ah,0dh,'Введите элемент массива:$'
```

```
      messmasP1 db 0ah,0dh,'Ввод элементов $' ;Строки сообщений
      messmasP2 db '-го массива:$'
      messMASSIV db 0ah,0dh,'Массив:$'
    messOK db 0ah,0dh,'В массиве есть 2 идущих подряд нулевых элемента!$'
      messNOT db 0ah,0dh,'В массиве нет 2 нулевых элементов рядом!$'
      messNUM db 7 dup(?),' ','$'   ; Память под буфер вывода
include str2bin.asm
include bin2str.asm
;  Макроопределение для вывода строки на экран.
OutPutMess MACRO message
             lea dx,message  ; Адрес строки -> в dx
             mov ah,09h      ; Номер функции вывода
             int 21h
           ENDM
; Макроопределение для организации ввода элементов массива
; с клавиатуры. Параметры – имя массива и количество элементов
Input_mas MACRO mas,amnt
        LOCAL metka1
        lea bx,mas
        push bx
        mov cx,amnt
metka1: lea dx,messin  ; Вывод приглашения на ввод элемента
        mov ah,09h
        int 21h
        lea dx,input_buf ; Организация непосредственного ввода
        mov ah,0ah
        int 21h
        mov bx,dx
        inc bx
        call str2bin
        pop bx            ; Запись считанного числа в массив
        mov word ptr[bx],ax
        add bx,2
        push bx
        loop metka1
      ENDM
;  Макроопределение для проверки массива на наличие
;  2-х идущих подряд нулевых элементов. Параметры – имя
;  массива и количество элементов.
MasView MACRO mas,amnt
        LOCAL cycl,m1,m2,m3
        mov cx,amnt
        lea bx,mas
        mov ax,bx
```

```
        add ax,2*amnt-2
cycl:   cmp word ptr[bx],0   ; сравним очередной элемент с 0
        jne m1 ;если не ноль, то переход к следующему элементу
        cmp bx,ax            ; обработан последний элемент?
        je m                 ; если да, то двух подряд нулей
; не нашли, и переходим на m, чтобы сообщить об этом и выйти
        cmp word ptr[bx+2],0 ; следующий элемент – ноль?
        jne m1               ; если нет, то на m1
        OutPutMess messOK    ; если да, то сообщим о находке
        jmp m2               ; и выйдем из цикла
m1:     add bx,2             ; перейдем к следующему элементу
        loop cycl            ; продолжим цикл
m3:     OutPutMess messNOT   ; вывод сообщения о неудаче
; поиска двух подряд идущих нулей
m2:     nop             ; пустой оператор для выхода из цикла
       ENDM
;  Основная программа
start:
       IF OutputFlag           ; Ввод массивов самостоятельно
         IRPC num,<12345>
            OutPutMess messmasP1
            mov ax,&num
            lea bx,messNUM
            call bin2str
            lea dx,messNUM+6
            mov ah,09h
            int 21h
            OutPutMess messmasP2
            Input_mas mas&num,5
            masview mas&num,5
         ENDM
       ELSE                    ; Для проверки используются
         IRPC num,<12345>      ; стандартные массивы
            OutPutMess messMASSIV
            lea bx,mas&num
            REPT ITR
              mov ax,word ptr[bx]
              push bx
              lea bx,messNUM
              call bin2str
              lea dx,messNUM+6
              mov ah,09h
              int 21h
              pop bx
```

```
                add bx,2
                ENDM
                masview mas&num,5
          ENDM
        ENDIF
      int  20h  ;завершение работы программы
code ends       ;конец сегмента кода
end prg
```

Варианты заданий :

1. Даны три массива. В каждом из них найти сумму элементов, больших заданного числа К. В качестве отладочной информации выдавать промежуточные значения суммы и индексы суммируемых элементов.

2. Даны три квадратных матрицы, состоящие из целых чисел. Найти сумму элементов главной диагонали каждой из этих матриц. В качестве отладочной информации вывести значения элементов главной диагонали.

3. Даны три квадратных матрицы, состоящие из целых чисел. В каждой из матриц найти количество элементов, значения которых принадлежат заданному отрезку [a, b]. В качестве отладочной информации для каждого элемента каждой матрицы выводить его значение и строку “принадлежит” или “не принадлежит” в зависимости от факта нахождения элемента на отрезке [a,b].

4. Даны две матрицы, состоящие из целых чисел. Для каждой матрицы вычислить сумму элементов в четных строках. В качестве отладочной информации выдавать на экран суммируемые элементы и их индексы.

5. Даны три массива. Проверить, является ли они возрастающими последовательностями. В качестве отладочной информации выдавать значения элементов массивов до тех пор, пока они образуют возрастающую последовательность.

6. Даны три матрицы. Проверить, есть ли в них элементы, равные 0. В качестве отладочной информации выдавать значения и индексы просматриваемых элементов матриц.

7. Даны три массива, каждый из которых хранится в слове. Для каждого из массивов посчитать, сколько элементов могут храниться в байте. В качестве отладочной информации выдать номера и значения элементов, которые в байте поместиться не могут.

8. Даны три матрицы. Для каждой из них найти номера строк, где элементы образуют убывающую последовательность. В качестве отладочной информации выдать элементы из таких строк.

9. Даны три матрицы. Для каждой из них найти сумму элементов первого и последнего столбцов. В качестве отладочной информации выдать элементы из этих столбцов.

10. Даны два массива A и B размерности N. Сформировать третий массив C, каждый элемент которого $C_{ij} = \max(A_{ij}, B_{ij})$. В качестве отладочной информации выдать элементы A_{ij} и B_{ij}, а также значения i и j.

11. Даны три квадратных матрицы. Для каждой строки каждой матрицы найти минимальный элемент. При отладке выдавать промежуточные значения минимального элемента из каждой строки.

12. Даны три матрицы. Для каждой из них выдать номер строки и номер столбца для элементов, равных заданному числу K. При отладке выдавать значения всех элементов с указанием строк “равно” или “не равно” в зависимости от равенства элемента числу K.

13. Даны массивы A1 и A2. Сформировать массивы B1 и B2, записав в них элементы соответственно массивов A1 и A2 в обратном порядке. В качестве отладочной информации выдать индексы и значения переписываемых элементов из A1 и A2.

3.5 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Какие из приведенных ниже утверждений верны:

1) макрогенерация осуществляется перед компиляцией программы;

2) если в программе повторяется один и тот же текст несколько раз, то эта часть текста может быть оформлена в виде макроопределения, а для реального использования этого макроопределения в программе используется макровызов;

3) формальный параметр не может быть подстрокой в некоторой строке в теле макроопределения;

4) псевдооператоры ***equ*** и **=** не отличаются по синтаксису написания, выполняемым действиям и ограничениям на использование;

5) фактические параметры подставляются в тело макроопределения вместо формальных в том порядке, в каком они указаны в макровызове.

Из 1-5 верны: а - 2, 3; 5; б - 2, 5; в - 1, 2, 3, 5; г - 1, 2, 5; д - 1, 2, 4, 5.

2. Какое макрорасширение будет сгенерировано, если описано макроопределение

```
a     macro q1, q2, q3, q4
      mov q1&x, q3
      mov q2&x, q4
      call prog1
      endm
```

и в программе указан макровызов вида

```
a     b,c,3,1
```

Варианты ответов:

а)
```
mov q1,3
mov q2,1
call prog1
```

б)
```
mov ax,3
mov bx,c
call prog1
```

в)
```
mov bx,3
mov cx,1
call prog1
```

г)
```
mov cx,1
mov bx,3
call prog1
```

д)
```
mov bx,3
mov cx,3
call prog1
```

3. В результате работы макрогенератора макрорасширением фрагмента программы

```
irp  alfa,<5,8,2>
     mov es:[bx+alfa],05h
endm
```

будет следующий текст:

а)
```
mov es:[bx+5],05h
mov es:[bx+5],08h
mov es:[bx+5],02h
```

б)
```
mov es:[bx+5], 05h
```

в)
```
mov es:[bx+5],05h
mov es:[bx+8],05h
mov es:[bx+2],05h
```

г)
```
mov es:[bx+alfa],05h
```

д)
```
mov es:[bx+2],05h
```

4. Условная макрогенерация нужна для того, чтобы

а) управлять включением в исходный модуль различных частей текста программы в зависимости от некоторых условий;

б) программист мог один раз описать макроопределение, а затем использовать его многократно по мере необходимости;

в) при написании программы была возможность повторять нужный фрагмент программы несколько раз;

г) для генерации минимального по объему текста макрорасширения.

Ответы: 1 - г; 2 -в; 3 - в; 4 - а.

4 ПРЕРЫВАНИЯ

4.1 ОБЩИЕ ПОНЯТИЯ О ПРЕРЫВАНИЯХ

При обмене с внешними устройствами необходимо так организовать совместную работу процессора и внешнего устройства, чтобы устройство само оповещало процессор о своей готовности к пересылке данных. Такое оповещение выполняется с помощью специального сигнала, называемого прерыванием. Прерывание - это временное прекращение процессором вычислительного процесса, необходимое для выполнения действий сервисного характера, обслуживания запросов программы на выполнение функций операционной системы, реакции на нештатные ситуации в вычислительной системе. Программа, выполняемая в ответ на запрос прерывания, называется программой обработки прерываний.

В зависимости от источника прерывания классифицируются следующим образом:

■ аппаратные, возникающие как реакция на сигнал от внешних устройств (например, нажатие клавиши на клавиатуре, щелчок по кнопке мыши). Эти прерывания являются асинхронными, т.е. их возникновение не привязано к конкретному моменту времени, известному программисту.

■ программные, которые вызываются ядром ОС для выполнения необходимых действий.

В простейшем случае можно вызвать системную программу явной командой ***int.***

int <номер системной программы>

Однако такой вызов не является прерыванием, так как выполнение процессором текущего кода не прерывается.

■ исключения процессора. Например, процессор может прервать выполнение программы и в том случае, если в ходе реализации ее команд обнаружит ошибку или какую-либо нестандартную ситуацию, в частности, если заданный в команде код операции не соответствует ни одной их существующих команд, или арифметическая команда пытается выполнить деление на ноль. При возникновении исключений процессор приостанавливает выполнение текущей программы и запускает программу обработки исключения. Эта программа выполняет действия, необходимые для восстановления после ошибки (если это возможно), или информирует о ней пользователя. Другим примером исключений являются исключения защиты. Для защиты ОС компьютера от разрушения пользовательскими программами некоторые команды разрешено выполнять только тогда, когда программа имеет определенный уровень привилегий. Если в программе сделана попытка выполнить

привилегированную команду или обратиться к запрещенной области памяти, то возникает исключение по нарушению общей защиты.

Обработка прерываний в реальном и защищенном режимах осуществляется по-разному. Рассмотрим обработку прерываний в реальном режиме.

Все прерывания нумеруются от 0 до 255. Для обработки каждого прерывания в ОС имеется своя процедура обработки прерывания (ПОП), 4-х-байтовый адрес которой (сегмент и смещение) записывается в специальную таблицу адресов, называемую вектором прерываний. Вектор прерываний - это 256-элементный массив, в котором i-ый элемент - адрес ПОП с номером i. Вектор прерываний занимает 1 Кб.

Программа обработки прерывания очень похожа на обычную подпрограмму. Однако между ними имеются очень важные различия. Подпрограмма выполняет функцию, необходимую той программе, из которой она вызвана, тогда как ПОП может не иметь ничего общего с выполняемой программой. Таким образом, перед вызовом ПОП необходимо сохранить всю информацию, которая может быть изменена в ходе ее выполнения. Перед выходом из ПОП эта информация должна быть восстановлена. К числу сохраняемой и восстанавливаемой информации обычно относятся значения регистра флагов, счетчика команд и содержимое всех тех регистров, которые используются и прерванной программой, и ПОП. Задача сохранения и восстановления информации может автоматически выполняться процессором или же командами программы. Большинство современных процессоров сохраняют только минимальное количество информации, необходимое для обеспечения целостности программ. Как правило, процессор сохраняет только содержимое счетчика команд и регистра состояния процессора. Любая дополнительная информация должна сохраняться программным путем в начале работы ПОП и восстанавливаться перед ее завершением.

Таким образом, при возникновении прерывания в стеке сохраняется регистр флагов и адрес возврата (содержимое ***cs*** и ***ip***), куда нужно будет вернуться после обслуживания прерывания. Затем из вектора прерываний извлекается адрес ПОП и управление передается по этому адресу. Отработав, ПОП должна обеспечить возврат управления. Для этого она завершается командой ***iret***, снимающей из стека адрес возврата и старое содержимое регистра флагов и загружающей эти значения в ***cs, ip*** и регистр флагов.

ПОП во многих случаях рассматриваются программистами как обычные процедуры для выполнения каких-либо действий (например, чтение кода нажатой клавиши или вывод символа на экран). Программные прерывания могут вызываться друг из друга, то есть допустима вложенность.

4.2 КОНТРОЛЛЕР ПРЕРЫВАНИЙ

Аппаратные прерывания возникают в связи с поступлением сигналов запросов на прерывания от внешних источников (таймер, клавиатура, мышь и т.д.). Обслуживанием аппаратных прерываний управляет специальное устройство - контроллер прерываний, поэтому для написания собственных обработчиков этих прерываний необходимо познакомиться с особенностями функционирования и программирования контроллера прерываний.

Обрабатывая прерывание от устройства, процессор должен запомнить, что запрос на прерывание от устройства распознан, и, возможно, проинформировать об этом устройство, а после передачи управления ПОП или после выполнения ПОП требуется снятие сигнала запроса на обслуживание прерывания. Альтернативой такому подходу является пересылка данных между процессором и интерфейсом ввода-вывода устройства. Для этого в программе обработки прерывания должна быть выполнена команда, которая изменяет значение в регистре состояния или регистре данных в интерфейсе устройства и тем самым явно информирует устройство о том, что его запрос прерывания получен процессором и обслужен.

Устройства ввода-вывода запрашивают прерывания путем активизации линии шины, называемой линией запроса прерывания ***IRQ***.

Несмотря на то, что в компьютере прерывания могут запрашиваться несколькими устройствами ввода-вывода одновременно, рассмотрим сначала возможный алгоритм обработки запроса на прерывание, поступающего от одного устройства. Когда устройство активизирует сигнал запроса прерывания, оно поддерживает этот сигнал активным до тех пор, пока не узнает, что процессор принял запрос. Это означает, что сигнал запроса прерывания будет активен еще какое-то время спустя после вызова ПОП – до тех пор, пока не будет выполнена команда обращения к данному устройству. **Важно**, чтобы этот активный сигнал не привел к следующему прерыванию и не заставил систему войти в бесконечный цикл. Существует несколько механизмов решения данной проблемы. Например, можно игнорировать сигнал на линии ***IRQ*** до окончания выполнения первой команды в ПОП. При этом первая команда ПОП должна запретить прерывания до окончания действия данной программы. Как правило, команда, разрешающая прерывания, является последней командой ПОП и предшествует команде возврата из прерывания. При этом необходимы гарантии, что выполнение команды возврата из прерывания будет завершено до того, как станут возможными следующие прерывания. Другой механизм предполагает, что у процессора имеется специальная линия запроса на обслуживание прерываний, и что схема управления

прерываниями отвечает только на передний фронт сигнала. При такой схеме работы процессор получает только один запрос прерывания, независимо от того, как долго линия остается активной. Это значит, что повторяющихся прерываний быть не может, и нет необходимости явно отключать запросы прерывания на данной линии.

Рассмотрим случай, когда с процессором соединено несколько устройств, способных инициировать прерывания. Так как эти устройства функционально независимы, то они генерируют прерывания асинхронно, в том числе и одновременно. Поэтому, если процессор будет получать запрос на прерывание по общей линии для этих устройств, то ему перед передачей управления на ПОП будет нужна дополнительная информация, чтобы определить, какое из устройств активизировало эту линию. Если устройства запросили обслуживание одновременно, то необходимо не только выбрать одно из них для обслуживания, но и не потерять запросы от остальных устройств. Для решения этих задач запросы на аппаратные прерывания первоначально обрабатываются контроллером прерываний, и только после их обработки контроллер прерываний подает процессору сигнал запроса на обслуживание прерывание и по специальной шине данных передает процессору информацию, позволяющую идентифицировать требующуюся ПОП.

Для ответа на вопрос об установлении устройства, запросившего обслуживание, напомним, что информация, необходимая для идентификации устройства, имеется в регистре состояния этого устройства. Поэтому простейший способ определения устройства, запросившего прерывание, заключается в опросе всех присоединенных к шине устройств ввода-вывода. Однако главным недостатком такой схемы является время, уходящее на проверку устройств, которые не запрашивали прерывание. Для исключения такой ненужной проверки в контроллере прерываний реализована однозначная взаимосвязь между линией IRQ запроса на прерывание и номером ПОП в векторе прерываний. Опрос устройств выполняется только тогда, когда по одной линии IRQ могут запрашивать обслуживание несколько устройств. Признак запроса на прерывание по линии IRQ хранится в специальном регистре контроллера прерываний – регистре запросов.

Таким образом, если предположить, что изначально прерывания разрешены, то последовательность событий, происходящих в ходе обработки запроса прерывания от одного устройства, будет следующей:

1. Устройство генерирует запрос прерывания.
2. Процессор прерывает текущую выполняемую программу.
3. Последующие прерывания запрещаются, для чего изменяется бит, управляющий прерываниями, в регистре флагов.

4. Устройство информируется о том, что его запрос распознан, и в ответ сбрасывает сигнал запроса на прерывание.
5. Запрошенное прерыванием действие выполняется программой обработки прерывания.
6. Прерывания разрешаются, выполнение программы возобновляется.

Несмотря на кажущуюся логичность вышеприведенного алгоритма, он имеет недостаток, связанный с тем, что существуют устройства, для которых время ожидания обслуживания запроса на прерывание не должно быть длительным. Поясним этот момент более подробно.

Если во время выполнения ПОП все прерывания запрещены, то начатая ПОП всегда выполняется до конца. ПОП в большистве своем достаточно коротки, и вызываемая ими задержка для преобладающей части устройств обычно бывает вполне приемлемой. Однако в некоторых случаях большая задержка с обслуживанием прерывания может привести к неверному функционированию устройств. Рассмотрим работу компьютера, отслеживающего время с помощью таймера реального времени – устройства, которое направляет процессору запросы прерываний через фиксированные промежутки времени. Пусть по каждому из таких запросов процессор выполняет короткую программу обработки прерывания, увеличивающую хранящийся в памяти набор значений счетчиков, содержащий количество минут и секунд. Правильное функционирование таймера возможно при условии, что время задержки перед обработкой запроса прерывания значительно меньше временного интервала между запросами. Это требование будет выполняться лишь в том случае, если запрос прерывания от таймера будет приниматься во время выполнения ПОП, вызванной другим устройством. Этот пример показывает, что для правильной организации ввода-вывода должна использоваться система приоритетов устройств. Во время обслуживания процессором прерывания от устройства им должны приниматься запросы прерываний от устройств с более высоким приоритетом. Многоуровневая система приоритетов означает, что в ходе выполнения ПОП запросы на прерывания от одних устройств будут приниматься, а от других – нет.

Для того чтобы реализовать такую схему обработки прерываний, возможно два основных способа. Первый способ заключается в том, что процессору присваивается уровень приоритета, который будет меняться в зависимости от выполняемой программы. Уровень приоритета процессора – это уровень приоритета текущей выполняющейся программы. Процессор принимает прерывания от устройств, имеющих более высокий приоритет, чем его собственный. Когда начинается выполнение ПОП некоторого устройства, процессору назначается приоритет этого устройства. Тем самым запрещаются прерывания от любых

устройств с тем же или более низким приориетом. Идея этого метода впервые реализована в WINDOWS 2000 при обслуживании прерываний несколькими процессорами (концепция IRQL – IRQ Level).

Второй способ заключается в том, что многоуровневая схема приоритетов может быть реализована с помощью отдельных линий запроса и подтверждения прерываний для каждого устройства. При этом каждой линии запроса прерывания присваивается свой уровень приоритета. Запрос на прерывание считается принятым, если у него более высокий уровень приоритета, чем у программы, выполняющейся в данный момент.

Сигналы запросов на прерывания от внешних устройств поступают в процессор не непосредственно, а через контроллер прерываний. Традиционный программируемый контроллер прерываний включает два контроллера, один из которых называется ведущим, а второй ведомым. Каждый из них имеет 8 входов ***IRQ*** (***Interrupt ReQuest*** - Запрос на прерывание), к которым подключены выводы внешних устройств (***IRQ0, IRQ1, IRQ3...IRQ7*** для ведущего контроллера, ***IRQ8…IRQ15*** для ведомого контроллера). Ко входу IRQ2 ведущего контроллера подключен выход ведомого контроллера, что позволяет увеличить возможное число входных устройств(8 у ведомого, 7 у ведущего). Выход ***INT*** ведущего контроллера подключается к входу микропроцессора ***INTR***, а выход ***INT*** ведомого – к входу ***IRQ2*** ведущего. В результате обслуживания запроса контроллер должен передать процессору сигнал ***INT*** (он попадает на вывод микропроцессора ***INTR***) и номер ПОП в векторе прерывания, используя линии данных. Номер ПОП вычисляется сложением базового номера (он равен 8h для ведущего контроллера или 70h для ведомого) с номером входной линии ***IRQ***. Так, по линии ***IRQ0*** поступают прерывания от таймера и вызывается ПОП ***int 08h***, а по линии ***IRQ1*** поступают прерывания от клавиатуры и вызывается ПОП ***int 09h***.

Ведущий контроллер программируется через порты ***20h, 21h***, а ведомый - через порты ***A0h, A1h***. Базовый номер у ведомого контроллера равен 70h, поэтому аппаратные прерывания находятся в пределах 70h...77h.

Очевидно, что номера векторов аппаратных прерываний связаны с устройствами, перечень которых для ведущего контроллера приведен в таблице 4.1.

Сигналы или запросы на прерывание от внешних устройств имеют разные приоритеты. Стандартно приоритеты снижаются с увеличением номера ***IRQ***, т.е. прерывание от таймера имеет наивысший приоритет.

Рассмотрим внутреннюю структуру контроллера (рисунок 4.1). В нем имеются регистр входных запросов, регистр маски, схема приоритетов и регистр обслуживаемых запросов.

Таблица 4.1

Номер линии запроса	Номер в векторе прерываний	**Устройство**
IRQ0	08h	Таймер
IRQ1	09h	Клавиатура
IRQ2	0Ah	Вход от ведомого
IRQ3	0Bh	Последовательный порт COM2
IRQ4	0Ch	Последовательный порт COM1
IRQ5	0Dh	Параллельный порт LPT2
IRQ 6	0Eh	Гибкий диск
IRQ7	0Fh	Параллельный порт LPT1

Рассмотрим, например, обработку контроллером запроса на прерывание от клавиатуры. Сигнал по линии ***IRQ1*** устанавливает в 1 соответствующий бит регистра запросов. Затем по содержимому регистра маски проверяется, разрешено или запрещено обслуживание поступившего запроса. Говорят, что прерывание замаскировано, если соответствующий ему бит в регистре маски равен 1 (на рисунке 4.1 замаскировано прерывание от COM1). Предположим, что прерывание от клавиатуры не замаскировано.

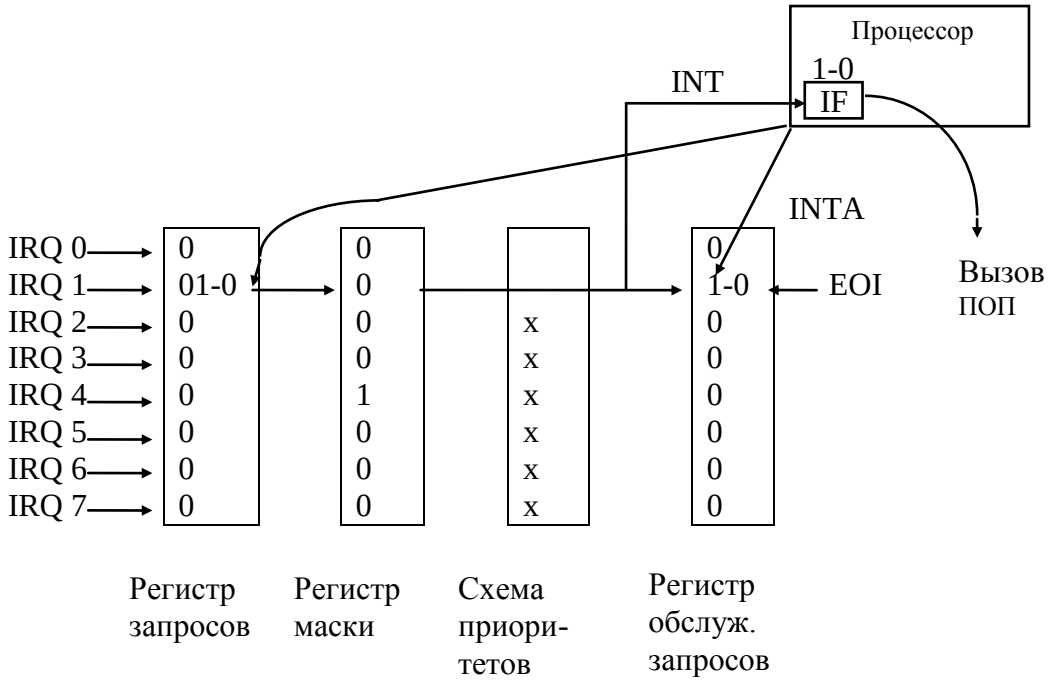


Рисунок 4.1 – Схема обслуживания запросов контроллером прерываний

Далее выполняется анализ схемы приоритетов. При стандартной настройке схемы приоритетов приоритеты сигналов ***IRQ*** снижаются по мере роста номера линии ***IRQ***, причем приоритеты у ведомого контроллера располагаются между приоритетами уровней ***IRQ1*** и ***IRQ3*** ведущего контроллера. Обработка более приоритетных прерываний блокирует обслуживание запросов прерываний с меньшими приоритетами. Если прерывание от

клавиатуры не заблокировано в схеме приоритетов из-за незавершенности обслуживания прерывания от таймера, как более приоритетного, то сигнал проходит через схему приоритетов.

Затем сигнал запроса на прерывание дает разрешение на установку признака обслуживания в регистре обслуживания (но не устанавливает соответствующий бит в регистре обслуживаемых запросов!) и поступает на вход ***INTR*** процессора. Процессор реагирует на сигнал ***INT*** только в том случае, если в регистре флагов установлен флаг прерываний ***IF.*** Отметим, что установка ***IF*** в 1 выполняется командой ***sti***, а сброс в 0 - командой ***cli.***

Процессор после получения сигнала ***INT*** выполняет следующие действия:

1) посылает в контроллер прерываний сигнал ***INTA*** (***Interrupt Acknowledge***, подтверждение прерывания), который устанавливает в 1 бит в регистре обслуживания запросов, устанавливает в 0 бит в регистре запросов, формирует номер вектора прерывания, передаваемый в процессор по линиям данных. Если снова возникнет запрос на это же прерывание, то он не потеряется, а запомнится в регистре запросов для последующего обслуживания;

2) сбрасывает флаг ***IF*** в 0, запрещая все прерывания. Прерывания остаются запрещенными либо до выполнения команды ***sti***, либо до установки флага ***IF*** любым другим способом, например, восстановления старого содержимого регистра флагов из стека.

Следует помнить, что установка в 1 бита в регистре обслуживаемых запросов вызывает блокировку в схеме приоритетов всех прерываний с более низкими приоритетами, чем обслуживаемое. Сбросить бит в регистре обслуживаемых запросов и снять блокировку в схеме приоритетов можно, заслав число 20h в порт 20h для ведущего контроллера или в порт A0h для ведомого контроллера. Эта засылка называется командой ***EOI*** (***End of Interrupt***):

```
mov al,20h
out 20h,al
```

Если ***EOI*** отсутствует в обработке аппаратного прерывания, то прерывания более низких приоритетов так и останутся заблокированными.

Отметим, что запросы на прерывания, поступающие в ведущий контроллер, блокируют только ведущий контроллер, а запросы на прерывания, поступающие в ведомый контроллер (***IRQ8-IRQ15***) блокируют не только низшие уровни в ведомом контроллере, но и уровни ***IRQ2-IRQ7*** в ведущем контроллере. Поэтому в обработчике аппаратных прерываний для линий ***IRQ8-IRQ15*** необходимо выполнить запись 20h в порт 20h и в порт A0h. Для изменения работы контроллера прерываний его можно перепрограммировать посылкой в порты 20h и 21h управляющих слов специального формата.

Для обработки прерываний в многопроцессорных системах стандартный контроллер прерываний непригоден из-за ориентированности сигналов ***INTR*** и ***INTA*** на единственность процессора. Начиная с PENTIUM II введен APIC (Advanced Programmable Interrupt Controller), который способен работать в многопроцессорных системах, а для совместимости с однопроцессорными вычислительными системами поддерживает режим, совместимый со стандартным контроллером прерываний.

4.3 СТРУКТУРА ОБРАБОТЧИКА ПРЕРЫВАНИЙ

Рассмотрев обслуживание запросов на прерывания контроллером прерываний и процессором, перейдем к написанию собственной программы обработки прерываний.

В ответ на поступивший от внешнего устройства запрос на прерывание процессор приостанавливает выполнение одной программы и начинает выполнение другой. Поэтому выполнение ПОП должно тщательно контролироваться, и должна быть возможность запрещать и разрешать прерывания по мере необходимости. Для этой цели используются команды **`cli`** и **`sti`**. Структура обработчика аппаратного прерывания зависит от желаемой программистом реакции процессора и контроллера прерываний на возникновение новых запросов на прерывание в то время, пока не завершена обработка текущего. Рассмотрим 3 типа структуры ПОП:

Тип 1

```
myint proc far
     <текст>
mov al, 20h
out 20h, al
iret
myint endp
```

Тип 2

```
myint proc far
sti
  <текст>
mov al, 20h
out 20h, al
iret
myint endp
```

Тип 3

```
myint proc far
    sti
     <текст>
    cli
    mov al, 20h
    out 20h, al
    iret
myint endp
```

В обработчике типа 1 запрещены все прерывания, т.к. при входе в обработчик флаг ***IF*** сбрасывается. При этом обработка прерываний более высокого приоритета также блокирована (например, при обслуживании запроса по линии ***IRQ1*** блокирован таймер).

Поэтому более грамотно писать обработчик аппаратного прерывания по типу 2, когда в начале ПОП командой **`sti`** разрешаются прерывания более высокого приоритета. Команду ***EOI*** посылают в контроллер перед завершением ПОП. Однако в промежуток времени после снятия блокировки в схеме приоритетов и выполнением команды ***iret*** может возникнуть запрос на прерывание того же уровня или более низкого. Эта ситуация называется

вложенным прерыванием. Если ПОП написана по типу 2, то при разрешенных прерываниях (установлен флаг ***IF***) запрос на прерывание любого уровня прервет выполнение нашего обработчика. В случае, если требуется обработка запроса на прерывание по той же линии ***IRQ***, программа обработчика, не успев завершиться, будет снова выполняться сначала. Подобное явление может нарушить работоспособность системы.

В связи с вышеизложенным оптимальной структурой обработчика аппаратного прерывания считается структура типа 3, где перед командой ***EOI*** все прерывания (в том числе более низких приоритетов и данного) запрещаются командой **cli**. После выполнения команды ***EOI*** флаг ***IF*** восстанавливается из стека командой **iret**.

Для того чтобы обработчик мог выполняться, необходимо поместить его адрес (сегмент и смещение) в вектор прерываний. Это можно осуществить прямой записью нужных значений в вектор прерываний, однако во избежание ошибок вычисления адресов модификация вектора прерываний обычно реализуется с помощью функции ***25h*** прерывания ***21h***, причем в регистр ***al*** должен быть помещен номер вектора прерывания, а в регистры ***ds*** и ***dx*** – соответственно сегмент и смещение для нового обработчика. Старое содержимое вектора прерывания следует предварительно извлечь из вектора прерываний с помощью функции ***35h*** прерывания ***21h***, причем сегмент и смещение старого обработчика возвращаются в регистрах ***es*** и ***bx***.

Пример 4.1.

```
segm segment para public 'code'
assume cs:segm,ds:segm,ss:segm,es:segm
org 100h
start: jmp main
<описание данных>
oldint dd ?     ;двойное слово для хранения адреса старого
                ;обработчика
newint08 proc far   ;новый обработчик прерывания от таймера
<текст программы обработчика>
newint08 endp
main:
  push cs
  pop ds
  mov ax, 3508h
  int 21h       ;в es:bx - адрес старого обработчика int 08
  mov word ptr oldint,bx ;в младшее слово сохраняем смещение
  mov word ptr oldint+2,es     ;в старшее слово - сегментную
                               ;часть адреса
```

```
mov ax, 2508h          ; установим свой обработчик
mov dx, offset newint08
int 21h
<операторы программы>
segm ends
end start
```

Прерывание может быть реакцией на какую-то особую для прикладного программиста ситуацию в системе. Программисту для вызова своего обработчика прерывания предоставляются так называемые пользовательские вектора прерываний - **60h - 67h**. В этом случае обработчик пишется как процедура, а затем его адрес обработчика запоминается в одном из пользовательских векторов прерываний. Вызов своего обработчика осуществляется по команде **int**.

Обработчик прерывания может либо полностью заменять системный, либо взаимодействовать с системным на различных этапах своей работы. Случай, когда свой обработчик полностью заменяет системный, рассмотрен в примере 4.1.

При написании своего обработчика прерывания, взаимодействующего с системным, необходимо учитывать момент вызова системного обработчика – до или после прикладной обработки. Такая методика “сцепления” собственной программы с системной широко используется при модификации как программных, так и аппаратных прерываний.

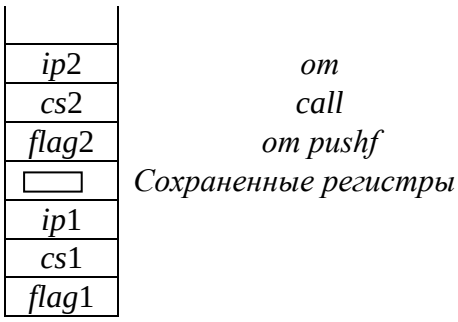
Случай 1. Прикладная обработка всегда выполняется после системной.

В двойном слове ***oldint*** необходимо сохранить адрес системного обработчика, а адрес нового обработчика ***newint*** помещается в вектор прерываний. ПОП ***newint*** имеет структуру следующего типа:

```
newint proc far
<сохранение регистров>
pushf
call cs: oldint
...<прикладная обработка>
<восстановление регистров>
iret
newint endp
```

При передаче управления на ***newint*** в стеке находятся содержимое регистра флагов (flag1) и содержимое регистров ***cs*** и ***ip*** (cs1 и ip1), представляющие собой адрес возврата в программу, вызвавшую наш обработчик ***newint***, а к моменту выполнения ***oldint*** в стек снова помещаются значения регистра флагов (flag2) и адреса возврата в точку, где начинается прикладная обработка (cs2 и ip2). Обработчик ***oldint*** заканчивается командой **iret**, которая

снимает из стека flag2, cs2, ip2, возвращаясь в ***newint***. Команда **iret** в ***newint*** снимает из стека ip1, cs1 и flag1, реализуя возврат в точку вызова нашего обработчика ***newint***.



Случай 2. Прикладная обработка всегда выполняется до системной. При этом переход на системный обработчик прерывания может быть осуществлен по команде **jmp**:

```
newint proc far
<прикладная обработка>
jmp cs:old_int
newint endp
```

Так как системный обработчик прерывания выполняется после прикладной обработки**,** и возврат в обработчик ***newint*** не нужен, то в обработчике ***newint*** команду **iret** можно не указывать**,** а снятие из стека содержимого регистра флагов и адреса возврата осуществит системный обработчик, заканчивающийся командой **iret.**

Случай 3. Прикладная обработка частично выполняется до системной, частично - после системной. Структура обработчика аналогична случаю 1, только часть прикладной обработки - перед оператором **pushf.**

Случай 4. В зависимости от некоторых условий выполняется либо собственная обработка ситуации, либо обработка ситуации передается системной ПОП:

```
new_int proc far
     <формирование признака  необходимости
     системной обработки: 0 -своя, 1 - системная.>
     jnz system
     <своя обработка>
     iret
     system: jmp cs:old_int
new_int endp
```

4.4 ПРИМЕРЫ ПРОГРАММ ОБРАБОТКИ НЕКОТОРЫХ ПРЕРЫВАНИЙ

4.4.1 Прерывание 1Ch.

Прерывание ***1Ch*** вызывается из обработчика аппаратного прерывания ***int 08h*** (прерывание от таймера). Сигналы от таймера поступают в контроллер прерываний с частотой 18,2 раза в секунду и инициируют вызов обработчика прерываний с номером ***08h***. Стандартно прерывание ***1Ch*** содержит только команду **`iret`**, т.е. является заглушкой. Прерывание ***1Ch*** служит для использования программистом сигналов от таймера без разрушения работы системных часов.

При написании обработчика ***1Ch*** нужно помнить, что вызов обработчика происходит не из программы пользователя, а значит, ***ds*** указывает на чужие данные, ***ss*** - на чужой стек. Чужим стеком в принципе можно воспользоваться, а вот чтобы обработчик работал с собственными данными программиста, нужно выполнить один из следующих пунктов:

а) расположить данные в кодовом сегменте и адресовать их через ***cs***;

б) расположить данные в сегменте данных, сохранить чужой ***ds***, загрузить собственный ***ds*** и восстановить его в конце обработки.

4.4.2 Обработка прерываний от клавиатуры

Клавиатура, как известно, состоит из набора переключателей, объединенных в матрицу. При нажатии клавиши процессор, установленный в самой клавиатуре (контроллер клавиатуры), определяет координаты нажатой клавиши в матрице. Связь клавиатуры с системным блоком осуществляется через последовательный канал, данные по которому передаются по 11 битов, 8 из которых представляют собой собственно данные, а остальные биты являются синхронизирующими и управляющими. Связь между клавиатурой и системным блоком является двусторонней, т.е. клавиатура может как передавать, так и принимать данные.

Обработка сигналов от клавиатуры выполняется программой обработки прерывания ***int 09h***. ***Int 09h*** - это аппаратное прерывание, соответствующее линии ***IRQ1*** в контроллере прерываний.

В процессе работы ПОП ***int 09h*** имеет дело с информацией, хранящейся в следующих специальных структурах:

1) порты ***60h*** и ***61h*** контроллера клавиатуры;

2) слово флагов клавиатуры по адресу ***40h:17h***;

3) кольцевой буфер клавиатуры.

Контроллер клавиатуры распознает, какая клавиша нажата на клавиатуре, и посылает код нажатой или отпущенной клавиши в порт ***60h***. Этот код называется *scan-кодом*. Можно считать, что scan-код - это номер клавиши на клавиатуре. Каждой клавише соответствует 2 кода, отличающиеся друг от друга на 80h. Меньший код - это код нажатия клавиши, больший - код отпускания. При нажатии некоторых клавиш генерируется не один scan-код, а так называемая scan-последовательность кодов, например, клавише Insert соответствует последовательность <E0h 52h>, клавише Delete - <E0h 53h>.

Скан-код считывается из порта в регистр ***al*** командой

```
in al,60h
```

Для многобайтовых scan-кодов нужно для помещения в порт 60h новых кодов из scan-последовательности посылать подтверждение чтения из 60h порта в порт 61h. Это подтверждение необходимо и для однобайтовых scan-кодов, чтобы в 60h мог быть помещен новый scan-код.

Команды подтверждения могут быть записаны следующим образом:

```
in al, 61h      ;чтение старого содержимого порта 61h
mov ah, al      ;сохраним старое содержимое порта 61h в ah
or al, 80h      ;в старший разряд al запишем 1
out 61h, al     ;запись al в порт 61 h
mov al, ah      ;в al - старое содержимое порта 61h
out 61h, al     ;запись в 61h порт его старого содержимого
```

Порт 61h доступен как для чтения, так и для записи. Он служит для управления не только клавиатурой, но и другими устройствами компьютера, например, встроенным динамиком, поэтому для изменения его содержимого необходимо пользоваться логическими операциями, чтобы не изменить остальные биты.

Слово флагов клавиатуры располагается по адресу 40h:17h. Каждому биту слова флагов соответствует клавиша из группы специальных клавиш (см. рисунок 4.2).

Пока клавиша нажата и не отпущена, бит, соответствующий этой клавише, установлен в 1. При отпускании клавиши бит становится равен 0. При отпускании клавиши (или комбинации клавиш) анализируются scan-коды отпускания и биты сбрасываются в 0. Слово флагов клавиатуры может быть использовано для определения нажатия комбинации клавиш Ctrl-Alt-Del, в частности, в обработчике прерываний от клавиатуры можно отслеживать нажатие клавиши Del и анализировать нажатие Ctrl и Alt через слово флагов клавиатуры.

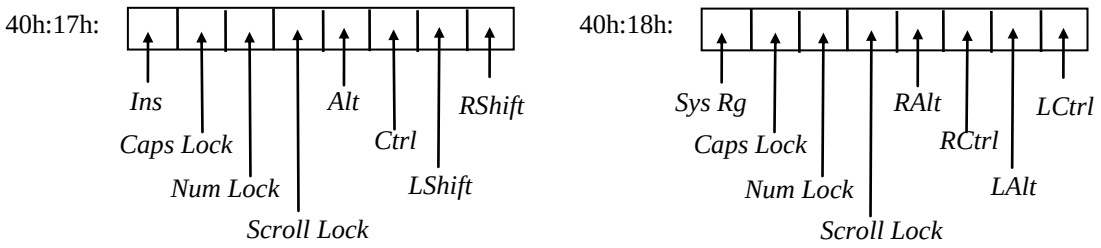


Рисунок 4.2.

Кольцевой буфер клавиатуры служит для синхронизации ввода с клавиатуры и обработки этих клавиш программой. Расположение буфера клавиатуры задается словами по адресам 0000h:0480h и 0000h:0482h, где хранятся соответственно смещения адреса начала и адреса конца буфера относительно сегментного адреса 0040h. Обычно там содержатся 001Eh и 003Eh.

Буфер имеет длину 16 слов и организован по принципу циклической очереди. Имеется 2 указателя - на хвост и голову очереди. В хвостовом указателе (хранится по адресу 40h:1Ch) содержится адрес первой свободной ячейки (слова) буфера, в указателе на голову буфера (хранится по адресу 40h:1Ah) содержится адрес самого старого кода, введенного с клавиатуры, но не использованного еще программой.

В начале работы оба указателя указывают на первую ячейку буфера. Если буфер пуст, то голова и хвост равны и указывают на начало буфера. В процессе занесения слов в буфер клавиатуры хвостовой указатель может дойти до конца буфера, что должно быть учтено при поступлении очередного слова в буфер. Адрес в указателе в этом случае должен не увеличиваться на 2, как обычно, а из текущего значения указателя должна быть вычтена длина буфера, чтобы хвост указывал на первую ячейку буфера клавиатуры. Аналогичным образом изменяется указатель на голову очереди при чтении символов. Если же значение хвостового указателя в процессе заполнения буфера клавиатуры таково, что может произойти затирание головы очереди при выполнении записи слова в буфер, то это означает переполнение буфера и вызывает предупреждающий звуковой сигнал.

Для каждой клавиши ее нажатие обрабатывается int 09h по следующему алгоритму:

1) из порта 60h считывается scan-код клавиши. Если это scan-код управляющей клавиши и является кодом нажатия, то в слове флагов клавиатуры устанавливается бит (рисунок 4.3). Биты в слове флагов не сбрасываются в 0 до тех пор, пока клавиши остаются нажатыми. Если управляющая клавиша отпускается, то анализируется scan-код отпускания клавиши и соответствующий бит в слове флагов сбрасывается в 0.

2) если нажата не управляющая клавиша, то выполняется трансляция (перевод) scan-кода в двухбайтовый код, старший байт которого - scan-код, а младший - ASCII-код,

определяющий символ. При трансляции используются специальные таблицы. Так как за каждой нажатой клавишей закреплено несколько ASCII-кодов (минимум 2 – “a” и “A”, например), то формируемый ASCII-код зависит от нажатия клавиш Shift и CapsLock, что требует анализа при трансляции слова флагов клавиатуры.

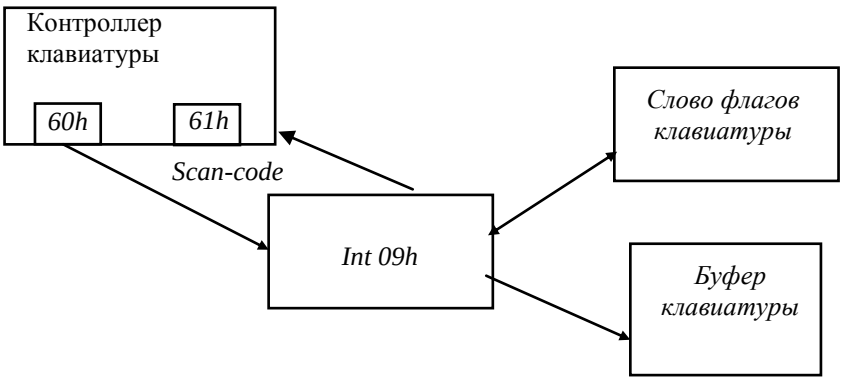


Рисунок 4.3 – Общая схема функционирования int 09h

3) полученные после трансляции 2 байта засылаются в кольцевой буфер клавиатуры по адресу в хвостовом указателе. Затем хвостовой указатель изменяется так, чтобы указывать снова на свободную ячейку. При движении по буферу вперед хвостовой указатель увеличивается на 2, при достижении до конца буфера от него отнимается длина буфера. Рассмотрим запись в буфер клавиатуры содержимого регистра **ax** (в ***ah*** – scan-код, в ***al*** – ASCII-код), если сегментный регистр **es** содержит 040h:

```
mov di,es:[01Ch]    ;хвост – в di
mov es:[di],ax      ;запись ax по адресу в хвостовом
                    ;указателе
cmp di, 03Ch        ;дошли до конца буфера ?
je beg          ;если да, то указатель переместим на начало
inc di              ;иначе увеличим указатель на 2
inc di
jmp save            ;на сохранение нового значения хвоста
beg: mov di, 01Eh   ;в di – адрес начала буфера
save: mov es:[01Ch],di ;сохраним новое значение хвоста
```

Если хвостовой указатель не может быть увеличен, т. к. при этом затрет информацию в голове буфера, то выдается звуковой сигнал. Головной указатель увеличивается после считывания двухбайтового кода из буфера клавиатуры.

Имеется ряд клавиш (F1-F12, Home, End, клавиши-стрелки), которые не отображаются на экране. Для этих клавиш двухбайтовый код имеет следующую структуру: первый байт - scan-код, второй байт равен 0, так как ASCII - кодировка символов отсутствует. Такие коды называются расширенными ASCII-кодами. Расширенные ASCII-коды генерируются также при нажатии комбинаций управляющих и функциональных клавиш, при этом в старший байт расширенного ASCII-кода помещается не scan-код, а код, специально выбранный для этой комбинации клавиш.

Рассмотрим пример ПОП от клавиатуры, игнорирующей нажатие клавиши “пробел”.

Пример 4.2.

```
old09 dd ?     ; адрес для старого обработчика
Int09 proc far
sti            ;разрешим прерывания более высокого приоритета
push ax        ;сохраним в стеке изменяемый регистр
in al, 60h     ;считаем scan-код
mov ah, al     ;и поместим его в ah
cmp ah,39h     ;сравним полученный код со scan-кодом пробела
je blok        ;если нажат пробел, то на метку blok
jmp pass       ;если нет, то на метку pass
blok: in al,61h    ;посылаем подтверждение
mov ah,al      ;о считывании scan-кода
or al,80h      ;из порта 60h
out 61h,al     ;в порт 61h
mov al,ah
out 61h,al
cli            ;запретим прерывания
mov al,20h     ;пошлем контроллеру прерываний
out 20h,al     ;сигнал EOI
pop ax         ;восстановим из стека ax
iret           ;завершим обработчик прерываний
pass:pop ax    ;восстановим ax
jmp cs:[old09];на выполнение старого обработчика
int09 endp     ;конец процедуры
```

Scan-коды клавиш приведены в приложении Б.

Порт 60h может быть использован для записи, что позволяет устанавливать время ожидания перед переходом клавиатуры в режим автоповтора, устанавливать время генерации скан-кода при автоповторе и управлять светодиодами на клавиатуре.

Для реализации этого необходимо сначала убедиться, что внутренняя очередь команд процессора клавиатуры пуста. Для этого осуществляется чтение слова состояния из порта 64h. Если 1 бит равен 0, то очередь пуста. После этого в порт 60h можно выводить команду (1 или несколько байтов). Если команда состоит более чем из одного байта, то перед посылкой каждого байта необходимо осуществлять проверку готовности Intel8042. Для установки периода автоповтора и задержки включения автоповтора в порт 60h записывается сначала код команды 0F3h, после чего записывается байт, имеющий следующую структуру:

биты 0-4 - количество генераций scan-кода клавиши в 1 секунду (0 - 30, 2 - 24, 4 - 20, 8 - 15, 0Ah - 10, 14h - 5, 1Fh - 2)

биты 5-6 - задержка включения автоповтора, мс (00 - 250, 01 - 500, 10 - 750, 11 - 1000);

бит 7 - должен быть равен нулю. BIOS устанавливает 500 мс и 10 повторов в секунду.

Для управления светодиодами в порт 60h сначала засылается команда 0EDh, затем байт, биты 0,1,2 которого (1 или 0) сигнализируют о включении/выключении соответственно клавиш ScrollLock, NumLock, CapsLock.

4.5 РЕЗИДЕНТНЫЕ ПРОГРАММЫ

Программы, которые, загрузившись один раз, продолжают функционировать до перезагрузки ОС или до тех пор, пока они не будут специальными средствами выгружены из памяти, называются резидентными. Резидентные программы по другому называются TSR-программами (Terminate and Stay Resident).

Обработчики прерываний могут быть как транзитными, так и резидентными. Рассмотрим структуру резидентной программы типа *.com. Обычно она состоит из 2 частей - резидентной, остающейся в памяти, и секции инициализации.

Пример 4.3.

```
myseg segment para public 'code'
assume cs:myseg,ds:myseg
org 100h
start:jmp init            ; переход на секцию инициализации
<данные резидентной части>
begrez: <код резидентной части>
init:
<секция инициализации>
mov dx, offset init
int 27h
myseg ends
end start
```

При запуске программы управление передается на секцию инициализации, которая должна обеспечить функционирование резидентной части. В конце секции инициализации обычно вызывается прерывание ***int 27h*** или прерывание ***int 21h*** с функцией ***31h***, которые реализуют закрепление за резидентной частью памяти, необходимой для ее функционирования. Секция инициализации после установки резидента отбрасывается. Резидентная же часть программы остается в памяти в пассивном состоянии. Активизация TSR-программы может осуществляться 3 способами:

1) через аппаратное прерывание;

2) через программное прерывание (int);

3) через call far-вызова.

Отметим, что любая резидентная программа имеет по крайней мере две точки входа. В данном примере ***start*** является точкой входа при загрузке, а ***begrez*** – это точка входа при активизации резидентной программы.

Рассмотрим более подробно способы установки резидента:

1. Через прерывание ***27h***. При этом ***cs*** должен указывать на начало ***PSP***, что так и есть в ****.com*** программах. Регистр ***dx*** должен содержать смещение конца программы, отсчитываемое от начала ***PSP***, т.е. длину резидентной части.

Для программ ****.exe*** надо записать ***27h*** во второй байт ***PSP*** (первый байт содержит код инструкции ***int,*** второй - код номера прерывания, по которому осуществляется выход из программы) и завершить программу ***ret far.***

2. Через функцию ***31h*** прерывания ***21***h. В ***dx*** должно содержаться количество параграфов, занятых резидентной частью программы. Это количество вычисляется по формуле (***init – start + 10Fh)/16. Init-start*** соответствует длине в байтах резидентной части программы, ***100h*** – это размер ***PSP, 0Fh*** необходимо добавлять к количеству байтов, чтобы после целочисленного деления на 16 результат был округлен в большую сторону.

В примере 4.3 не рассмотрены средства удаления резидентной программы из памяти. Наиболее простым способом выгрузки из памяти резидентной программы является использование второй копии этой же программы, которая при повторном запуске выгружает из памяти первую копию и сама в памяти не остается. Отметим, что перед выполнением освобождения памяти, занимаемой резидентной программой и ее окружением, необходимо восстановить все векторы прерываний, перехваченные резидентной программой. При этом понятно, что корректное восстановление старых обработчиков прерываний, перехваченных резидентом, возможно только в том случае, если те же самые вектора не были позже перехвачены другой программой.

Собственно освобождение памяти можно реализовать с использованием функции ***49h*** прерывания ***int 21h***. Единственным параметром, необходимым для корректной отработки освобождения памяти, является сегментный адрес освобождаемого блока памяти, который должен быть занесен в регистр ***es.***

Для выполнения корректного освобождения памяти необходимо рассмотреть механизм выделения памяти для программы. Напомним, что для каждой программы в память загружается не только сама эта программа с PSP, но и специальная структура, называемая окружением. Окружение представляет собой область памяти, в которой в виде символьных строк хранятся имена и значения некоторых переменных, необходимых для работы

программы. Эти переменные называются переменными окружения. Имеется ряд переменных окружения, имена которых зарезервированы и известны системе, однако пользователь может включать в окружение свои переменные с помощью команды SET. Системные и прикладные программы могут извлекать значения переменных окружения и анализировать их. Ценной информацией, хранящейся в окружении, является имя загруженной программы с указанием полного пути файла, где хранится запущенная программа ****.exe*** или ****.com***. Обычно окружение размещается перед программой, а сегментный адрес окружения помещается в ***PSP*** программы по смещению ***2Ch*** от начала ***PSP***. В связи с этим необходимо освобождать не только память, занятую программой, но и память, занимаемую окружением.

Рассмотрим пример структуры резидентной программы, если необходимо оставить в памяти резидентно обработчик прерывания ***myint***.

Пример 4.4.

```
  myseg segment para public 'code'
  assume cs:myseg, ds:myseg
  org 100h                ; PSP
  start: jmp init
  oldint dd ?         ;для хранения адреса старого обработчика
  myint proc FAR      ;обработчик прерывания
              …….
  myint endp
  init:    <проверка загруженности резидента>
           <если не загружен - то инсталляция, >
           <иначе – выгрузка в точке OutMem>
  je OutMem
  Install: mov ah, 35h
         mov al,<номер вектора прерывания, соотв. myint>
         int 21h     ;в es:bx вернется адрес старого обработчика
         mov word ptr OldInt,bx
         mov word prt OldInt+2,es
         mov bx, cs
         mov ds, bx
         mov dx, offset myint ; установим наш обработчик
         mov ah, 25h
         mov al,<номер вектора для myint>
         int 21h
         lea dx,init
         int 27h               ;оставим резидентом
OutMem:  mov dx,word ptr es:oldint ;oldint адресуем через es!
         mov bx, word ptr es:oldint+2
         mov ds,bx
         mov ah, 25h      ;восстановим старый обработчик
         mov al,<номер вектора для myint>
         int 21h
         push es
         mov es, es:[2Ch];освобождаем блок, где хранится
         mov ah, 49h      ;окружение программы
         int 21h
```

```
          pop es            ;освобождаем память, занятую
          mov ah,49h        ;программой, в es-сегментн. адрес
          int 21h                ;освобождаемого блока
          int 20h
myseg ends
end start
```

Резидентная программа, рассмотренная в примере 4.4, включает в себя блок, в котором проверяется, не была ли загружена резидентная *.com или *.exe программа в память многократно. Если резидентную программу запустить повторно, то в памяти появится еще 1 экземпляр той же программы. Если резидентная программа не перехватывает вектора прерываний, то вредным последствием загрузки второй копии будет напрасное расходование памяти. Если в резидентной программе сохраняются адреса старых обработчиков некоторых прерываний и устанавливаются свои, то вторая копия в качестве адресов старых системных обработчиков прерываний сохранит адреса, уже модифицированные первой копией. Подобной ситуации следует избегать, так как это может нарушить работоспособность программы. В связи с этим обязательным элементом любой резидентной программы является процедура ее защиты от повторной установки.

Для предотвращения повторной загрузки целесообразно использовать один из описанных ниже способов. Наиболее простым способом является установка сигнатуры. Сигнатура - это специальный код, находящийся в резидентной части программы и занимающий несколько байтов, по содержимому которых определяется факт установки программы. Для простоты вычисления адреса сигнатуры она обычно располагается перед программой какого-либо обработчика прерывания. Использование сигнатуры для предотвращения повторной загрузки иллюстрируется следующим примером.

Пример 4.5

```
  start: jmp init
  sign dw ADA0h      ;сигнатура
  int09 proc far
                     ;обработчик, оставляемый резидентно
  int09 endp
  init:              ;начало секции инициализации
  mov ax,3509h       ;получим адрес обработчика int 09h
  int 21h            ;в es:bx - адрес обработчика int09h
; Если резидент уже в памяти, то в векторе прерываний - адрес
;нашего обработчика и двумя байтами раньше располагается
;сигнатура, которую нужно проверить:
  mov ax,es:[bx-2]   ;в ax-содержимое двух байтов перед
                     ;обработчиком
  cmp ax,cs:sign     ;сравним эти 2 байта с нашей
                     ;сигнатурой
  jne no_exist       ;если не равны,то установка-в первый раз
  jmp exist          ;иначе-не в первый раз, и надо выгружаться
```

```
  no_exist:
     <загружаем и оставляем резидентно>
exist:
     <восстанавливаем вектора и освобождаем память>
```

Вторым способом проверки резидента на повторную загрузку является использование мультиплексного прерывания ***int 2Fh***, предназначенного для связи с резидентными программами. Для связи с резидентной программой необходимы некоторые соглашения о связях, в частности, для большей надежности идентификации "своей" функции резидентная программа может возвращать в регистрах заранее обусловленные значения.

Для организации взаимодействия с резидентной программой также можно использовать задание различных ключей в командной строке при ее запуске (например, загрузка или выгрузка). Все символы, введенные в командной строке после имени программы (так называемый хвост команды), помещаются в PSP. По смещению 80h от начала PSP хранится количество символов в хвосте команды, а начиная со смещения 81h хранятся символы, введенные с клавиатуры до нажатия Enter. Последним кодом хвоста является код 0Dh (возврат каретки). Таким образом, командную строку можно просмотреть посимвольно и определить, что хотел пользователь.

4.6 ПРЕРЫВАНИЯ В ЗАЩИЩЕННОМ РЕЖИМЕ

Так же, как и в реальном режиме, все прерывания защищенного режима имеют свои номера, причем их общее количество не должно превышать 256. Распределение номеров прерываний в защищенном режиме не совпадает с их распределением в реальном режиме. В современных моделях процессоров номера прерываний с 0 до 31 зарезервированы. Однако в реальном режиме номера прерываний некоторых обработчиков BIOS накладываются на номера из зарезервированного диапазона, что может являться источником конфликтов. В защищенном режиме соблюдается правило резервирования прерываний с номерами 0-31, а аппаратные прерывания и прерывания, определяемые пользователем, имеют номер вектора в диапазоне 32-255 (20h-0ffh).

В защищенном режиме аналогом таблицы векторов прерываний R-режима является дескрипторная таблица прерываний - ***IDT*** (Interrupt Descriptor Table). ***IDT*** является общесистемной и содержит 8-байтовые дескрипторы, определяющие расположение и атрибуты программ обработчиков прерываний. Элементы IDT называются шлюзами или вентилями. Шлюз включает 16-разрядный селектор и 32-разрядное смещение. Местоположение таблицы IDT определяется по содержимому системного регистра ***idtr***.

В общем случае при возникновении прерывания с номером N выполняется определение местонахождения ***IDT***, из таблицы по смещению N*8 извлекается дескриптор, определяющий расположение ПОП и в зависимости от типа шлюза осуществляется переход на ПОП. Следует отметить, что в отличие от реального режима, где перед передачей управления ПОП в стек заносятся 3 слова, в защищенном режиме в стек заносятся 3 или 4 двойных слова, включающих расширенный регистр флагов, селектор сегмента команд, смещение точки возврата, и, возможно, 32-битовый код ошибки. Код ошибки, если он есть, должен быть снят со стека соответствующим обработчиком. Команда **`iret`** должна снять со стека также 3 или 4 двойных слова.

Обработка прерываний в защищенном режиме зависит от типа прерывания. В Р-режиме прерывания и исключения можно разделить на 3 группы:

- **нарушение или сбой** (***fault***) - в стек записывается адрес команды, вызвавшей нарушение. Нарушение обрабатывается операционной системой, после чего можно осуществить рестарт ошибочной команды.
- **ловушка** (***trap***) - в стек записывается адрес команды, следующей за той, которая вызвала данную ловушку. Рестарт команды, вызвавшей прерывание или исключение типа ловушки, выполняется сложнее, так как от адреса в стеке нужно вычесть длину “плохой” команды. В случае, если осуществлен переход по jmp, такой возврат невозможен.
- **авария** - это ошибки аппаратуры, неверные значения в системных таблицах. При аварии контекст программы теряется и ее невозможно выполнять.

Возможны 3 типа шлюзов, различающихся по источнику, вызвавшему прерывание, и по передаче управления в программу обработки прерывания:

1. **Шлюз ловушки**. Шлюз содержит селектор, указывающий на дескриптор в таблицах ***GDT*** или ***LDT***, смещение в сегменте, минимальный уровень привилегий задачи, которая может передать управление ПОП через данный шлюз. При возникновении прерывания, дескриптор которого имеет тип “шлюз ловушки” в стеке сохраняются ***`eflags`***, ***`cs`***, ***`eip`*** и управление передается обработчику. При возврате из ПОП нужно иметь в виду, что некоторые прерывания вырабатывают код ошибки и сохраняют его в стеке, поэтому этот код должен быть удален командой pop перед выполнением **`iret`**.
2. **Шлюз прерывания.** Обработка шлюза прерывания отличается от обработки шлюза ловушки тем, что процессор сбрасывает **IF** в ***eflags*** перед передачей управления ПОП.
3. **Шлюз задачи** - используется для переключения между задачами.

4.7 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Классификация прерываний.
2. Назначение регистров контроллера прерываний.
3. Зачем в конце обработчика аппаратного прерывания нужно записывать 20h в порт 20h?
4. От чего зависит получение управления обработчиком аппаратного прерывания?
5. Поясните оптимальную структуру обработчика аппаратного прерывания.
6. Что такое scan-код?
7. Понятие многобайтового scan-кода и его обработка.
8. От чего зависит формирование ASCII-кода ?
9. Что такое расширенный ASCII-код ?
10. Как организован буфер клавиатуры ?
11. Содержимое указателей на голову и хвост буфера клавиатуры.
12. Способы активизации TSR-программ.

4.8 УПРАЖНЕНИЯ

Задание 1. *Написать программу, которая позволяет по истечении пяти секунд после ее запуска замаскировать линию **IRQ1** на 5 секунд, а затем снять маскирование.*

Для демонстрации работы программы реализуем циклический вывод введенных пользователем символов на экран с использованием функции 01h прерывания int 21h. Завершение цикла наступит по истечению 15 секунд от момента запуска программы. Отсчет секунд реализуем через обработчик 1Ch, 5 секунд составляют 91 тик таймера. Для определения текущего периода из 15 секунд будем использовать вспомогательную переменную f. Будем считать, что на протяжении первых 5 секунд переменная f равна нулю, на протяжении вторых 5 секунд f равна 1 (в этот момент прерывания от клавиатуры замаскированы и нажатие пользователем клавиш не будет обработано), на протяжении третьих 5 секунд f равна 2 (линия ***IRQ1*** размаскирована). После истечения 15 секунд переменной f присваивается значение 3.

```
.model tiny
segm segment para public 'code'
assume cs:segm, ds:segm, ss:segm, es:segm
org 100h
start: jmp beg
m1 db ':','$'          ;приглашение ко вводу символов
```

```
  nl db 13,10,'$'          ;переход на новую строку
  f db 0                   ;флажок для определения временного
                           ;интервала
  tic db 0                 ;переменная для подсчета количества тиков
  oldadr dd ?              ;адрес старого обработчика int 1Ch
  mytime proc far          ;наш обработчик прерывания 1Ch
  inc cs:tic               ;при входе в обработчик увеличим
                           ;количество тиков
  cmp cs:tic,91            ;сравним количество тиков с числом
                           ; 91, соответствующим 5 секундам
  jne quit                 ;если 5 с не прошло, то выход из ПОП
  mov cs:tic,0             ;иначе обнулим количество тиков,чтобы
                           ;можно было отсчитать следующие 5 с
  inc cs:f                 ;увеличим значение f (теперь оно
                           ; равно 1,2 или 3)
  cmp cs:f,1               ;сравним f с 1 для определения промежутка
  jne not_mask             ;если не 1, то f равно 2 или 3, что
;соответствует периоду, когда клавиатура не замаскирована,
;перейдем на метку not_mask
  in al,21h                ;иначе замаскируем линию IRQ1, считав
  or al,02h                ;из регистра маски и записав 1 в бит,
                           ;соответствующий IRQ1
  out 21h,al               ;выведем новое содержимое регистра
                           ;маски в порт
  jmp quit                 ;перейдем на выход из обработчика
  not_mask:
  in al,21h                ;размаскируем линию IRQ1, установив в
  and al,0fdh              ;регистре маски нужный бит в 0
  out 21h,al               ;выведем в порт новое содержимое
                           ;регистра маски
  quit: iret
  mytime endp
;начало основной программы, где устанавливается наш
;обработчик и организуется цикл вывода символов на экран,
;иллюстрирующий маскирование линии IRQ1
beg:
  push cs                  ;установим ds явно на наш сегмент
  pop ds
  mov ax,351ch             ;получим адрес старого обработчика1Ch
  int 21h
  mov word ptr oldadr,bx ;сохраним смещение старого int
```

```
                          ;1Ch по адресу oldadr
  mov word ptr oldadr+2,es ;сохраним сегмент старого int 1Ch
                          ;по адресу oldadr+2
  mov dx,offset mytime    ;смещение нового обработчика
                          ;запишем в dx
  mov ax,251ch            ;вызовем функцию установки нашего
                          ;обработчика
  int 21h
  mov ah,09h              ;выведем строку, реализующую
  mov dx,offset nl        ;переход на новую строку
  int 21h
  mov dx, offset m1       ;в dx поместим смещение строки-
                          ;приглашения ко вводу и
  int 21h                 ;выведем эту строку на экран
cycl: mov ah,1            ;в цикле вызываем функцию 01h
  int 21h                 ;int 21h, которая вводит символы
                          ;с клавиатуры и выводит их на экран
  cmp f,3                 ;проверим, не истекли ли 15 с
  je vosst                ;если да, то переход на
;восстановление в векторе прерывания адреса старого обработчика
  jmp cycl                ;если не прошло 15 секунд, то
                          ;продолжим цикл
  vosst: mov dx,word ptr oldadr; в dx – смещение старого
                          ;обработчика
  mov ax,word ptr oldadr+2    ;в ds – сегментный адрес
  mov ds,ax               ; старого обработчика
  mov ax,251ch            ; установим адрес старого
                          ;обработчика обратно
  int 21h                 ;в вектор прерываний
  mov ax,4c00h            ;выход
  int 21h
segm ends
end  start
```

Пример 2. *Написать обработчик прерывания от клавиатуры, который при каждом нажатии клавиши реагирует на нее только один раз*

Факт нажатия клавиш будем запоминать в массиве из 128 элементов, соответствующих scan-коду нажимаемых клавиш. Если в массиве значение элемента, соответствующего некоторому scan-коду, равно 1, значит, клавиша уже нажималась. Для

удобства отладки программы при нажатии ESC будем обнулять наш массив, «забывая» тем самым все нажатия.

```
jumps ; для генерации «длинных» переходов
segm segment para public 'code'
assume cs:segm,ds:segm
org 100h
start:jmp init
mas db 128 dup (0)        ; массив для хранения нажатий
old09 dd ?                ; адрес старого обработчика
sign dw 0fedch            ; сигнатура
Int09 proc far
      sti
      push ax
      push bx
      push dx
      push di
      push es
      mov ax,40h
      mov es,ax
      in al,60h           ; считаем scan-код из порта
      mov ah,al           ; проверим, что это не управляющая
      IRP X,<1DH,2AH,36H,3BH,3CH,3DH,3EH,3FH,40H,41H,42H,43H,44H>
      cmp ah,X            ; клавиша, и, если да, проигнорируем
      je ignor            ; для этого используем циклическую
      ENDM                ; макрогенерацию
      cmp ah,01h          ; если нажата ESC
      je sbros            ; перейдем на метку sbros
      cbw                 ; расширим scan-код до слова
      lea bx,cs:[mas]     ; загрузим в bx адрес начала массива
      mov si,ax           ; в si scan-код – индекс в массиве
      mov dh,byte ptr cs:[bx][si]  ; из массива в dh поместим
      cmp dh,0            ; флаг нажатия и сравним его с 0
      jne ignor           ; если нажата клавиша – ее игнорируем
      mov byte ptr cs:[bx][si],1   ; иначе установим флаг
      jmp pass      ; и перейдем к стандартному обработчику
ignor:
      in al,61h           ; подтвердим обработку scan-кода
      mov ah,al
      or al,80h
      out 61h,al
```

```
      mov al,ah
      out 61h,al
      cli
      mov al,20h          ; пошлем EOI в контроллер прерываний
      out 20h,al
      pop es
      pop di
      pop dx
      pop bx
      pop ax
      iret
sbros:
      lea bx,cs:[mas]     ; в bx загрузим адрес начала массива
      mov si,0
      REPT 128            ; и сбросим флаги нажатий всех
      mov byte ptr cs:[bx][si],0   ; клавиш
      inc si
      ENDM
pass:
      pop es
      pop di
      pop dx
      pop bx
      pop ax
      jmp cs:[old09]      ; передадим управление стандартному
Int09 endp                ; обработчику
init:
    mov ax,3509h          ; проверим адрес обработчика int 09h
    int 21h
    mov ax,es:[bx-2]      ; это наш обработчик в памяти?
    cmp ax,cs:sign
    je outmem             ; если наш – на выгрузку
    mov word ptr old09,bx     ; системный – сохраним его
    mov word ptr old09+2,es   ; адрес в двойном слове old09
    mov bx,cs
    mov ds,bx             ; установим ds на наш сегмент
    mov dx,offset int09   ; в dx поместим адрес нашей ISR
    mov ax,2509h          ; и запишем адреса в вектор
    int 21h               ; прерываний
    lea dx, init
    int 27h               ; оставим резидентно
```

```
outmem:                  ; выгрузка
    mov dx, word ptr es:old09
    mov bx, word ptr es:old09+2
    mov ds,bx
    mov ax,2509h         ; запись в вектор прерываний
    int 21h              ; адреса стандартного обработчика
    push es
    mov es,es:[2Ch]      ; выгрузка окружения
    mov ah,49h
    int 21h
    pop es
    mov ah,49h           ; выгрузка программы
    int 21h
    int 20h
segm ends
end start
```

Пример 3. *Написать обработчик прерывания от клавиатуры, инвертирующий действие клавиши Caps Lock.*

В данном случае проще не анализировать и самостоятельно обрабатывать scan-коды нажимаемых клавиш, а по максимуму использовать стандартный обработчик, который записывает в буфер клавиатуры scan и ascii-коды. Нам достаточно только проверить состояние клавиши CapsLock и скорректировать данные в буфере клавиатуры.

```
old_9int  dd ?       ;для хранения адреса обработчика старого
                     ;прерывания 09h
;Обработчик прерывания
new_9int proc far
     push ax         ;сохраняем регистры в стеке
     push bx
     push di
     push es
     sti             ;разрешить прерывания
     pushf           ;для iret в системном прерывании
     call cs:[old_9int]  ;вызов системного прерывания
     mov ax,40h      ;устанавливаем регистр es на область
                     ;данных BIOS
     mov es,ax
     mov di,es:[1ch];получаем указатель на голову
     cmp di,01Eh     ;он указывает на начало буфера?
```

```
     jne minus2      ;если нет, то для доступа к только
     mov di,03Ch     ;что помещенным в буфер scan и ascii-
;кодам необходимо отнять 2 от нового "хвоста"
     jmp el          ;иначе старый "хвост" указывал на
     minus2: sub di,2    ;конец буфера, что и запишем в di
                     ;и перейдем на el.
el:  mov ax,es:[di] ;считаем scan код из буфера клавиатуры
     mov bl,01000000b    ;запишем в bl для test строку с
                     ;установленным 6 битом
     test es:[17h],bl    ;в слове флагов клавиатуры есть
                     ;признак нажатия CapsLock
     jz Caps_NO      ;если 0 , то CapsLock не нажата
     cmp al,41h      ;проверка, что буква от A до Z
     jb do           ;если нет , то выход – на метку do
     cmp al,5ah
     ja do
     add al,32       ;если буквы большие, то прибавим к
     mov es:[di],ax ;ascii-коду 32, получим маленькую
                     ;букву и запишем в буфер
     jmp do          ;перейдем на конец
Caps_NO:             ;Caps Lock не нажат
     cmp al,61h      ;проверка что буква в буфере от a до z
     jb do           ;если нет , то переход на метку do
     cmp al,7ah
     ja do
     sub al,32       ;отнимем от ascii-кода 32 - получим
     mov es:[di],ax ;большую букву и запишем в буфер
do:  cli             ;запретить прерывания
     mov  al,20h     ;выдаем сигнал о завершении
     out  20h,al     ;аппаратного прерывания
     pop  es         ;восстанавливаем регистры из стека
     pop  di
     pop  bx
     pop  ax
     iret            ;возврат из прерывания
new_9int endp
```

Варианты :

1. Написать резидентную программу, которая при нажатии клавиши пробел позволяет циклически заменять пробел на цифры от 0 до 9.

2. Написать резидентную программу, которая первые 5 секунд не реагирует на нажатие клавиш, вторые 5 секунд работает нормально, и третьи 5 секунд реагирует только на нажатие пробела.
3. Резидентная программа, которая позволяет с интервалом времени в 1 секунду блокировать цифровую клавишу. После выполнения блокировки всех цифр их можно восстановить нажатием Ctrl-Alt-Del.
4. Написать резидентную программу, обработчик прерывания от клавиатуры в которой работает в 2 режимах: в первом режиме обработка нажатия клавиш осуществляется стандартным обработчиком, во втором – блокированы все функциональные клавиши. Переключение между режимами – нажатие Ctrl-Alt-Del.
5. Резидентная программа, в которой реализована замена нажатия на цифровые клавиши 0-9 на основной клавиатуре на нажатие соответственно функциональных клавиш F10, F1, F2, … , F9.
6. Резидентная программа, в которой нажатие клавиш второй линейки клавиатуры (qwertyuiop) заменяется на нажатие цифр 1234567890.
7. Написать резидентную программу, которая функционирует в цикле следующим образом: в течение 5 секунд не реагирует на функциональные клавиши, в течение 10 секунд не реагирует на буквы “a” и “o”.
8. Резидентная программа, которая заменяет вторую линейку клавиатуры (qwerty…) на пробел, а цифры заменяются на букву “A”.
9. Резидентная программа, которая при нажатии клавиш из второй линейки клавиатуры (qwerty…) заменяет их на цифры, соответствующие порядковому номеру клавиши в этой линейке.
10. Резидентная программа, в которой реализуется замена клавиши пробел на буквы от A до Z в цикле.
11. Резидентный обработчик прерывания от клавиатуры, который первые 5 секунд из каждых 10 не реагирует на нажатие Ctrl-Alt-Del.
12. Резидентная программа, в которой при нажатии PrtScr буквы нижней линейки (zxcvbnm,./) меняются с буквами верхней линейки (qwertyuiop).
13. Написать резидентную программу, в которой запоминать количество нажатий цифровых клавиш на основной клавиатуре. Если это количество нажатий превысит заданное число N, то перейти в режим игнорирования цифр.

4.9 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Сигнал EOI необходим для того, чтобы:

а) сбросить в регистре запросов бит, соответствующий нужной линии IRQ, тем самым подтвердив факт обслуживания запроса;

б) послать процессору сигнал завершения прерывания, по которому он аппаратными способами реализует возврат к прерванной программе;

в) сбросить нужный бит в регистре обслуживаемых запросов и разблокировать схему приоритетов

г) сигнал EOI необходимо посылать в контроллер перед завершением обработчика прерываний любого типа, как программного, так и аппаратного;

д) размаскировать соответствующую обслуживаемому запросу линию IRQ.

2. В момент выполнения обработчика прерываний от клавиатуры возникло прерывание от таймера. При этом дальнейшие события в системе развиваются следующим образом:

1) независимо ни от каких условий прерывание от таймера будет обслужено, прервав выполнение обработчика от клавиатуры;

2) если линия IRQ0 не замаскирована, то прерывание от таймера не будет обслужено до тех пор, пока не отработает прерывание от клавиатуры, если в обработчике прерываний от клавиатуры отсутствует команда sti ;

3) если в регистре маски контроллера прерываний разрешены прерывания от таймера и в начале обработчика прерываний от клавиатуры есть команда sti, то прерывание от таймера будет обслужено;

4) если линия IRQ0 не замаскирована, то прерывание от таймера обязательно будет обслужено, прервав выполнение обработчика от клавиатуры;

5) при любых условиях прерывание от клавиатуры будет обслужено, после завершения которого будет обслуживаться прерывание от таймера.

При этом верными сценариями работы будут: а – 1; б – 5; в – 2,4; г – 3,4; д – 2,3

3. Какое из нижеприведенных высказываний является верным:

а) номер приоритета аппаратного прерывания хранится в регистре обслуживаемых запросов;

б) приоритет аппаратных прерываний обычно уменьшается с ростом номера линии IRQ, при этом приоритет линий, подключенных к ведомому контроллеру, меньше приоритета IRQ1, но больше приоритета IRQ3;

в) приоритет аппаратных прерываний обычно уменьшается с ростом номера линии IRQ, при этом приоритет линий, подключенных к ведомому контроллеру, является самым низким по сравнению с остальными линиями IRQ;

г) более приоритетные прерывания всегда способны прервать выполнение обработчиков менее приоритетных;

д) приоритеты линий IRQ соответствуют их номерам линий.

4. Обработчик прерываний 1Ch нужен для того, чтобы:

а) подсчитывать тики таймера, не портя системных часов;

б) прерывать выполнение программы пользователя;

в) программист мог изменить количество тиков таймера в секунду на произвольное значение

г) программист мог узнать показания системных часов

5. Какое из следующих утверждений является верным:

а) буфер клавиатуры организован в виде циклического односвязного списка, причем указатель на хвост служит для удаления элементов из буфера;

б) указатель на голову в буфере клавиатуры служит для отслеживания последнего записанного в буфер элемента;

в) указатель на хвост буфера клавиатуры хранит адрес последнего элемента, записанного в буфер;

г) указатель на хвост буфера клавиатуры хранит адрес первой свободной ячейки буфера за последним занесенным в буфер элементом с учетом цикличности;

д) буфер клавиатуры организован в виде циклического двусвязного списка.

6. Расширенный ASCII-код представляет собой:

а) код, считываемый из порта 60h;

б) код, состоящий из 2 байтов, один из которых содержит 0, а второй – специальный код, который может совпадать со scan-кодом нажатой клавиши;

в) код, состоящий из двух байтов, один из которых – scan-код, второй – ascii-код символа, соответствующего нажатой клавише;

г) ASCII-код символа, увеличенный на некоторое число;

д) scan-код нажатой клавиши.

Ответы: 1 – в, 2 – д, 3 – б, 4 – а, 5 - г,6 - б.

5 АППАРАТНЫЕ ОСНОВЫ ВЫВОДА ГРАФИКИ И ТЕКСТА

5.1 ОБЩИЕ СВЕДЕНИЯ О ВИДЕОСИСТЕМЕ

Видеосистема персонального компьютера состоит из двух главных компонентов: монитора (дисплея) и видеоадаптера (видеоплаты или графической платы). Сигналы, которые управляют работой монитора, поступают от электронных схем, размещенных внутри компьютера. В некоторых компьютерах эти схемы располагаются на материнской плате. Однако в большинстве систем используются отдельные платы, которые вставляются в слоты системной шины или шины расширения. Такие платы расширения, вырабатывающие сигналы управления изображением, называются видеоадаптерами.

Информация на мониторе может отображаться несколькими способами. Например, для отображения может использоваться электронно-лучевая трубка, содержащая три электронных пушки (для трех компонент цвета) и экран, покрытый люминофором. Электронные пушки испускают потоки электронов, под воздействием ударов которых люминофор излучает свет. Электронный луч движется очень быстро, прочерчивая экран слева направо и сверху вниз по траектории, которая называется растром. Сканирование экрана лучом обеспечивается генераторами горизонтальной и вертикальной разверток монитора. Луч может оставлять след в виде светящихся точек (пикселов) только во время прямого хода по строке (слева направо). Когда луч пробежит от начала до конца горизонтальной строки, он перемещается в начало следующей строки, это перемещение называется обратным горизонтальным ходом луча. Если луч не выключается при этом перемещении, то по краям экрана наблюдаются цветные выбеги развертки за счет того, что время, нужное для обратного горизонтального хода луча, меньше, чем период между окончанием отображения верхней строки и началом отображения нижней. После пробега всех горизонтальных строк луч снова перемещается в верхний левый угол экрана, это перемещение называется обратным вертикальным ходом луча. Процесс постоянного возобновления изображения называется *регенерацией*.

В настоящее время вместо мониторов на электронно-лучевой трубке используются жидкокристаллические мониторы или LCD (Liquid Crystal Display). Основой LCD-дисплея является матрица из жидкокристаллических ячеек. Каждая ячейка свободно пропускает свет, если его плоскость поляризации параллельна оптической оси стержнеобразных молекул в ячейке. Под воздействием электрического заряда молекулы в ячейке могут изменять свою

ориентацию, что приводит к изменению яркости ячейки. Для цветных LCD-мониторов каждому пикселу изображения соответствует три ячейки – для красного, зеленого и синего цветов. В LCD-дисплеях с пассивной матрицей яркостью каждой ячейки управляет электрический заряд, который протекает через транзисторы, номера которых равны номерам строки и столбца данной ячейки в матрице экрана. Так, при разрешении 800x600 матрица управляется 1400 транзисторами. На жидкокристаллические ячейки дисплея с пассивной матрицей подается пульсирующее напряжение. В LCD-дисплеях с активной матрицей каждая ячейка управляется своим транзисторным ключом. Количество транзисторов при том же разрешении составит 480000, что удорожает монитор, но зато позволяет получить более качественное изображение.

Классическими компонентами видеосистемы являются:

1) BIOS
2) графический адаптер
3) видеопамять
4) цифроаналоговый преобразователь DAC Digital to Analog Converter
5) видеодрайвер для работы с мультимедийными изображениями.

Видеоадаптеры имеют свою BIOS, которая подобна системной, но полностью независима от нее. BIOS видеоадаптера хранится в микросхеме ROM. Она содержит основные команды, которые предоставляют интерфейс между оборудованием видеоадаптера и программным обеспечением. Обращаться к функциям BIOS видеоадаптера может автономное приложение, операционная система или системная BIOS.

Для подключения дисплея к графическому адаптеру компьютера используются специализированные/интерфейсы, по которым передается информация о мгновенном значении яркости базисных цветов и сигналы строчной и кадровой синхронизации. Первые адаптеры (например, EGA) имели цифровой интерфейса, затем в адаптере VGA применен аналоговый интерфейс, после чего произошел возврат к цифровому способу (DVI - Digital Visual Interface). Видеоинтерфейсы используются для вывода информации на обычные телеприемники и телевизионные мониторы, а также ввода видеоданных в компьютер. Видеоданные в цифровом виде могут передаваться и приниматься по шине Fire Wire, а также по USB.

С точки зрения способа формирования изображения считается, что видеосистема компьютера состоит из графической подсистемы, формирующей изображение программным путем, и подсистемы обработки видеоизображений. Средства для работы с видеоизображениями, передаваемыми в форматах PAL, SECAM, NTSC относятся к мультимедийному оборудованию и оперируют с изображением, поступающим в компьютер

извне или воспроизводимыми с CD-ROM. Графическая подсистема реализуется с помощью графического адаптера, служащего для программного формирования графических и текстовых изображений и являющегося промежуточным элементом между монитором и шиной компьютера.

5.2 ОСНОВНЫЕ КОМПОНЕНТЫ ГРАФИЧЕСКОГО АДАПТЕРА

Роль графического адаптера в видеосистеме может различаться. В простейшем случае графический адаптер отвечает только за хранение и регенерацию изображения на мониторе, а само изображение строится по программе, выполняемой центральным процессором. При этом на ЦП ложится огромная нагрузка, поскольку он должен полностью управлять построением всех деталей изображения. В BIOS существуют функции поддержки формирования текстовых и графических изображений (видеосервис BIOS int 10h), но они работают очень медленно (медленнее в 100 раз по сравнению с прямым выводом в видеопамять), и поэтому для непосредственной работы с изображением практически не используются.

В современной компьютерной графике применяется архитектура видеоадаптера, основанная на использовании специализированного графического сопроцессора, который выполняет вычисления для построения изображения. Существует промежуточный вариант архитектуры – видеоакселератор с ограниченным набором функций. Эта архитектура предполагает, что электронные схемы видеоадаптера решают алгоритмически простые, но отнимающие много времени задачи. В частности, электронные схемы видеоадаптера выполняют построение графических примитивов – прямых линий, окружностей и т.п., а за центральным процессором компьютера остается конструирование изображения, разложение его на составляющие и пересылка в видеоадаптер инструкций, например, нарисовать прямоугольник определенного размера и цвета.

Рассмотрим основные компоненты графического адаптера и их назначение.

В **видеопамяти** размещаются данные, отображаемые монитором. Появление изображения на экране связано с тем, что видеоадаптер циклически осуществляет сканирование видеопамяти. Логически видеопамять может быть организована по-разному, как линейная или как многоплоскостная в зависимости от режима работы графического адаптера и количества битов, соответствующих одному пикселу – простейшему элементу изображения.

Видеопамять VGA организована, например, как четыре 64-килобайтовые плоскости. Каждая плоскость представляет собой линейный битовый образ байтов, отображаемых на

экране, т.е. каждый байт плоскости соответствует 8 пикселам, расположенным на экране рядом. Адресация битовых плоскостей зависит от режима, все 4 битовые плоскости могут адресоваться как один сегмент памяти. При этом каждому адресу видеопамяти соответствует 4 байта (по одному байту для каждой плоскости или слоя). За один цикл обращения к видеопамяти возможна запись во все 4 слоя, чтение же возможно только из одного слоя. Слои видеопамяти по-разному используются в текстовых и графических режимах работы видеоадаптера, ее начальный адрес для текстового режима B800h:0000h, для графического A000h:0000h. Во время выполнения центральным процессором чтения байта из видеопамяти одновременно происходит запись байтов из всех слоев в так называемые **регистры-защелки** (latch-registers). Каждому слою соответствует один 8-битовый регистр-защелка.

Традиционно для видеопамяти была выделена область адресов A0000h-BFFFFh, доступная процессору x86. Количество битов, отведенное для хранения информации об одном пикселе изображения, может быть различным, и в соответствии с этим по-разному организуется видеопамять. В случае одного или двух битов на пиксел каждый байт видеопамяти соответствует восьми или четырем соседним пикселам строки. Такой способ отображения называется линейным, т.к. линейной последовательности пикселов соответствует линейная последовательность битов видеопамяти. Однако разработчики адаптера EGA при увеличении количества битов на пиксел до 4 разбили видеопамять на 4 слоя или видеоплоскости, называемые также цветовыми плоскостями. В каждом слое используется линейная организация, где каждый байт содержит по одному биту для восьми соседних пикселов. При сканировании видеопамяти 4 байта по одному из каждого слоя считываются в 4 регистра-защелки. При этом байты из всех видеоплоскостей имеют один и тот же адрес. Это в принципе позволяет осуществлять запись в несколько цветовых плоскостей одновременно, конкретно же возможность записи в каждый слой зависит от режима записи **графического контроллера** и от содержимого некоторых регистров **синхронизатора**. Решение о многоплоскостной модели видеопамяти позволило снизить частоту считывания видеопамяти, однако для программиста, работающего с видеопамятью напрямую, значительно затруднило работу.

При использовании для представления пиксела 15, 16 (режим High Color) или 24 битов (True Color) используется только линейная организация видеопамяти. При организации соответствия одному пикселу 8 битов используется как многоплоскостная, так и линейная организация видеопамяти. Для 32 битов на пиксел в режиме с разрешением 1280x1024 пиксела требуется 5 Мб видеопамяти без использования 3D – акселератора. Для организации работы с линейным образом одного экрана современные графические адаптеры

имеют возможность адресации видеопамяти в области адресов, превышающих 16 Мб в защищенном режиме.

Содержимое таблицы регистров ЦАП влияет на вывод изображения только в 256-цветных графических режимах. В режимах Direct Draw (HighColor и TrueColor) информация из видеопамяти поступает непосредственно на ЦАП. При этом используется линейная организация видеопамяти, минимизирующая вычисления. Количество битов, соответствующее цвету для 1 пиксела, составляет 15, 16, 24 или 32 бита. Согласно схеме прямого кодирования цвета биты для пиксела разбиваются на 3 группы для красной, зеленой, синей компоненты цвета, которые передаются на ЦАП. В режимах HighColor пиксел кодируется 2 байтами, причем допустим режимы HighColor15 и HighColor16, в которых красной, зеленой и синей компонентам соответствуют распределения 1:5:5:5 и 5:6:5, старший бит в первом случае не используется. В режимах TrueColor для хранения каждого компонента цвета точки выделено по одному байту видеопамяти. Существует 2 формата представления данных в TrueColor: TrueColor24 и TrueColor32, причем дополнительный байт добавлен в 32-битный режим только для выравнивания, так как процессор может передавать данные контроллеру только байтами, словами и двойными словами. В режиме TrueColor24 приходится поэтому выполнять побайтовую пересылку информации, что в три раза снижает скорость передачи. В режиме TrueColor32 было решено пожертвовать лишним байтом для каждого пиксела, но увеличить скорость.

Рассмотрим компоненты графического адаптера. Для управления обменом информацией между видеопамятью и центральным процессором служит **графический контроллер. Синхронизатор** служит для синхронизации временных циклов обращения к видеопамяти с процессом регенерации изображения на экране. В соответствии со структурой видеопамяти ее байты преобразуются последовательным преобразователем в биты. **Контроллер атрибутов** служит для формирования по этой последовательности битов цвета на экране. Все компоненты видеоадаптера имеют свои наборы регистров, доступ к которым осуществляется через собственные порты. Для синхронизатора и графического контроллера доступ осуществляется через два смежных порта. Первый порт (так называемый индексный порт) является адресным и в него записывается номер регистра, к которому нужно обратиться, а во второй порт (порт данных) записываются данные. К регистрам контроллера атрибутов доступ осуществляется через один порт.

Например, для синхронизатора индексный порт - ***3C4h***, порт данных - ***3C5h***. Во второй регистр записываем число 4:

```
mov  dx,3C4h
mov  al,02h
out  dx,al
```

```
inc  dx
mov  al,04h
out  dx,al
```

Если учесть, что при последовательной адресации портов запись в порт сходна с записью в память, то можно использовать более короткую запись

```
mov  dx, 3C4h
mov  ax, 0402h
out  dx, ax
```

В регистре ***ax*** в ***al*** записываем номер регистра, в ***ah*** - данные. Так как младший байт в памяти располагается перед старшим, то по адресу ***3C4h*** будет записано ***02h***.

Для графического контроллера индексный порт и порт данных - ***3CEh*** и ***3CFh.*** Для контроллера атрибутов индексным портом и портом данных порт является ***3C0h***. В каждый момент записываемые в порт данные могут восприниматься либо как номер регистра, либо как данные, записываемые в регистр, в зависимости от состояния внутреннего триггера. Этот триггер можно установить в исходное состояние чтением из порта ***3DAh***. После этого данные, записываемые в ***3C0h***, будут восприниматься как номер регистра. Каждая последующая запись в порт переключает триггер.

5.3 УСКОРЕНИЕ ОБРАБОТКИ ВИДЕОИЗОБРАЖЕНИЙ

Для решения многих задач с использованием компьютера необходима высококачественная графика. Изображение такого качества требует вывода на экран большого количества пикселов. Но сначала цвет каждого пиксела необходимо сформировать и записать в видеопамять. Оттуда информация должна пересылаться на монитор с такой скоростью, чтобы экран обновлялся по меньшей мере 30 раз в секунду. Вычисление цвета пикселов может выполняться программно и записываться в видеобуфер, однако при этом объемы обрабатываемых данных будут настолько велики, что, если возложить их обработку только на графический контроллер и процессор, то у процессора просто не останется времени для выполнения других задач. Кроме того, использование шины компьютера для пересылки содержимого видеобуфера на дисплей приведет к тому, что шина также почти полностью будет занята этими данными. Если один пиксел занимает 32 бита, то для изображения 1024x1024 пиксела нужно 4 Мбайта, и для его пересылки потребуется шина со скоростью передачи не менее 120 Мбайт/с.

Кроме того, в большинстве графических приложений на экран выводятся трехмерные объекты. В частности, в компьютерных играх создается искусственный трехмерный мир с видеоизображениями, формируемыми программным путем. В компьютерной графике

трехмерный объект представляется в виде поверхности, состоящей из большого количества маленьких многоугольников (как правило, треугольников). Основной задачей графической обработки является преобразование трехмерного изображения в двумерное, максимально близкое к тому, как оно видится человеческим глазом. Для определения проекции и перспективы объектов требуется вычислять местоположения вершин треугольников, далее с помощью сложных алгоритмов создания реалистичного изображения вычисляются цвета и тени каждого треугольника. При этом учитывается расположение источника света, его отражение от различных поверхностей, тени и т.п. В случае движущихся изображений эти вычисления повторяются по много раз в секунду. Важной частью данного процесса является формирование определенной текстуры поверхности, например, древесных волокон или кирпичной кладки. Скрытые части изображения удаляются путем отсечения, и вычисляется цвет каждого пиксела, отправляемого на монитор.

Если выводом графики управляет процессор, то канал связи процессора с видеопамятью является узким местом для огромного потока битов. Имеется несколько способов решения данной проблемы.

Во-первых, возможно расширение разрядности внутренней шины видеоадаптера для работы с видеопамятью и его внешней интерфейсной шины, применение высокопроизводительных шин (PCI и AGP). Расширение разрядности позволяет за один цикл обращения передать большее количество данных. Во-вторых, повысить скорость видеопостроений можно с помощью кэширования видеопамяти. В этом случае при записи в область видеопамяти запись будет произведена не только в видеопамять, но и в кэш, а при чтении из этой области видеопамяти будет читаться только кэш. В третьих, можно передать часть функций стандартного графического адаптера специализированному процессору, способному формировать растровое изображение в видеопамяти по командам, полученным от центрального процессора. В этом случае говорят об интеллектуальном графическом адаптере (ИГА). Рассмотрим некоторые его возможности.

Интеллектуальный графический адаптер строит изображения из примитивов более высоких уровней, чем пиксел. Примерами команд, обрабатываемых ИГА, являются команды рисования, копирования блоков, команды работы со спрайтами (прямоугольный фрагмент изображения, рассматриваемый как отдельное целое). Команды рисования обеспечивают построение графических примитивов (точка, отрезок, прямоугольник, дуга, эллипс), описываемых в векторном виде, что сокращает объем передаваемой графической информации. К командам рисования относится также заливка замкнутого контура с известным цветом границы некоторым цветом или узором. При программной реализации заливки рассмотренными стандартными методами процессор читает цвет пикселов из

видеопамяти, начиная от некоторой точки внутри замкнутой области, и затем расширяет область поиска до обнаружения пикселов с заданным цветом границы. При этом требуется чтение большого объема байтов видеопамяти, их анализ и запись новых байтов обратно в видеопамять. Встроенный процессор графического адаптера выполняет эту операцию без выхода с потоком данных на внешнюю шину. Команды копирования блока с одного места экрана на другое сводятся к пересылке группы битов, также минуя внешнюю шину.

Для двумерной графики реализована аппаратная поддержка окон (Hardware Windowing), которая упрощает и ускоряет работу с экраном в многозадачных системах. При использовании стандартного графического адаптера необходимо отслеживать координаты обрабатываемых точек, чтобы не выйти за пределы своего окна. При аппаратной поддержке окон каждой задаче выделяется свое окно – область видеопамяти необходимого размера, в котором задача работает монопольно. Если взаимное расположение окон известно ИГА, то он может сканировать не всю видеопамять линейно, а только те ее области, которые относятся к активному окну.

Ускорение построений в ИГА обеспечивается несколькими факторами:

1) сокращение объема передачи данных по внешней шине;

2) наличие собственного процессора в ИГА, который ориентирован на выполнение небольшого количества инструкций и поэтому выполняет их быстрее универсального ЦП;

3) расширение разрядности внутренней шины графического адаптера.

Среди ИГА различают графические сопроцессоры и акселераторы. **Графический сопроцессор** представляет собой специализированный процессор, который подключается к шине компьютера и имеет доступ к его оперативной памяти. В процессе работы сопроцессор использует оперативную память и конкурирует с ЦП по доступу к памяти и к шине. **Графический акселератор** работает автономно и может не выходить на внешнюю шину с объемом обрабатываемых данных. Акселераторы являются традиционной частью всех современных графических адаптеров.

Различают 2D- и 3D-акселераторы. Акселераторы двумерных операций (2D-accelerators) необходимы для реализации графического интерфейса пользователя GUI (Graphic User Interface) в Windows, 3D-акселераторы выполняют трехмерные построения. Как отмечалось выше, трехмерное изображение состоит из ряда поверхностей различной формы, собранных из элементов-полигонов, чаще всего из треугольников, каждый из которых имеет трехмерные координаты вершин и описание поверхности (цвет, узор). При перемещениях происходит пересчет координат, учитывается освещенность объекта, прозрачность и т.п. При выполнении трехмерных построений графическому акселератору снова перестает хватать ограниченного объема памяти графического адаптера, и он снова

начинает использовать ОП компьютера, нагружая своими данными внешнюю шину. Для обеспечения простоты использования графическим акселератором основной памяти компьютера был разработан специальный канал связи графического адаптера с памятью – AGP (Accelerated Graphic Port).

Основная идея AGP заключается в предоставлении графическому процессору максимально быстрого доступа к системной памяти, причем более приоритетного, чем доступ к памяти других устройств. Порт AGP позволяет акселератору работать в двух режимах: DMA (Direct Memory Access) и DIME (Direct Memory Execute). В режиме DMA акселератор при вычислениях рассматривает видеопамять как первичную, а когда ее недостаточно, обращается к оперативной памяти. В режиме DIME акселератору доступна непрерывная область памяти, часть которой составляет его локальная память или локальный буфер. Остальная часть адресуемой им памяти отображается на системное ОЗУ с помощью специальной таблицы переопределения графических адресов GART (Graphics Address Remapping Table). В этой таблице каждой странице графической памяти ставится в соответствие своя страница системного ОЗУ. Механизм переадресации памяти через GART входит в состав логики порта AGP, его конкретная физическая реализация не стандартизована, ее должен знать драйвер порта AGP. Сама таблица GART может располагаться в системном ОЗУ.

Область физических адресов памяти, доступных акселератору, называется апертурой AGP. Ее размер задается программированием управляющих и конфигурационных регистров порта AGP, доступных как центральному, так и локальному графическому процессору, и может составлять 8, 16, 32, …, 256 Мбайт. Установка размера апертуры, равного объему локальной видеопамяти, фактически отключает режим DIME, поскольку размер GART становится нулевым. Оптимальное значение апертуры – половина ОЗУ. Благодаря 32 или 64-разрядной адресации, используемой в порте AGP, локальная память графического адаптера может быть отображена в область памяти, не используемую другими устройствами.

Локальная память акселератора включает в себя, как минимум, два буфера экрана (во время отображения одного буфера рисуется изображение во втором), Z-буфер, альфа-буфера, память для хранения текстур. Z-буфер представляет собой матрицу, элементы которой соответствуют пикселам экрана и хранят информацию о «глубине» расположения точки на плоском экране с учетом ее перекрытия другими элементами. Альфа-буфер хранит информацию о прозрачности объектов (альфа - это число от 0 до 1, 1 – полностью непрозрачный объект, 0 – полностью прозрачный). Коэффициент прозрачности хранится в 8 битах, часто этот байт дополняет 24-битную модель кодирования цвета до 32 битов, тогда такой формат видеопамяти называют RGBA.

Таким образом, для получения высококачественных графических изображений требуются очень сложные вычисления, которые целесообразно выполнять на отдельном специализированном процессоре. Такой процессор, называемый Graphics Processing Unit (GPU), является основой графических плат, установленных в большинстве персональных компьютеров. Кроме процессора, графическая плата содержит высокоскоростную память, которая используется графическим процессором для выполнения вычислений и хранения результирующего изображения, предназначенного для вывода на экран. Дисплей обменивается с графической платой информацией без помощи шины компьютера. Высококачественные графические платы могут обновлять экран со скоростью от 75 до 200 раз в секунду.

5.4 ОБЩИЕ СВЕДЕНИЯ О ТЕКСТОВОМ РЕЖИМЕ

В текстовом режиме экран разбивается на строки (стандарт 25строк), каждая из которых включает обычно 80 знакомест. Количество строк может быть увеличено. В каждом знакоместе может быть отображен символ (текст или псевдографика).

Как известно, информация об отображаемом символе хранится в видеопамяти. Ее начальный адрес для цветного текстового режима B800h:0000h.

Текстовый режим является классическим режимом вывода данных, который работает при минимальных требованиях к качеству отображения символов. Видеопамять в текстовом режиме является многоплоскостной, что соответствует стандартам EGA и VGA.

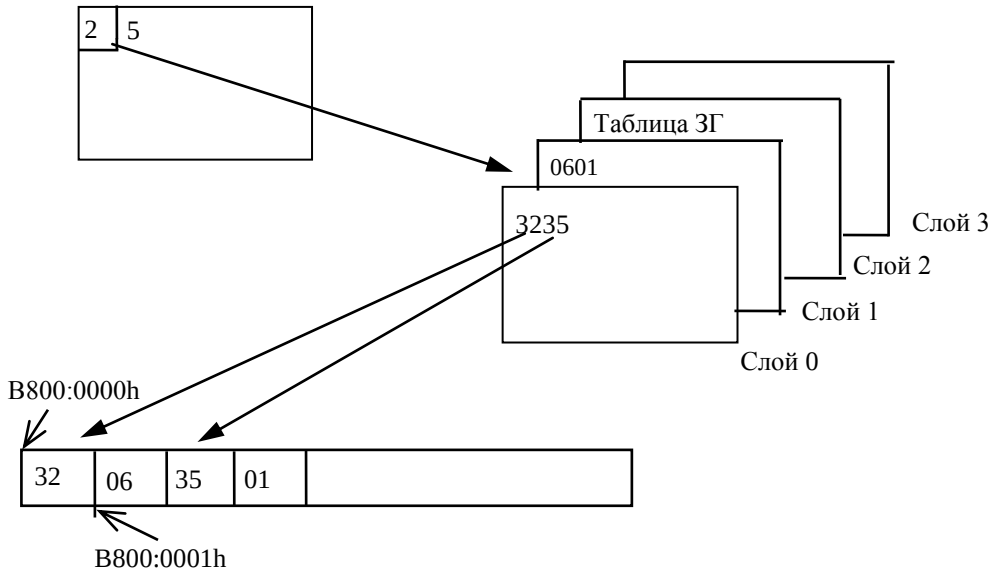


Рисунок 5.1 - Адресация слоев видеопамяти в текстовом режиме

Можно считать, что ASCII-код*32 - это смещение в таблице знакогенератора, задающее адрес начала битовой матрицы изображения. Например, рассмотрим изображение римской цифры 1:

```
char1 db 11111111b        ;всего16 строк в изображении
      db 00011000b
      ....                ;повтор строки 12 раз
      db 00011000b
      db 11111111b        ;последняя строка изображения
```

или:

```
char1 db 0FFh, 018h, 018h, 018h, 018h, 018h, 018h, 018h,
018h, 018h,018h,018h,018h,018h, 018h, 0FFh
```

Таблица знакогенератора хранится во втором слое видеопамяти и загружается туда при установке текстового режима работы из ПЗУ. Длина таблицы знакогенератора - 2000h. Видеоадаптеры позволяют одновременную загрузку в видеопамять нескольких таблиц знакогенератора (для VGA - 8 таблиц). Однако активными могут быть только две таблицы, что позволяет отображать одновременно 512 символов.

Какая именно из двух таблиц используется для отображения символа, указывается в байте атрибутов, имеющем следующую структуру:

биты 0-2 определяют цвет символа;

бит 3, если активна только одна таблица знакогенератора, также участвует в формировании цвета символа, если активны две таблицы, то этот бит задает таблицу знакогенератора для отображения;

биты 4-6 задают цвет фона;

бит 7 управляет либо мерцанием символа на экране, либо цветом фона, увеличивая количество цветов фона до 16.

Таким образом, 4 или 3 младших бита байта атрибутов хранят номер регистра палитры, определяющий цвет символа, а 4 или 3 старших бита служат для хранения номера регистра палитры, задающего цвет фона. Напомним, что контроллер атрибутов имеет в своем составе 16 регистров палитры (номера 0-15), в которых 6 значащих разрядов определяют интенсивность красного, зеленого и синего цветов, дающих при смешивании требуемый цвет (рисунок 5.2).

Рисунок 5.2.

Например, число 01h определяет синий цвет, число 63h - ярко-белый.

5.5 РАБОТА С ТАБЛИЦАМИ ЗНАКОГЕНЕРАТОРА

Стандартно видеопамять в текстовом режиме представлена как чередование байтов из 0 и 1 видеоплоскостей. Таблица знакогенератора (ТЗГ) находится во второй видеоплоскости,

поэтому для работы с ней необходимо получить к ней доступ путем изменения значений отдельных битов некоторых регистров синхронизатора и графического контроллера.

Рассмотрим структуру регистров, содержимое которых должно быть изменено для получения доступа к таблице знакогенератора.

1) Регистр № 2 синхронизатора - регистр разрешения записи цветового слоя. Биты 0,1,2,3 этого регистра при установке в 1 разрешают запись соответственно в 0,1,2,3 слои видеопамяти. Нули в указанных битах служат для запрещения записи в соответствующий цветовой слой. Исходное содержимое регистра 003h, т.е. можно писать в нулевой и первый слои в соответствии со стандартным отображением слоев видеопамяти на адресное пространство в текстовом режиме. Нам нужно записать 1 во второй бит, т.е. разрешить запись во 2 слой, а значит, в регистр должно быть записано число 004h.

Так как индексный порт у синхронизатора 3C4h, то это выполняется командами:

```
mov   dx,3C4h
mov   ax,0402h
out   dx,ax
```

Для восстановления старого содержимого регистра в порт 3C4h необходимо записать число 0302h.

2) регистр № 4 синхронизатора - регистр определения структуры памяти, имеет следующую структуру:

бит 0 - если 1, то установлен текстовый режим, если 0, то графический;

бит 1 - указывает на размер видеопамяти, 0 - объем видеопамяти 64Кб, 1 - объем превышает 64 Кб;

бит 2 - если 0, то по четным адресам осуществляется доступ к нулевому слою видеопамяти, а по нечетным - к первому; если 1, то используется последовательная адресация;

биты 3-7 - не используются, должны быть равны нулю.

В исходном состоянии в этом регистре записано число 003h, нам нужно для изменения шрифтов использовать последовательную адресацию, а значит, необходимо дополнительно установить 2 бит в 1, записав в порт данных число 07h:

```
mov   dx,3C4h    ; если этот адрес не загружен в dx
mov   ax,0704h
out   dx,ax
```

Для восстановления старого содержимого регистра в порт 3C4h необходимо записать число 0304h.

3) регистр № 4 графического контроллера - регистр выбора читаемого слоя, в битах 0 и 1 которого указывается номер читаемого слоя. В исходном состоянии номер

читаемого слоя - 0, если нам нужно читать содержимое таблицы знакогенератора, то в регистр выбора читаемого слоя запишем число 02h. Напомним, что индексный порт графического контроллера - ***3CEh***.

```
mov   dx, 3CEh
mov   ax, 0204h
out   dx, ax
```

Для восстановления старого содержимого регистра в порт 3CEh необходимо записать число 0004h.

4) регистр № 5 графического контроллера.

Нам нужно, чтобы адресация была последовательной для работы с ТЗГ, поэтому в регистр запишем 0:

```
mov   dx, 3CEh
mov   ax, 0005h
out   dx, ax
```

Для восстановления старого содержимого регистра в порт ***3CEh*** необходимо записать число 1005h.

5) регистр № 6 графического контроллера - регистр смешанного назначения - имеет следующую структуру:

бит 0 - 1- графический режим, запрещающий генерацию символов, 0 - текстовый режим (стандартно);

бит 1 - 1 - формировать из 4 слоев 2, 0 - не формировать;

биты 2,3 - определяют начальный и конечный адреса видеопамяти:

00 - A000:0000h - B000:FFFFh (объем видеопамяти ≥ 128Кб)

01 - A000:0000h - A000:FFFFh (объем видеопамяти <128Кб)

10 - B800:0000h - B800:7FFFh

11 - B800:0000h - B800:FFFFh

Стандартно в битах 2,3 записаны 11, т.е. в регистре записано число 0Eh. Нам надо, чтобы начальный адрес видеопамяти был A000:0000h и память была больше или равна 128 К, а следовательно, в регистр должен быть записан 0. Если объем видеопамяти невелик, то пишем в регистр число 04h.

```
mov   dx,3CEh
mov   ax,0006h
out   dx,ax
```

Для восстановления старого содержимого регистра в порт 3CEh необходимо записать число 0E06h.

После получения доступа ко второму слою видеопамяти с таблицей знакогенератора можно работать как с обычной памятью, установив для нее сегментный адрес A000h:

```
        mov  ax,0A000h
        mov  es,ax
```

Рассмотрим одновременное использование двух таблиц ТЗГ. Номера используемых таблиц хранятся в регистре № 3 синхронизатора. На номер таблицы знакогенератора указывает бит 3 байта атрибутов. Если этот бит равен 0, то при отображении символа используется таблица знакогенератора с номером, записанным в битах 0, 1, 4 регистра № 3 синхронизатора. Если 3 бит байта атрибутов равен 1, то при отображении символа используется таблица знакогенератора с номером, хранящимся в битах 2, 3, 5 регистра № 3 синхронизатора. Если используется только одна таблица, то содержимое битов 4, 1, 0 и 5, 3, 2 должно совпадать.

Соответствие между содержимым битов, номером таблицы знакогенератора и смещением в видеопамяти начала таблицы знакогенератора приведено в таблице 5.1:

Таблица 5.1

Содержимое битов	Номер таблицы	Смещение
000	1	0000h
001	2	4000h
010	3	8000h
011	4	C000h
100	5(VGA,SVGA)	2000h
101	6 (-//-)	6000h
110	7 (-//-)	A000h
111	8 (-//-)	E000h

Например, можно заполнить вторую таблицу знакогенератора, сдвинув влево на 2 бита изображения символов, взятые из стандартной таблицы:

```
        mov  cx,2000h
        mov  si,0
        mov  ax,0A000h
        mov  es,ax
zap:    mov  al,es:[si]          ;читаем очередной байт
        shl  al,2           ;сдвигаем его на 2
        mov  es:[si+4000h], al   ;пишем его в новую таблицу
        inc  si                  ;к следующему байту
        loop zap
```

Чтобы сделать новую таблицу доступной, необходимо выполнить команды:

```
        mov  dx,3C4h
        mov  ax,0403h
```

```
out  dx,ax
```

Для возврата к 1 таблице нужно в третий регистр синхронизатора записать 0h.

Ширина символа в текстовых режимах может быть 8 или 9 пикселов, однако в таблицах знакогенератора ширина символов всегда 8 битов. Поэтому 9 пиксел либо совпадает с цветом фоном, либо копирует 8 пиксел.

Стандартное значение высоты символов в текстовом режиме - 8,14, 16 пикселов. Для изменения высоты символов используется 9 регистр контроллера монитора. В младших 5 битах этого регистра хранится число, на 1 меньшее высоты символа. Высоту таким образом можно установить от 1 до 32. При установке нового значения высоты надо помнить, что старшие биты этого регистра должны остаться без изменения.

5.6 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Типы дисплеев.
2. Перечислить назначение основных компонент графического адаптера.
3. Формирование цвета в плоской модели видеопамяти.
4. Методы ускорения обработки графических изображений.
5. Назначение порта AGP.
6. Что такое видеопамять?
7. Структура видеопамяти в текстовом режиме.
8. Назначение таблицы знакогенератора.
9. Доступ к регистрам видеоадаптера.
10. Как формируется цвет фона и цвет символа в текстовом режиме?
11. Структура байта атрибутов.
12. Какая информация содержится в регистрах палитры?

5.7 УПРАЖНЕНИЯ

Задание 1. Изменить цвет всех латинских букв, выведенных на экран.

```
     .model small
     .code
     org  100h
begin:
; Настроим сегментный регистр es на начало видеобуфера
     mov  ax,0B800h      ;Загрузка сегментного адреса
     mov  es,ax          ;видеобуфера в es
```

```
     mov  cx,80*25        ;в cx - количество символов на экране
     mov  bx,0            ;в bx - начальное смещение
lodstor:
     mov  al,es:[bx]      ;считываем ASCII-код символа
     cmp  al,'A     '     ;проверим, латинская ли это буква?
     jb   next
     cmp  al,'z'
     ja   next
     cmp  al,'Z'
     jbe  ok
     cmp  al,'a'
     jb   next
ok:  mov  ah,es:[bx+1]    ;считаем атрибут
     and  ah,11110100b    ;изменяем атрибут
     mov  es:[bx+1],ah    ;запишем его обратно
next: add bx,2            ;переходим на следующий символ
     loop lodstor
     mov  ax, 4C00h       ;Выход
     int  21h
end  begin
```

Задание 2. Создать новую таблицу знакогенератора, которая получается сжатием в 2 раза в горизонтальном направлении символов стандартной таблицы. Реализовать переключение между таблицами через 2 секунды.

Для сжатия таблицы будем считывать байт из стандартной таблицы знакогенератора, выделять старшую часть байта и сдвигать ее вправо, затем выделять младшую часть байта и сдвигать ее влево. Старшую и младшую часть снова будем собирать в байт, который запишем в 1-ую таблицу знакогенератора. Эти действия будем повторять 16 раз т.к. стандартная высота символа равна 16-ти. Далее пропустим 16 байтов. Организуем цикл по вышеописанному заполнению 32 байтов для всех 256 символов в таблице с номером 5.
Для отслеживания 2 секунд пишем свой обработчик int 1Ch.

```
l segment para public 'code'
assume         cs:l,ds:l,ss:l
org  100h
Begin:    jmp Init          ;переход к главной программе
pres dw   1234h             ;сигнатура
Int_1c    proc far          ;обработчик прерывания 1C
     pusha                  ;сохраняем регистры в стеке
     inc  cs:Tik     ;увеличиваем значение счетчика "тиков"
     mov  al,cs:Tik         ;занесем полученное значение в al
     cmp  cs:Time,al        ;сравним с нашим промежутком времени
     jne  Vce          ; если значения равны, переключаем таблицы
     mov  al,03             ;в al номер регистра синхронизатора
     mov  dx,3c4h    ;в dx индексный порт граф. контроллера
```

```
     out  dx,ax          ;выводим в порт
     inc  dx             ;получим адрес порта данных
     cmp  cs:Yes,0       ;если активна стандартная таблица
     je   Zer            ;то переключаемся на 5-тую таблицу
     mov  al,0110000B    ;в al содержимое битов 5,3,2 и 4,1,0
                         ;для 5-ой таблицы
     mov  cs:Yes,0
     jmp  out_port       ;переход на запись в порт
Zer: mov  al,0           ;в al номер стандартной таблицы
     mov  cs:Yes,1
out_port:
     out  dx,al          ;выводим значение в порт
     mov  cs:Tik,0       ;обнулим значение счетчика "тиков"
Vce: popa                ;восстановим регистры из стека
     iret                ;выходим из прерывания
Int_1c    endp
int1c     dd   ?         ;адрес оригинального вектора
Yes  db   0              ;флажок вывода
Tik  db   0              ;счетчик "тиков"
Time db   18*2           ;заданный промежуток времени
@DS  dw   ?              ;переменная для хранения DS
Init:                    ;главная программа
     cli                 ; запретить прерывания
     mov  dx,3c4h        ;в dx адрес порта синхронизатора
     mov  si,offset SPar ;в si смещение параметров
     mov cx,2            ;выводит 4 параметра
l1:  lodsw               ;в ah значение параметра,
                         ;в al номер регистра синхронизатора
     out  dx,ax          ;вывести параметр в регистр
     loop l1             ;повторять 4 раза
     sti                 ;разрешить прерывания
     mov  dx,3ceh        ;в dx адрес порта граф. контроллера
     mov  si,offset GPar ;в si смещение параметров
     mov  cx,3           ;выводить 3 параметра
l2:  lodsw               ;в ah значение параметра,
                         ;в al номер регистра
     out  dx,ax          ;вывести параметр в регистр
     loop l2             ;повторять 3 раза
     mov  ax,0a000h      ;загрузить в es адрес
     mov  es,ax          ;видеобуфера
     mov  di,0           ;обнулить смещение
```

```
     mov  cx,256          ;будем просматривать все 256 символов
ll2: push cx              ;сохраним cx в стеке
     mov  cx,16           ;для высоты символа16
ll1: mov  al,es:[di]      ;в al байт определения символа
     mov  bl,al           ;сохранить значение в bl
     and  al,0f0h         ;взять младшую часть байта
     shr  al,1            ; сдвинуть вправо
     mov  ah,bl
     and  ah,0fh          ;взять старшую часть байта
     shl  ah,1            ;сдвинуть влево
     or   ah,al           ;получим результат в ah
     mov  es:[2000h+di],ah ;занесем результат в 5-ую таблицу
     inc  di              ;увеличим смещение
     loop ll1             ;повторять 16 раз
     pop  cx              ;восстановить cx
     add  di,16           ;пропустить 16 байтов
     loop ll2             ;повторить 256 раз
; Восстанавливаем значения регистров синхронизатора
; и графического контроллера
     mov  dx,3c4h         ;в dx адрес инд. порта синхронизатора
     mov si,offset SSP    ;в si смещение параметров
     mov cx,2             ;выводить 2 значения
E1:  lodsw                ;в ax полученный результат
     out  dx,ax           ;вывести в порт
     loop E1              ;повторить 2 раза
     mov  dx,3ceh         ;в dx адрес инд. порта граф.контроллера
     mov  si,offset GGP   ;в si смещение параметров
     mov cx,3             ;выводить 3 параметра
E2:  lodsw                ;в ax результат
     out  dx,ax           ;вывести в порт
     loop E2              ;повторить 3 раза
     mov  cs:@DS,es       ;запомнить значение es в переменной
     mov  ax,351ch        ;получить адрес вектора 1C
     int  21h
     mov  ax,es:[bx-2]    ;проверяем наличие программы
     cmp  ax,cs:pres      ;в памяти
     je   Exit            ;если есть, то на выход
     mov  word ptr cs:int1c+2,es   ;запомнить оригинальный
     mov  word ptr cs:int1c,bx     ;адрес
     mov  ax,251ch        ;установим вектор 1C
     lea  dx,int_1c       ;на свой обработчик
```

```
     int  21h
     lea  dx,Init          ;Выходим
     int 27h               ;и оставляем резидентом
Exit:
     <восстановим старый обработчик int 1Ch, выгрузим окружение
программы и саму программу>
     mov  dx,3c4h          ;установим стандартную таблицу
     mov  ax,03
     out  dx,ax            ;шрифтов
     mov  ax,4c00h         ;завершаем
     int  21h              ;программу
SPar dw   0402h            ;запись только во 2-ой видеослой
     dw   0704h            ;последовательная адресация
GPar dw   0204h            ;выбор видеослоя для считывания
     dw   0005h            ;запрещение нечетной адресации
     dw   0006h            ;видеопамять начинается с A000:0000h
SSP  dw   0302h
     dw   0304h
GGP  dw   0004h            ;доступны 0 и 1 слои для считывания
     dw   1005h            ; Разрешение нечетной адресации
     dw   0e06h            ; Видеопамять начинается с B800:0000h
ends l
end  begin
```

Варианты:

1. Написать программу, которая переворачивает символы "вверх ногами".
2. Предоставить пользователю возможность набора математических знаков.
3. Реализовать возможность набора в текстовом режиме римских цифр.
4. Написать программу, позволяющую каждые 5 секунд переворачивать цифры.
5. Уменьшить высоту заглавных букв латинского алфавита.
6. Заменить в таблице знакогенератора русские буквы на греческие.
7. Реализовать возможность набора иероглифов.
8. Написать программу, заменяющую английские буквы их русскими "эквивалентами", т.е "i" - "и", "n" - "н", "u" - "ю", "s" - "с" и т.д.
9. Заменить цифры на пробел и инвертировать буквы.
10. Реализовать подчеркивание строчных букв латинского алфавита.
11. Сжать символы в таблице знакогенератора в 2 раза.
12. Обвести латинские буквы в рамочку.
13. Создать ТЗГ с высотой символов в 6 битов и заменить стандартную таблицу на новую.

14. Написать программу, обеспечивающую стирание левой половины шрифта при нажатии клавиши F1.

5.8 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Порт AGP необходим для:

   а) увеличения количества битов для хранения цвета пиксела;

   б) расширения разрядности шины графического контроллера;

   в) ускорения видеопостроений путем использования оперативной памяти графическим процессором.

   Ответы : 1 - б, 2 - а, 3 - д, 4 – а, 5 - с.

2. Какой из фрагментов программ выполняет вывод буквы А в правый верхний угол экрана?

   а)
   ```
   mov  ax,0B800h
   mov  es,ax
   mov  bx,80*2
   mov  es:[bx],'A'
   ```

   б)
   ```
   mov  ax,0B800h
   mov  es,ax
   mov  bx,79*2
   mov  es:[bx],'A'
   ```

   в)
   ```
   mov  ax,0B800h
   mov  es,ax
   mov  bx,80*2
   mov  es:[bx+1],'A'
   ```

   г)
   ```
   mov  ax,0B800h
   mov  es,ax
   mov  bx,79*2
   mov  es:[bx+1],'A'
   ```

   д)
   ```
   mov  ax,0A000h
   mov  es,ax
   mov  bx,80*2
   mov  es:[bx],'A'
   ```

3. Имеется несколько правильных и неправильных утверждений об использовании слоев видеопамяти в текстовом режиме:

   1) Стандартно доступен только нулевой слой видеопамяти, в котором хранится ASCII-код символа.
   2) Нулевой и первый слои видеопамяти отображаются на общее адресное пространство, причем по четным адресам хранятся байты из 0 слоя, по нечетным - байты из 1 слоя.

3) Стандартно таблица знакогенератора является недоступной, для получения к ней доступа необходимо изменить содержимое регистров синхронизатора и графического контроллера.
4) Во втором слое видеопамяти хранится таблица знакогенератора, содержащая изображения символов.
5) Байты из 0 и 1 слоев видеопамяти отображаются на общее адресное пространство по принципу: сначала все байты из 0 слоя, потом все байты из 1 слоя.

Из этих утверждений верно: а) 1,4; б) все, кроме 5; в) 1,2,3; г) 2,3,4; д) 3,4.

4. Какой из нижеприведенных фрагментов программы выполняет создание новой таблицы знакогенератора из стандартной путем инверсии битового изображения символа, если в es загружен сегментный адрес 0A000h:

а)
```
   mov  si,0
   mov  cx,2000h
c: mov  al,es:[si]
   not  al
   mov  es:[si+4000h],al
   inc  si
   loop c
```

б)
```
   mov  si,0
   mov  cx,2000
c: mov  al,es:[si]
   xor  al,al
   mov  es:[si+4000h],al
   inc  si
   loop c
```

в)
```
   mov  si,0
   mov  cx,256
c: mov  al,es:[si]
   not  al
   mov  es:[si+4000h],al
   inc  si
   loop c
```

г)
```
   mov  si,0
   mov  cx,2000
c: mov  al,es:[si]
   or   al,0Fh
   mov  es:[si+4000h],al
   inc  si
   loop c
```

д)
```
   mov  si,0
   mov  cx,2000h
c: mov  al,es:[si]
   and  al,0FFh
   mov  es:[si+4000h],al
   inc  si
```

```
loop c
```

5. Высота символов в текстовом режиме может принимать значения:

   а) не более 32
   б) 8, 14, 16
   в) от 1 до 32
   г) не более 16
   д) от 8 до 16

Ответы: 1 - б; 2 - г; 3 - а; 4 - в.

6 УСТРОЙСТВА ХРАНЕНИЯ ДАННЫХ

6.1 УСТРОЙСТВО ЖЕСТКИХ ДИСКОВ НА ФИЗИЧЕСКОМ УРОВНЕ

В настоящее время несмотря на все возрастающее значение flash-накопителей лидером при хранении данных остаются магнитные носители. Оптические же устройства часто используются в качестве вспомогательных при хранении данных.

В основе функционирования дисковых накопителей лежит следующее их устройство. Слой носителя информации нанесен на рабочие поверхности дисков. Диски вращаются с помощью двигателя шпинделя, обеспечивающего требуемую частоту вращения в рабочем режиме, измеряемой в оборотах в минуту. На диске имеется индексный маркер, который, проходя мимо специального датчика, отмечает начало каждого оборота диска. Информация на диске располагается на концентрических дорожках (*треках, tracks*), нумерация которых начинается с внешней дорожки, имеющей номер 0. Каждая дорожка разбита на секторы фиксированного размера. Сектор является минимальным блоком информации, который может быть считан или записан на дорожку. Нумерация секторов на физическом уровне начинается с 1 и привязывается к индексному маркеру. Если накопитель имеет несколько рабочих поверхностей, т.е. на шпинделе размещается пакет дисков, то совокупность всех дорожек с одинаковыми номерами называется цилиндром (*cylinder*). Для каждой рабочей поверхности в накопителе имеется своя головка (*head*), обеспечивающая запись (если это позволяет накопитель) и считывание информации. Для записи или чтения сектора необходимо выполнение следующих условий: шпиндель должен вращаться с заданной скоростью и блок головок должен быть подведен с помощью специального двигателя к нужному цилиндру. Когда требуемый сектор подойдет при вращении диска к выбранной головке, начнется физическая операция обмена данными.

Электромеханическая часть накопителя - пакет дисков со шпиндельным двигателем и блок головок с приводом - называется блоком HDA (Head Disk Assembly) и заключается в защитный кожух. Дисковый накопитель имеет также блок электроники, управляющий приводами двигателей, а также обслуживающий сигналы рабочих головок записи-считывания. Контроллером накопителя называют электронное устройство, которое является посредником между блоком HDA и программами, в т.ч. ОС, и имеет интерфейс, позволяющий вести обмен байтами команд, состояния и данных, читаемых с диска или записываемыми на него.

Жесткие диски отличаются емкостью, быстродействием и интерфейсом. Быстродействие зависит от скорости доступа к данным и скорости чтения-записи данных. Под интерфейсом понимается тип подключения дискового контроллера. В современных накопителях на жестких дисках контроллер расположен на плате электроники, смонтированной вместе с блоком HDA. Контроллер гибких дисков вынесен на специальную плату адаптера или размещается на системной плате. Характерной особенностью современных дисков является поддержка протокола прямого обмена между диском и памятью (DMA, Direct Memory Access).

В накопителях на гибких дисках в нерабочем состоянии головка поднята над поверхностью диска на несколько миллиметров, а в рабочем прижимается к поверхности диска. Такой непосредственный контакт головки с поверхностью допустим только при малых скоростях движения носителя. В накопителях на жестких дисках головки поддерживаются на микроскопическом расстоянии от рабочей поверхности аэродинамической подъемной силой. Головки как бы плавают над поверхностью жесткого диска на специальной воздушной подушке и могут перемещаться вдоль радиуса диска с помощью двигателя.

В качестве привода для позиционирования головок на нужный цилиндр для накопителей на гибких магнитных дисках и жестких дисков устаревшей конструкции применяют шаговые двигатели, которые под действием серии импульсов способны поворачивать свой вал на определенный угол, который кратен минимальному шагу, определяемому конструкцией двигателя. Поворот вала на 1 шаг приводит к перемещению блока головок на 1 цилиндр. При этом ошибки позиционирования откорректировать нельзя, можно только вернуться на нулевой цилиндр и снова попробовать переместиться к нужному цилиндру. Операция возврата на нулевой цилиндр называется рекалибровкой диска. В современных накопителях используется привод головок с подвижной катушкой, где блок головок в зависимости от силы тока и напряжения можно переместить в произвольное положение. При этой системе позиционирования контроллеру постоянно необходима информация о текущем положении головок, чтобы обеспечить точное позиционирование головок по сигналу обратной связи. Для этой цели используется размещение на диске сервометок, которые записываются на диск при сборке. По месту размещения сервометок различают накопители с выделенной сервоповерхностью и со встроенными сервометками. В первом случае в пакете дисков выделяется одна поверхность, используемая только для хранения сервометок, и соответствующая ей головка является сервоголовкой. Ошибка позиционирования в такой системе может возникать вследствие перекоса головок в блоке. В

накопителях со встроенными сервометками информация записывается на рабочих поверхностях между секторами с данными.

Для лучшего понимания работы дискового контроллера рассмотрим структуру дорожки. Для дискеты признаком начала дорожки служит физический индекс, который генерируется при каждом обороте диска. При операциях записи происходит перезапись отдельных секторов, что приводит к сбою намагниченности в месте обрыва записи, причем физическая длина сектора не всегда точно соответствует длине сектора, ранее записанного на этом месте. Поэтому на дорожке информация о содержимом секторов чередуется с межсекторными промежутками, служащими для синхронизации. Начало дорожки содержит индексную адресную метку (IAM). Формат дорожки в общем виде приведен на рисунке 6.1.

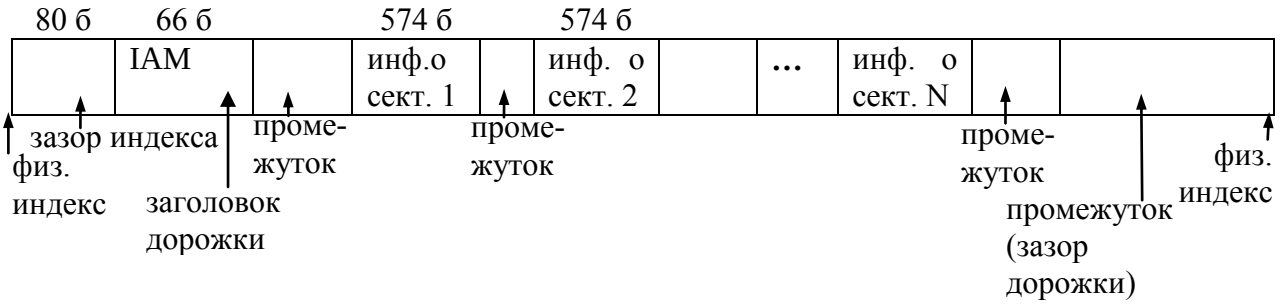


Рисунок 6.1

Для жесткого диска начало дорожки определяется сигналом наличия индекса. Генерация сигнала осуществляется при каждом обороте диска.

Информация о содержимом сектора включает в себя не только данные, хранимые в секторе, но и служебную информацию, позволяющую контроллеру идентифицировать сектор и проверить правильность считывания-записи. Пример структуры данных об одном секторе приведен на рисунке 6.2. Отметим, что для различных дисков эта структура может незначительно отличаться.

маркер идентификатора	№ цилиндра C	№ головки H	№ сектора S	код длины сектора	контр. сумма идентификатора	промежуток	маркер данных	данные сектора	контр. сумма данных
14 б	1 б	1 б	1 б	1 б	2 б	22 б	16 б	512 б	2 б

Рисунок 6.2.

Номер цилиндра, головки, сектора и код длины сектора называются идентификатором адреса сектора. В маркере идентификатора делается отметка о дефектности сектора. При чтении и записи информации вычисляется контрольная сумма всех байтов как заголовочной части, так и части данных. Если при чтении контрольная сумма не совпадает с ее значением, хранящимся на диске, то контроллер считает это ошибкой.

Для лучшего понимания логики обращения к диску совершим краткий исторический экскурс. Диски современных персональных компьютеров развились из диска машины IBM PC XT. Это был диск Seagate на 10 Мбайт, у него было 4 головки, 306 цилиндров и по 17 секторов на дорожке. Контроллер мог управлять двумя дисками. Операционная система считывала с диска и записывала на диск информацию. Для этого она передавала параметры в регистры процессора и вызывала систему BIOS для собственно обмена с диском. Для работы с диском на физическом уровне использовалось прерывание BIOS int13h, для чтения сектора - функция 02h, для записи - 03h, причем шла адресация непосредственно к цилиндрам, головкам и секторам устройства в формате **Cylinder-Head-Sector** (C-H-S).

Сначала контроллер помещался на отдельной плате, а с выходом в середине 80-х годов устройств IDE (Integrated Drive Electronics — устройство со встроенным контроллером) стал встраиваться в печатную плату, расположенную в корпусе винчестера. Обращение к секторам производилось по номерам головки, цилиндра и сектора, причем головки и цилиндры нумеровались с 0, а секторы — с 1. Есть предположение, что такая ситуация сложилась из-за ошибки одного из программистов BIOS. Имея 4 бит для номера головки, 6 бит для сектора и 10 бит для цилиндра, диск мог содержать максимум 16 головок, 63 сектора и 1024 цилиндра, то есть всего 1032192 сектора. Емкость такого диска составляла 504 Мбайт, и в те времена эта цифра считалась огромной (а вы бы стали сегодня ругать новую машину, неспособную манипулировать дисками объемом более 1 Пбайт?).

Вскоре появились диски объемом более 504 Мбайт, но у них была другая геометрия (4 головки, 32 сектора, 2000 цилиндров). Операционная система не могла обращаться к ним из-за того, что соглашения о вызовах системы BIOS не менялись (требование совместимости). Хотя электрический интерфейс и регистры контроллера накопителя даже при адресации C-H-S позволяли хранить на диске до 31,5 Гбайт информации! Этот пример иллюстрирует расхождение между потенциальными возможностями интерфейса и его программной реализацией. Подобных ограничений при программной реализации существует не одно, и они имеют программный характер. Для преодоления программных барьеров в расширенных версиях BIOS используется трансляция логических параметров вызова int 13h в физические параметры, передаваемые контроллеру жесткого диска. Логический трехмерный адрес (цилиндр-головка-сектор) (C, H, S) переводится в физический, причем способы такой трансляции могут быть различны.

На смену IDE-дискам пришли устройства **EIDE** (Extended IDE — **усовершенствованные устройства со встроенным контроллером),** поддерживающие дополнительную схему адресации **LBA** (Logical Block Addressing —**линейная адресация блоков).** При линейной адресации секторы просто нумеруются от 0 до 2^{28} - 1. Хотя

контроллеру приходится преобразовывать LBA-адресав адреса головки, сектора и цилиндра, зато объем диска может превышать 504 Мбайт. Однако, к сожалению, в результате родилось новое ограничение на уровне 2^{28} x 2^{9} байт (128 Гбайт). В 1994 году, когда принимался стандарт EIDE, никому и в голову не приходило, что через некоторое время появятся диски такой емкости.

Стандарт EIDE совершенствовался, и его преемника назвали **ATA-3** (AT Attachment), (сокращение AT, образованное от словосочетания Advanced Technology). Следующая версия стандарта, названная **ATAPI**-4 (ATA Packet Interface — **пакетный интерфейс ATA),** отличалась скоростью 33 Мбит/с. В версии ATAPI-5 она достигла 66 Мбит/с. Поскольку ограничение в 128 Гбайт, установленное 28-разрядными линейными адресами, становилось все более болезненным, в стандарте ATAPI-6 размер LBA-адреса был увеличен до 48 бит. Лимит этого стандарта — 2^{48}x2^{9} (128 Пбайт).Если емкость дисков будет ежегодно возрастать на 50%, 48-разрядные LBA-адреса останутся актуальными приблизительно до 2035 года. Очевидно, следующим шагом будет увеличение размера LBA-адреса до 64 бит. Настоящий прорыв был совершен в стандарте ATAPI-7. Появилась спецификация последовательного интерфейса ATA (Serial ATA, **SATA),** повысить скорость передачи до 1,5 Гбит/с и уменьшить общий уровень энергопотребления.

Наряду с IDE-дисками существуют SCSI-диски(Small Computer System Interface — **интерфейс малых вычислительных систем)**, которые имеют другой интерфейс и более высокую скорость передачи данных. Аббревиатура SCSI произносится как «скази». SCSI — это не просто интерфейс жесткого диска. Это шина, к которой могут подсоединяться SCSI-контроллер и до семи дополнительных устройств. Ими могут быть один или несколько жестких SCSI-дисков, устройства для *чтения* и записи компакт-дисков, сканеры, накопители на магнитной ленте и другие периферийные устройства. Каждое устройство имеет свой идентификационный код. SCSI-диски используют LBA как естественную систему адресации, а для совместимости с традиционным дисковым сервисом используется фиктивная трехмерная геометрия.

Простейшим способом установления соответствия между C-H-S и LBA является соотношение:

LBA = (C*<количество головок>+H)*<количество секторов на дорожке>+S-1,

где C, H, S - это номера цилиндра, головки и сектора в пространстве координат C-H-S. Однако следует помнить, что приведенная формула является весьма условной, и реальную физику диска знает только его контроллер.

Начальная загрузка компьютера всегда выполняется через вызовы в формате C-H-S. При перенесении физического диска на другой компьютер возможно возникновение

проблем, связанных с версией BIOS. Если, например, отформатировать винчестер на компьютере с расширенной BIOS, а эксплуатировать на машине со стандартной BIOS, то после выхода за программный барьер снова пойдет запись по нулевому и последующим цилиндрам.

Как было отмечено выше, непосредственно передачей данных и позиционированием головок управляет контроллер диска. Контроллеры современных дисковых накопителей представляют собой устройства, базирующиеся на микропроцессорах. Рассмотрим некоторые интересные возможности современных контроллеров.

Так как хранение данных на магнитном носителе всегда сопровождается появлением ошибок (причина - дефект поверхности носителя, попадание посторонней частицы под головку, неточность позиционирования головки над треком), то контроллер должен следить за ошибками чтения. Если считывание произошло с ошибкой, то возможно повторение позиционирования головки над дорожкой в случае шагового привода головки или поиск оптимального положения головки для привода с подвижной катушкой. Если сектор все равно не читается, то он исключается из дальнейшего использования. В заголовке сектора делается отметка о дефектности и вместо плохого сектора используется сектор из резервной области, которая имеется в конце каждой дорожки, поэтому дефектные секторы у дисков не видны.

Надежность считывания в значительной степени зависит от точности позиционирования головки относительно продольной оси трека. Позиционирование для каждой головки может требовать коррекции, которая необходима из-за изменения температуры. Контроллер диска хранит карту отклонений для цилиндров и головок, которую он автоматически изменяет в процессе работы. Процесс термокалибровки накопителя запускается контроллером в случайно выбранные моменты времени. Процесс термокалибровки заметен для пользователя - винчестер, к которому нет обращений, начинает «жить собственной жизнью», выполняя серию позиционирований. Доступ к данным при этом приостанавливается, поэтому контроллеры накопителей, используемых для обеспечения длительной непрерывной передачи данных, не должны запускать термокалибровку случайным образом.

Кроме термокалибровки, контроллер может асинхронно запускать свипинг - если к диску долгое время нет обращений, то головки перемещаются в новое случайное положение. Свипинг служит для выравнивания степени износа поверхности диска.

Современные контроллеры имеют встроенную кэш-память, структура которой и алгоритм использования в значительной мере определяют производительность обмена данными. Общепринятой технологией кэширования диска является упреждающее

считывание (Read Ahead). Если контроллер получает запрос на чтение сектора, то он считает и последующие секторы на той же дорожке, как, например, SCSI-контроллер. Если у накопителя контроллер с адаптивным кэшированием, то размер областей, выделяемых под упреждающее чтение, зависит от текущей статистики обращений к диску. Если обращения одиночные, то большие области под упреждающее чтение не выделяются.

Отметим, что BIOS в процессе инициализации создает таблицы параметров жесткого диска, которые располагаются в области данных BIOS. В ней хранится код размера сектора в байтах (0-128 байтов; 1-256; 2-512; 3-1024), длина межсекторного промежутка, байт-заполнитель для форматирования (обычно это F6h). Размер таблицы параметров жесткого диска - 16 байтов.

Таким образом, за функционирование дискового устройства отвечает дисковый контроллер, который может использоваться для управления более чем одним диском. В 1988 году специалистами из Калифорнийского университета Беркли была предожена система хранения данных на основе нескольких дисков. Ее назвали RAID (Redundant Array of Inexpensive Disks – избыточный массив недорогих дисков. Для RAID-массивов было разработано шесть базовых конфигураций, которые названы уровнями RAID. RAID0 – это базовая конфигурация дискового массива, предназначенная для повышения производительности системы, которая предусматривает объединение частей различных физических дисков в один том, называемый перекрытым. Архитектура RAID1 позволяет повысить надежность хранения данных путем записи их идентичных копий на двух дисках, называемых зеркальными. Уровни RAID 2,3 и 4 предназначены для повышения надежности системы с помощью контроля правильности записи данных на дисках путем хранения их контрольной суммы по модулю 2 на дополнительном диске. Наличие контрольной суммы по модулю 2 позволяет восстанавливать сбойные данные по остальным данным и контрольной сумме. В RAID5 также используется схема восстановления данных, основанная на контрольной сумме по модулю 2, однако информация, предназначенная для контроля данных, хранится не на отдельном диске, а распределяется между всеми дисками.

6.2 ЛОГИЧЕСКАЯ СТРУКТУРА ФИЗИЧЕСКОГО ДИСКА

С аппаратной точки зрения любой диск можно представить как совокупность секторов, адресуемых тем или иным способом (CHS или LBA). Однако для большинства прикладных программ интерес представляет не обращение к отдельным секторам, а возможность обращения к файлам, которые могут занимать произвольное количество

секторов. Для облегчения обращения к файлам и упорядочения использования пространства секторов диска в состав любой операционной системы входит файловая система.

Физический диск, как правило, разбивается на логические диски. . Дискеты в такой разбивке не нуждаются из-за небольшого объема. Исторически разбиение выполнялось, во-первых, для удобства работы с информацией (этот фактор важен и сейчас) и, во-вторых, из-за невозможности младших версий DOS работать с дисками более 32 Мб из-за 16-разрядной адресации секторов. В настоящее время разбиение физического диска на логические выполняется не только для удобства, но и для реализации возможности загрузки различных операционных систем. При разбиении диска на области, называемые разделами, в каждой из них может быть создана своя файловая система, соответствующая определенной ОС (например, FAT32, NTFS и т.д.).

Рассмотрим логическую структуру физического диска. Первый сектор жесткого диска (дорожка 0, головка 0, сектор 1) содержит главную загрузочную запись MasterBoot Record (MBR). В ней находится главная программа начальной загрузки, помещаемая по адресу 0h:7C00h, и таблица разделов диска Partition Table, начинающаяся со смещения 1BEh относительно начала сектора MBR. Программа начальной загрузки определяет активный раздел и загружает самый первый сектор этого раздела, где находится загрузчик операционной системы. Partition Table содержит описания четырех разделов диска, каждое из которых занимает 16 байтов.

Формат одного описателя в таблице разделов приведен в таблице 6.1.

Таблица 6.1 - Структура описателя раздела

Смещение	Длина,байт	Назначение
00h	1	флаг активности раздела (80h- активный загружаемый, 0h -нет)
01h	1	номер начальной головки
02h	2	номер начального сектора и цилиндра в формате загрузки регистра cx в BIOS int 13h
04h	1	системный код
05h	1	номер конечной головки
06h	2	номер конечного сектора и цилиндра
08h	4	относительный номер начального сектора раздела
0Ch	4	количество секторов в разделе

Системный код служит для характеристики типа и размера раздела. Элементы логической структуры диска рассматривались в курсе "Операционные системы", однако для

связности изложения материала некоторые ключевые моменты будут повторены в данном учебном пособии.

Напомним, что заполнение элементов таблицы разделов осуществляется не обязательно с первого. Расширенный раздел позволяет создать не 4 логических диска соответственно таблице разделов, а произвольное их количество (до . . . z) (рисунок 6.3).

Если код системы равен 5, то элемент таблицы разделов указывает на так называемую таблицу логических дисков. Таблица логических дисков содержит два элемента, первый из которых указывает на первый сектор логического диска, а второй элемент содержит код системы, равный 0 или 5. Если код системы равен 0, то это означает что логические диски закончились, а если код системы равен 5, то соответствующий элемент текущей таблицы логических дисков указывает на следующую таблицу логических дисков. Структура элемента таблицы логических дисков аналогична структуре описателя раздела в MBR.

Отметим, что секторы, содержащие MBR и таблицы логических дисков, не принадлежат логическим дискам, а значит, их можно прочитать только прерыванием ***int 13h.***

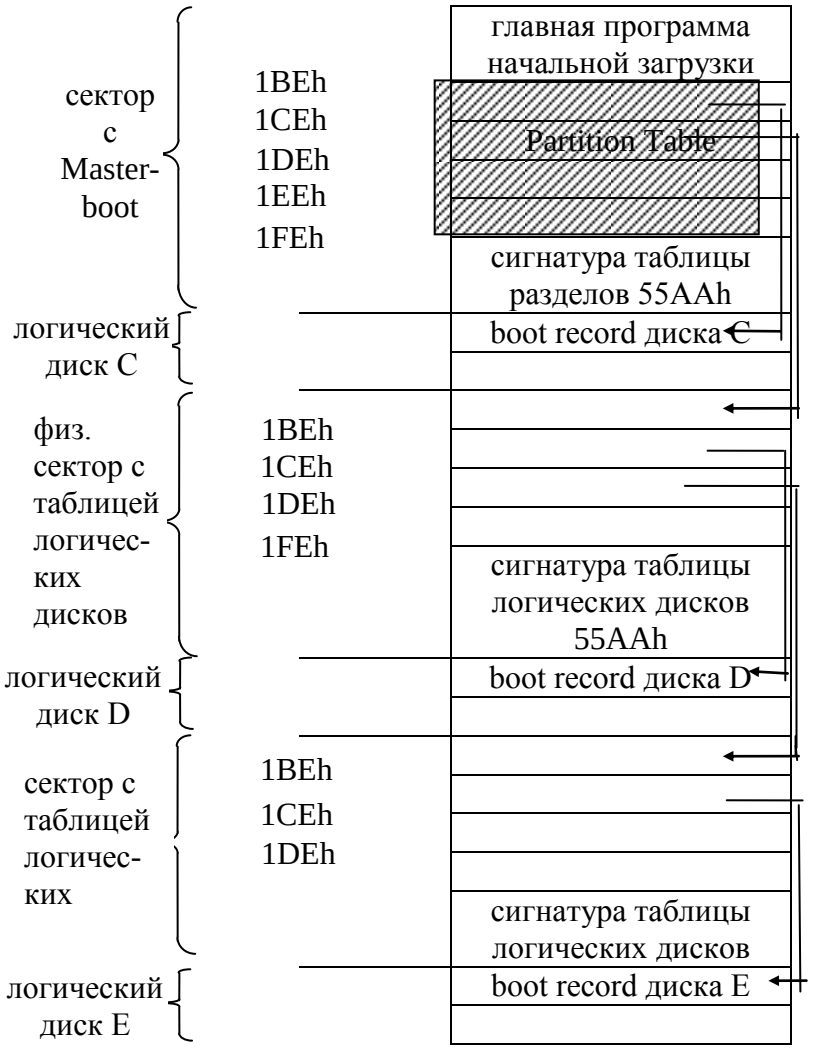


Рисунок 6.3 - Структура физического диска

Основополагающей операцией при обработке информации на логическом диске является чтение и запись секторов. Для того чтобы обрабатывать информацию о файлах и каталогах, хранящуюся в секторах диска, необходимо рассмотреть саму структуру логического диска. Логический диск начинается с сектора загрузчика (***boot record***), после которого располагаются служебные структуры для файловых систем (например, одна или несколько копий таблицы размещения файлов ***FAT*** (File Allocation Table)) и собственно область данных (рисунок 6.4).

Рисунок 6.4

Boot record хранится в секторе логического диска с номером 0 и помимо программы, загружающей операционную систему (если раздел активен), содержит следующую информацию (приведем отдельные поля):

1) ***смещение 0***: код команды jmp с адресом программы начальной загрузки, занимает 3 байта, т.к. происходит ближний переход на программу, находящуюся здесь же, в Boot. Если раздел не является активным, то здесь записаны нули.

2) ***смещение 3:*** фирма - автор ОС и версия ОС - размер 8 байтов.

3) ***смещение 11***: расширенный блок параметров BIOS (EBPB) размером 25 байтов.

4) ***смещение 36***: физический номер устройства (0 - дискета, 80h - 1MD, 81h - 2MD) - размер 1 байт.

5) ***смещение 39***: серийный номер диска, формирующийся на основе даты и времени форматирования. Размер поля - 4 байта.

EBPB содержит информацию, необходимую для обработки файлов и каталогов. Часть полей EBPB приведены в таблице 6.2.

Таблица 6.2 - Поля расширенного блока параметров BIOS

Смещение	Размер	Содержание	Название
0	2	количество байтов в одном секторе	ByteInSect
2	1	количество секторов в одном кластере	SectInClas
3	2	количество зарезервированных секторов, фактически это номер начального сектора первой копии FAT	ResSectors
5	1	количество FAT (обычно 2)	KolFat
6	2	максимальное количество записей (дескрипторов файлов) в Root	KolRecRoot

8	2	общее количество секторов на носителе	AllSect
10	1	описатель носителя данных по количеству сторон диска и количеству секторов диска	Media
11	2	количество секторов в первой копии FAT	SectInFat
13	2	количество секторов на дорожке	SectInTrk
15	2	количество головок	NumHead
17	2	количество скрытых секторов для раздела меньше 32 Мб. Скрытые сектора не принадлежат логическим дискам и могут содержать таблицы логических дисков	Hidden1
19	2	количество скрытых секторов для раздела более 32 Мб.	Hidden2
21	4	общее количество секторов на логическом диске более 32 Мб	TotSect

Информация из boot-record используется для организации обхода древовидной структуры каталогов.

Для различных ОС хранение файлов и информация о файловой системе организованы по разному. Для файловой системы FAT область данных логического диска разбивается ОС на участки одинакового размера, которые называются кластеры. Кластер содержит группу секторов. Нумерация кластеров начинается с 2. Количество секторов в кластере хранится в EBPB Boot Record - поле SectInClas (см. таблицу 6.2).

Информация в файле записывается в свободные кластеры на диске. Так как свободные кластеры не обязательно располагаются подряд, то для отслеживания, в каких кластерах расположена информация файла, используется FAT. FAT представляет собой массив из 12-ти, 16-ти или 32-битовых элементов, соответствующих кластерам. 12-битовый формат FAT использовался для дискет, где количество секторов является небольшим.

Первые два элемента FAT содержат байт - описатель среды (поле Media в boot-record), после которого остальные байты (2 или 3) содержат 0FFh. Первые два элемента FAT для хранения информации о файле не используются. Если элемент FAT хранит 0, то кластер свободен; если FFF0 . . . FFF6 - кластер зарезервирован; дефектный кластер отмечен в FAT значением FFF7. Во всех остальных случаях элемент FAT хранит либо номер следующего кластера, распределенного файлу, либо, если это последний кластер файла, значение FFF8 - FFFFh. Таким образом, кластера одного файла связаны в цепочку или список.

Номер кластера можно пересчитать в начальный номер сектора, соответствующего кластеру, используя формулу:

NSect = (NClast - 2)·SectInClas + ResSectors + KolFat·SectInFAT + SectInRoot.

Все значения, кроме SectInRoot, берутся из EBPB в boot-record (см. таблицу 7.2). Вычисление количества секторов в Root рассмотрим ниже.

Рассмотрим выдачу содержимого элемента FAT с номером, хранящимся в регистре ***ax*** (формат FAT - 16-битовый). Для чтения сектора, содержащего ***ax***-тый элемент FAT, нужно вычислить его номер, а для извлечения из этого сектора нужного элемента FAT нужно вычислить смещение элемента FAT от начала сектора. Сначала номер элемента FAT умножается на два, что дает смещение в байтах нашего элемента от начала FAT, а потом делением на количество байтов в секторе определяется, в каком секторе от начала FAT находится нужный элемент:

```
mov   cx,2
mul   cx              ; умножим ax и cx. Результат - в ax
mov   cx,ByteInSect
div   cx
```

После выполнения деления в ***ax*** находится номер сектора от начала FAT, в ***dx*** - остаток от деления, т.е. смещение от начала сектора.

Напомним, что каталог логического диска состоит из 32-байтовых элементов, называемых дескрипторами файлов и каталогов.

Формат дескриптора приведен в таблице 6.3. Структура байта атрибутов указывает на назначение файла, в том числе на то, является ли файл каталогом.

Таблица 6.3 - Описание полей дескриптора

Смещение	Размер в байтах	Назначение поля
0h	8	имя файла или каталога
08h	3	расширение
0Bh	1	байт атрибутов
16h	2	время создания или последнего изменения
18h	2	дата создания или последнего изменения
1Ah	2	номер первого кластера файла
1Ch	4	размер файла в байтах

Назначение битов байта атрибутов (при установке их в 1):

бит 0 - файл только для чтения;

бит 1 - скрытый файл;

бит 2 - системный файл;

бит 3 - дескриптор описывает метку тома (11 байтов - имя + расширение);

бит 4 - файл является каталогом;

бит 5 - бит архивации, устанавливается в 1 после обновления содержимого файла.

Время создания или последней модификации файла занимает 2 байта, младшие 5 битов которого служат для запоминания секунд (хранится количество секунд, разделенное на 2 (0-29)), старшие 5 битов служат для запоминания часов (0-23), а оставшиеся 6 битов в середине двухбайтового поля хранят минуты (0-59).

Дата создания файла тоже занимает 2-байтовое поле, младшие 5 битов которого отведены для хранения дня (1-31), старшие 7 битов предназначены для хранения года (0-119), а оставшиеся "средние" 4 бита служат для хранения месяца (1-12). Для получения значения года нужно к содержимому старших 7 битов прибавить число 1980.

Отметим, что в поле “длина файла” у каталогов записано нулевое значение. Через номер первого кластера файла его дескриптор связан с FAT. При удалении файла в первый байт его имени заносится E5h, кластеры этого файла отмечаются в FAT нулями как свободные при сохранении содержимого кластеров, что дает возможность восстановления файлов.

Все каталоги представляют собой файлы специального типа. Пространство для них выделяется из области данных логического диска. Структура каталога физически представляет собой. массив дескрипторов. Любой подкаталог содержит в начале своего массива дескрипторов две специальные записи, имеющие в поле имени “.” и “..”. Первый дескриптор определяет размещение на диске данного каталога, т.е. каталог содержит информацию о самом себе. Поле начального кластера этой записи указывает на кластер, где находится каталог. Второй дескриптор указывает на размещение каталога-родителя, т.е. поле начального кластера указывает на первый кластер родительского каталога. Если родительским является корневой каталог, поле кластера, записываемого с “..”, содержит 0.

Отметим, что рекурсивная структура дерева каталогов предполагает рекурсивную процедуру его обхода. При организации обхода каталогов нужно учесть, что признаком конца каталога служит содержимое первого байта имени файла, равное 0 или 0F6h (пустой каталог).

На основе изложенной выше информации рассмотрим процесс загрузки ОС. При загрузке BIOS считывает MBR в память и передает управление ее загрузочному коду, который сканирует таблицу разделов в поисках активного раздела. Найдя активный раздел, код начального загрузчика считывает в память первый сектор загрузочного раздела и передает ему управление. В первом секторе загрузочного раздела находится программа-загрузчик и таблица, подробно описывающая раздел. Загрузчик является короткой программой, загружающей либо ОС, либо ее ядро, либо представляет собой boot manager, дающий возможность выбора загружаемой ОС. Формирование загрузочных секторов осуществляется при установке ОС. Например, программа установки Windows 2000 создает в первом загрузочном разделе жесткого диска свой загрузочный сектор.

Использование остальной части раздела зависит от файловой системы ОС. Рассмотрим более подробно структуру хранения информации в FAT32. Количество резервных секторов для FAT32 обычно равно 32, а не 1, как для FAT12-16. Резервные

сектора хранят копию загрузочного сектора и специальную структуру FSINFO. Специфичная информация для FAT32 хранится, начиная со смещения 24h загрузочного сектора. Рассмотрим некоторые поля, специфичные для FAT32. Они перечислены в таблице 6.4.

Таблица 6.4.

Смещение	Размер, байт	Назначение
24h	4	Количество секторов, занимаемых одной копией FAT
28h	2	Поле флагов активности FAT
2Ch	4	Номер кластера для первого кластера корневого каталога (обычно 2)
30h	2	Номер сектора структуры FSINFO в резервной области логического диска (обычно 1)
32h	2	Номер сектора в резервной области логического диска, используемого для хранения резервной копии загрузочного сектора (обычно 6)

Если бит 7 слова по смещению 28h равен нулю, то изменения в процессе работы отражаются во всех FAT, если бит равен 1, то активна только копия FAT, номер которой хранится в битах 0-3 (нумерация FAT начинается с нуля).

Так как размер FAT может быть очень велик, то для ускорения операций с FAT ввели специальную структуру FSINFO. Сектор с FSINFO содержит в начале специальную сигнатуру-признак наличия в секторе FSINFO, текущее число свободных кластеров на диске, начальный номер кластера на диске, с которого начинается поиск свободных кластеров. Отметим также, что в FAT32 корневой каталог отсутствует. Это связано с неудобством жесткого ограничения на 512 дескрипторов в корневом каталоге. В FAT32 корневой каталог представляет собой файл произвольного размера.

6.3 ОСНОВНЫЕ ПРИНЦИПЫ РАБОТЫ DMA

Прямой доступ к памяти DMA (Direct Memory Access) обеспечивает высокоскоростной обмен данными между периферийными устройствами и оперативной памятью (обычно это диски, стриммеры) без посредничества ЦП. При обычном программном обмене данными для пересылки блока байтов по команде **rep insb**, например, из порта в память, происходят следующие действия:

а) процессор считывает данные из порта во внутренний шинный буфер;

б) процессор генерирует шинный цикл записи в память по заданному адресу;

в) пункты а) и б) автоматически повторяются с изменением адреса памяти. Количество повторов определяется содержимым регистра ***cx***, направление изменения адреса (инкремент или декремент) - флагом DF. Во время передачи всего блока процессор занят.

В режиме прямого доступа к памяти процессор инициализирует контроллер DMA, а именно в регистрах контроллера указываются начальный адрес памяти для обмена, счетчик пересылаемых единиц информации (например, байтов) и режим обмена.

Контроллер DMA имеет независимые каналы для обслуживания различных устройств. Например, 2 канал обычно служит для обмена с контроллером НГМД, 3 канал - для обмена с жестким диском. Назначение каналов может быть изменено.

Режимы работы каждого канала записываются в специальные регистры контроллера DMA (по 1 на каждый канал). В регистре указывается номер канала, направление передачи (например, запись в память или чтение из памяти), увеличение или уменьшение текущего адреса, вид передачи (по требованию, одиночная, блочная), а также разрешение или запрет автоинициализации.

При одиночной передаче (Single Transfer Mode) после каждого цикла передачи контроллер освобождает шину для процессора. При блочной передаче (Block Transfer Mode) шина не освобождается до завершения передачи всего блока. Передача по требованию (Demand Transfer Mode) идет непрерывно до тех пор, пока активен сигнал запроса, состояние которого проверяется после каждого цикла передачи. Если медленное устройство не может продолжить передачу, сигнал запроса сбрасывается и передача приостанавливается.

Обычно после завершения передачи использованный канал DMA маскируется, и для работы с этим каналом контроллер DMA должен быть повторно перепрограммирован. При включенной автоинициализации канал не маскируется, регистры текущего адреса и счетчика циклов снова загружаются из регистров начальных значений адреса и счетчика циклов. Для повторного обмена в режиме автоинициализации нужно только осуществить запрос на прямой доступ к памяти по данному каналу.

Регистр команд контроллера позволяет разрешить или запретить передачу «память-память», разблокировать или заблокировать контроллер, изменить режим смены приоритетов каналов (установить фиксированные приоритеты каналов или обеспечить циклический сдвиг приоритетов, когда только что обслуживаемому каналу присваивается минимальный приоритет). Обычно этот регистр инициализируется BIOS и ошибки при инициализации приводят к нарушению работы ОС.

Регистр состояния сигнализирует о завершении передачи по каналам, если не задан режим автоинициализации, и о на наличии активных запросов на DMA по каналам. Регистр программного запроса на DMA позволяет установить или сбросить запрос на DMA по каналу с заданным номером. Запрос на DMA сбрасывается при сбросе контроллера и при окончании передачи по каналу. Для обслуживания программного запроса на DMA канал должен быть в

режиме блочной передачи. Для маскирования и демаскирования каналов служит регистр маски DMA.

По окончании передачи контроллер DMA обычно формирует сигнал окончания приема/передачи, который может быть использован периферийным устройством для генерации аппаратного прерывания. которое служит признаком завершения передачи.

6.4 ОПТИЧЕСКИЕ ДИСКИ

Большие запоминающие устройства можно создавать на основе оптической технологии. Первым результатом практического применения этой технологии стали компакт-диски, используемые в аудиосистемах. Используемая в CD-системах оптическая технология основана на применении лазерного луча. Лазерный луч направляется на поверхность вращающегося диска, вдоль дорожек которого располагаются впадины, отражающие сфокусированный луч в направлении фотоприемника, фиксирующего записанные на диске двоичные данные. Лазер излучает когерентный свет, состоящий из синхронизированных волн одинаковой длины. Если объединить два одинаковых луча в одной фазе, получится более яркий луч, а если сдвинуть лучи на полфазы, они погасят друг друга. Если два таких луча будут направлены на фотоприемник, то в первом случае он зафиксирует яркое пятно, а во втором случае – темное.

Рассмотрим сечение CD. Его нижний слой выполняется из поликарбонатного пластика, играющего роль прозрачной основы. Данные наносятся на поверхность диска в виде впадин (*pit*), чередующихся с плоскими участками (*land*). Поверх диска с записанной на него информацией нанесен тонкий слой отражающего алюминия, а на него – защитное акриловое покрытие. Сверху клеится этикетка. Источник лазерного излучения и фотоприемник располагаются под поликарбонатным пластиком. Лазерный луч скользит по пластику, отражается от алюминиевого слоя и попадает на фотодетектор. Со стороны лазера впадины выглядят как выпуклости по отношению к плоской поверхности. При вращении диска возможны три случая взаимного расположения источника луча и детектора относительно впадин и площадок. Когда свет отражается только от впадины или только от площадки, детектор фиксирует яркое пятно. Однако на границе впадины и площадки волна, отраженная от впадины, смещается на 180 градусов по фазе относительно волны, отраженной от площадки, и они гасят друг друга. Таким образом, на границах «впадина-площадка» и «площадка-впадина» детектор не видит отраженного луча и фиксирует темное пятно. Если каждый переход, фиксируемый как темное пятно, обозначается двоичным значением 1, то получается двоичная последовательность. Однако эта двоичная

последовательность не является непосредственным представлением хранящихся на диске данных. Для CD применяется сложная система кодирования информации, в которой каждый байт представлен 14-разрядным кодом, позволяющим выявлять и исправлять ошибки.

Впадины располагаются вдоль дорожки, длина которой составляет около 5 км. Данные на дорожке CD-ROM организованы в блоки, которые называются секторами. Существует несколько различных форматов секторов. В соответствии с одним из них, называемым Mode1, размер сектора должен быть равен 2352 байтам. Каждый сектор снабжен 16-байтовым заголовком, который содержит поле синхронизации, используемое для определения начала сектора, и адресную информацию, предназначенную для идентификации сектора. Далее следуют 2048 байт данных. В конце сектора располагаются еще 288 байт, предназначенных для коррекции ошибок.

Для записываемых дисков CD-R спиральная дорожка наносится на диск в процессе производства, а лазер используется для выжигания отверстий в покрывающем диск слое органического вещества. Когда такое вещество нагревается до критическйо температуры, оно темнеет, и при чтении диска темные пятна отражают меньше света. Произвести запись поверх однажды записанных данных на CD-R невозможно, но неиспользованные части диска могут пригодиться для записи дополнительных данных.

Базовая структура перезаписываемых дисков CD-RW подобна структуре CD-R, однако вместо органического слоя здесь используется сплав серебра, индия, сурьмы и теллура. Если этот сплав нагреть до температуры плавления 500 градусов, а потом охладить, то он перейдет в аморфное состояние и приобретет способность поглощать свет. Но если этот сплав будет нагрет до 200 градусов и некоторое время выдержан в таком состоянии, то сплав перейдет в кристаллическое состояние и будет способен пропускать свет. Сплав в кристаллическом состоянии представляет собой площадки диска, а в аморфном состоянии – впадины. Записанные на диск данные можно стирать, возвращая диск в кристаллическое состояние. В дисководах CD-RW используется лазерный луч, имеющий три уровня мощности. Самая высокая необходима для нанесения впадин, средняя используется для стирания, а луч с наименьшей мощностью нужен для чстения. Диски CD-RW выдерживают до 1000 перезаписей.

Единицей хранения файлов и каталогов на оптическом диске является экстент – непрерывная последовательность секторов. Файловые системы оптических дисков могут быть построены либо на основе использования таблицы содержимого диска (Table Of Contents, TOC), включающей атрибуты файлов и описание расположения их экстентов (файловая система ISO 9660), либо на основе пакетной записи, которая предусматривает

хранение экстентов файла вместе со служебной информацией, необходимой для их обработки (файловая система UDF).

В файловой системе ISO 9660 файлы хранятся в одном или нескольких экстентах, представляющих собой непрерывный массив секторов. Все файлы, присутствующие на диске, описаны в TOC. Для одного файла хранится его имя, дата создания, расположение на диске всех его экстентов. Файлы могут располагаться в каталогах, имеющих древовидную структуру. Каталог содержит список файлов и указатели на его первый экстент. Для ускорения поиска файлов на диске имеется таблица путей, содержащая в символьном формате список путей ко всем подкаталогам диска и адреса их начальных секторов. Для стандарта ISO 9660 имеется три уровня совместимости, различные по ограничениям на имена файлов, вложенность каталогов и количество экстентов для файла. Первый уровень допускает имена формата 8.3, вложенность каталогов не более 8, файл занимает не более одного экстента. Второй уровень разрешает длинные имена файлов, вложенность каталогов до 32. Третий уровень дополнительно к возможностям второго допускает размещение файла во множестве экстентов, которые на диске могут чередоваться с экстентами других файлов. Однако для перезаписываемых дисков файловая система ISO является не лучшим решением.

В файловой системе UDF файлы хранятся рядом со своими описаниями, допустимы имена до 127 символов. Основой этой файловой системы является понятие пакета. Пакет содержит файл или экстент файла, в начале пакета имеется описание файла (имя, дата, атрибуты, длина файла и длина данного экстента). Общих хранимых на диске таблиц размещения файлов и экстентов в UDF нет. Для быстрого поиска в памяти компьютера строится виртуальная таблица размещения файлов. Пакеты имеют переменную длину. Формат с переменной длиной пакетов очень экономно расходует дисковое пространство, и устойчив к авариям при записи. Это объясняется отсутствием общих таблиц для файловой системы, которые записываются на диск в самый последний момент.

Существует также формат, основанный на пакетах фиксированной длины. Его недостаток – около 18% места уходит на организацию формата и необходимо хранить прямо на диске таблицу размещения файлов.

6.5 УСТРОЙСТВА ХРАНЕНИЯ НА ОСНОВЕ ФЛЭШ-ПАМЯТИ

В 1988 году Intel представила флэш-память с архитектурой NOR. Годом позже Toshiba разработала флеш-память архитектуры NAND. Рассмотрим их особенности.

Название (NOR — Not OR в булевой математике обозначает отрицание «ИЛИ». Эта схема используется для преобразования входных напряжений в выходные, соответствующие

«0» и «1». Ячейка памяти представляет собой транзистор с двумя изолированными затворами: управляющим (control) и плавающим (floating), который умеет удерживать электроны, то есть заряд. Также в ячейке имеются так называемые «сток» и «исток». При программировании между стоком и истоком при воздействии положительного поля на управляющем затворе, создается канал — поток электронов. Некоторые из электронов, благодаря наличию большей энергии, преодолевают слой изолятора и попадают на плавающий затвор. На нем они могут храниться в течение нескольких лет. Количество электронов (заряда) на плавающем затворе, не превышающее некоторой величины, соответствует логической единице, а все, что больше этой величины, — нулю. При чтении заряд измеряют через измерение порогового напряжения транзистора. Для стирания информации на управляющий затвор подается высокое отрицательное напряжение, и электроны с плавающего затвора переходят (туннелируют) на исток. В технологиях различных производителей этот принцип работы может отличаться по способу подачи тока и чтению данных из ячейки.

Из недостатков, в частности, у флэш-памяти с архитектурой NOR стоит отметить плохую масштабируемость: нельзя уменьшать площадь чипов путем уменьшения размеров транзисторов. Эта ситуация связана со способом организации матрицы ячеек: в NOR архитектуре к каждому транзистору надо подвести индивидуальный контакт. Гораздо лучше в этом плане обстоят дела у флэш-памяти с архитектурой NAND.

NAND — Not AND — в той же булевой математике обозначает отрицание «И». Для записи и стирания данных в NAND-памяти используется туннелирование электронов методом Фаулера — Нордхейма через диэлектрик (FN-туннелирование), что не требует высокого напряжения и позволяет сделать ячейки минимального размера. Однако именно процесс туннелирования заряда физически изнашивает эти ячейки, поскольку при помощи электрического тока заставляет электроны проникать в затвор, проходя сквозь барьеры из диэлектрика. Собственно, срок хранения информации в такой памяти декларируется достаточно длительный — 10 лет, но изнашивает микросхему памяти не чтение информации, а процессы стирания и записи, ведь для чтения через канал просто пропускается электрический ток, не изменяющий его структуры. Еще одно отличие — архитектура размещения ячеек и их контактов. Для памяти архитектуры NAND имеется контактная матрица, в пересечениях строк и столбцов которой располагаются транзисторы. Это сравнимо с пассивной матрицей в дисплеях :) Площадь микросхемы можно значительно уменьшить за счет размеров ячеек. Недостатки заключаются в более низкой по сравнению с NOR скорости работы в операциях побайтового произвольного доступа.

Сфера применения какого-либо типа флэш-памяти зависит в первую очередь от его скоростных показателей и надежности хранения информации. Адресное пространство NOR-памяти позволяет работать с отдельными байтами или словами (2 байта). В NAND ячейки группируются в небольшие блоки (по аналогии с кластером жесткого диска). Из этого следует, что при последовательном чтении и записи преимущество по скорости будет у NAND. Однако с другой стороны NAND значительно проигрывает в операциях с произвольным доступом и не позволяет напрямую работать с байтами информации. К примеру, для изменения одного байта требуется:

- считать в буфер блок информации, в котором он находится;
- в буфере изменить нужный байт;
- записать блок с измененным байтом обратно.

Если еще ко времени выполнения перечисленных операций прибавить задержки на выборку блока и на доступ, то получим отнюдь неконкурентоспособные с NOR показатели (именно для случая побайтовой записи!!). При последовательной записи/чтении NAND показывает значительно более высокие скоростные характеристики. Поэтому, а также из-за возможностей увеличения объема памяти без увеличения размеров микросхемы, NAND-флэш нашел применение в качестве хранителя больших объемов информации и для ее переноса. Наиболее распространенные сейчас устройства, основанные на этом типе памяти, это флэшдрайвы и карты памяти. Что касается NOR-флэша, то чипы с такой организацией используются в качестве хранителей программного кода (BIOS, RAM КПК), иногда реализовываются в виде интегрированных решений (ОЗУ, ПЗУ и процессор на одной мини-плате или в одном чипе).

Устройства хранения данных на основе flash-памяти состоят из собственно памяти и встроенного контроллера, который организует выполнение команд чтения-записи и блоков данных. Для работы с картой памяти на основе flash используется адаптер, который предоставляет только интерфейс между памятью и хостом, а функции обработки запросов на чтение-запись выполняет драйвер. Из-за разнообразия моделей flash-памяти USB нужны разные драйверы. ОС, как правило, предоставляет драйверы, работающие с большинством моделей flash. BIOS даже позволяет включать flash-накопитель в список устройств, с которых может производиться загрузка.

Преимущества и недостатки flash-памяти

Flash-память обладает как преимуществами, так и недостатками. Если говорить кратко, то все плюсы и минусы flash-устройств можно свести к нижеследующим двум перечням.

Преимущества flash-памяти:

1. Для хранения данных не требуется дополнительной энергии, то есть flash-память является энергонезависимым устройством.

2. Энергия требуется для записи данных, но затраты энергии при работе с flash-устройством минимальны.

3. Flash-микросхема позволяет многократно (но, увы, не бесконечно!) перезаписывать данные. То есть flash-память – перезаписываемое устройство хранения данных.

4. Накопитель на основе flash-микросхемы не содержит в себе никаких движущихся механических узлов и устройств, поскольку это твердотельная память. А раз так, то flash-устройства отличаются устойчивостью к механическим воздействиям.

5. Компактность.

6. информация, записанная на флэш-память, может храниться очень длительное время (порядка 10, а по некоторым данным, и до 100 лет). То есть flash-микросхема является устройством для долговременного хранения данных.

Недостатки flash-памяти:

1. Высокое соотношение цена/объём. flash-память стоит дороже, чем компакт-диски и компьютерные винчестеры. В связи с этим и объёмы флеш-накопителей не так велики.

2. Flash-память работает существенно медленнее, чем оперативная память на основе микросхем SRAM и DRAM. И даже по сравнению с жестким диском flash-накопитель является аутсайдером.

3. flash-память имеет ограничение по количеству циклов перезаписи. Предел колеблется от 10 000 до 1 000 000 циклов для разных типов микросхем. И хотя миллион операций записи/стирания – это совсем немало, однако наличие физического предела использования микросхемы памяти можно считать серьезным недостатком flash-устройств. Она может быть прочитана сколько угодно раз, но писать в такую память можно лишь ограниченное число раз (максимально — около миллиона циклов).

Производители памяти принимают меры для увеличения срока службы твердотельных накопителей: в первую очередь они стремятся обеспечить равномерность процессов записи/стирания по всем ячейкам массива, чтобы какие-то из них не подвергались б*о*льшему износу, чем другие. Равномерность нагрузки обеспечивается причем в основном программными способами. Например, применяется технология «выравнивания износа» (wear leveling) — часто изменяемые данные перемещаются по адресному пространству флэш-памяти, так что запись производится по разным физическим адресам. В каждый контроллер заложен свой алгоритм выравнивания, и сравнивать их эффективность у тех или иных моделей затруднительно, поскольку подробности реализации не разглашаются.

Отметим также, что в служебную область любого накопителя записывается таблица файловой системы, чтобы предотвратить сбои чтения данных на логическом уровне, возможные, к примеру, при некорректном отключении накопителя или при внезапном отключении электроэнергии. А поскольку в случае применения сменных устройств система не использует кэширования записи (на случай, если пользователь решит выдернуть флэшку сразу после обращения к ней), то от частой перезаписи особенно страдают области оглавления каталогов и таблицы размещения файлов. Например, если пользователь при однократном обращении переписал тысячу файлов и вроде бы всего лишь по разу использовал на запись те блоки, где они размещаются, то служебные области переписывались при каждом обновлении любого из файлов, то есть таблицы размещения файлов переписывались тысячу раз, а следовательно, занимаемые ими блоки выйдут из строя в первую очередь. Технология «выравнивания износа», конечно, работает и с такими блоками, но ее эффективность ограниченна.

В схемах флэш-памяти применяются различные средства и приемы:

- прерывание процессов записи при обращениях процессора для чтения (Erase Suspend). Без этого возникали бы длительные простои процессора, т. к. запись занимает достаточно большое время. После прерывания процесс записи возобновляется.
- внутренняя очередь команд, управляющих работой флэш-памяти, которая позволяет организовать конвейеризацию выполняемых операций и ускорить процессы чтения и записи
- оценка длины хранимых в ЗУ слов для согласования с различными портами ввода/вывода.
- введение режимов пониженной мощности на время, когда к устройству нет обращений, в том числе режима глубокого покоя, в котором мощность снижается до крайне малых значений.
- приспособленность к работе при различных питающих напряжениях (5 В; 3,3 В и др.). Сама схема «чувствует» уровень питания и производит необходимые переключения для приспособления к нему.
- введение в структуры памяти страничных буферов для быстрого накопления новых данных, подлежащих записи. Два таких буфера могут работать в режиме, называемом «пинг-понг», когда один из них принимает слова, подлежащие записи, а другой в это время обеспечивает запись своего содержимого в память. Когда первый буфер заполнится, второй уже освободится, и они поменяются местами.
- меры защиты от случайного или несанкционированного доступа. Флэш-память с адресным доступом, ориентированная на хранение не слишком часто изменяемой

информации, может иметь одновременное стирание всей информации (архитектура Bulk Erase) или блочное стирание (архитектура Boot Block Flash-Memory).

6.6 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Поясните понятия "сектор", "цилиндр", "дорожка".
2. Что такое дисковый контроллер ?
3. Зачем нужна таблица логических дисков ?
4. Поясните структуру логического диска.
5. Какие способы нумерации секторов диска Вам известны?
6. Поясните основные принципы пересылки данных с использованием DMA.
7. Физические принципы чтения и записи информации на оптические диски.
9. Типы файловых систем для оптических дисков.
10. Какими характеристиками обладают различные типы флэш-памяти?

6.7 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Сервометки нужны для того, чтобы:

   а) определить, каким образом физический диск разбивается на логические;

   б) иметь возможность форматирования только дорожек, отмеченных сервометками;

   в) контроллер мог постоянно получать информацию о текущем положении головок для обеспечения точного позиционирование головок по сигналу обратной связи;

   г) отмечать цилиндры на диске с заданным номером.
2. Имеется несколько утверждений относительно Master Boot Record:

   1) Прочитать MBR можно прерыванием int 25h, задав в качестве номера сектора 0;
   2) MBR возможно прочитать прерыванием int 13h;
   3) MBR находится на дорожке 0, стороне 0, в секторе 1;
   4) MBR содержит таблицу разделов, содержащую 2 элемента;
   5) Таблица разделов начинается с начала главной загрузочной записи.
   6) MBR содержит главную программу начальной загрузки и таблицу разделов.

   Из этих утверждений верны:

   а) 1,3,4; б) 2,4,6; в) 1,6; г) 2,3,6; д) 1,4,5; е) 2,3,5,6; ж) 2,3.
3. DMA предназначен для того, чтобы:

   а) реализовать низкоуровневое форматирование дискет;

б) осуществлять обмен данными между периферийными устройствами и оперативной памятью без посредничества центрального процессора;

в) обеспечивать высокую скорость чтения данных с диска или записи их на диск за счет занесения процессором данных в специальный кэш-буфер и передачи их по специально выделенному каналу.

4. FSINFO представляет собой:

а) файловую систему оптических дисков;

б) структуру хранения информации о FAT32;

в) тип архитектуры флэш-памяти.

5. Архитектура NAND эффективна при:

а) работе с большими массивами информации;

б) обращении к отдельным адресам.

Ответы : 1 - в; 2 - г; 3 – б, 4 – б, 5 -а.

7 ПОСЛЕДОВАТЕЛЬНЫЙ, ПАРАЛЛЕЛЬНЫЙ И USB ИНТЕРФЕЙСЫ ПЕРЕДАЧИ ДАННЫХ

7.1 ПОСЛЕДОВАТЕЛЬНАЯ ПЕРЕДАЧА ДАННЫХ

71.1 Основные понятия последовательной передачи данных. Структура сигнала

Последовательный порт используется для соединения процессора с устройствами ввода-вывода, которые передают данные по одному биту за раз. Важной особенностью интерфейсной схемы последовательного порта является то, что она способна передавать данные в последовательном режиме со стороны устройства и в параллельном режиме со стороны шины. Взаимное преобразование последовательных и параллельных форматов данных выполняется при помощи сдвиговых регистров, обладающих функцией параллельного доступа. Входной сдвиговый регистр принимает от устройства ввода-вывода последовательные биты. После получения всех 8 битов данных содержимое этого регистра загружается в регистр данных. Аналогичным образом данные из регистра данных загружаются в выходной сдвиговый регистр, откуда биты по очереди отправляются устройству ввода-вывода.

Последовательная передача данных может осуществляться как в синхронном, так и в асинхронном режимах передачи. В синхронном режиме посылка информации начинается с синхробайта, за которым следует поток информационных битов, т.е. данные передаются одним сплошным потоком. В асинхронном режиме информационные биты передаются в соответствии с протоколом передачи, обеспечивающем синхронизацию передающего и принимающего устройства.

В компьютере для организации асинхронной последовательной связи имеется специальный адаптер (RS 232C), обычно содержащий несколько COM-портов (до 4), через которые можно подключать внешние устройства. Каждому COM-порту соответствует группа регистров, через которые осуществляется работа с этим портом. Эти регистры располагаются, начиная с фиксированного базового адреса. Базовые адреса COM-портов записываются при инициализации в области данных BIOS, начиная с адреса 0040:0000h. Порт COM1 имеет базовый адрес ***3F8h,*** COM2 - ***2F8h,*** COM3 - ***3E8h,*** COM4 - ***2E8h***. В основе аппаратной реализации работы COM-порта лежит микросхема ***UART*** (Universal Asynchronous Receiver Transmitter) - универсальный асинхронный приемопередатчик. Через

порты этой микросхемы осуществляется как инициализация, так и обработка последовательной асинхронной передачи данных. Минимальный набор портов - 7, через которые адресуется 11 основных регистров.

UART-микросхема может вырабатывать аппаратное прерывание при изменении состояния COM-порта. Каждому COM-порту соответствует определенная линия ***IRQ*** в контролере аппаратных прерываний. Так, прерывания, поступающие по линии COM1 соответствуют ***IRQ4*** - int 0Ch, COM2 - ***IRQ3*** (int 0Bh).

В современных UART-микросхемах введена возможность внутренней буферизации принимаемых и передаваемых данных. Буферы реализованы по схеме FIFO.

К COM-порту могут подключаться различные устройства, например, мышь, модем.

UART-микросхема имеет следующие входные и выходные линии:

1) принимаемые данные Received Data (RD) - вход,
2) передаваемые данные Transmitted Data (TD) - выход,
3) запрос для передачи (Request To Send) (RTS) - выход,
4) сброс для передачи (Clear To Send) (CTS) - вход,
5) готовность выходных данных - (DTR) - выход,
6) готовность данных на входе - (DSR) - вход,
7) детектор принимаемого с линии сигнала - Data Carrier Detect (DCD) - вход,
8) индикатор вызова - Ring Indicator (RI) - вход.

Только две линии используются для передачи и приема данных, остальные служат для управляющих сигналов. Для различных устройств, подсоединяемых к COM-порту, используется разное подмножество линий.

При асинхронной передаче данных для ее инициализации необходимо описать структуру передаваемого сигнала, установить скорость передачи и определить, каким образом будет обрабатываться принимаемый сигнал.

При передаче информации с помощью асинхронного адаптера используется так называемый старт-стопный метод. Информационным битам обычно предшествует стартовый бит, позволяющий определить начало передачи данных и тем самым синхронизировать передающее и принимающее устройства. Это достигается за счет перевода линии передачи данных с уровня логической единицы на уровень логического нуля. После старт бита передаются биты данных (вначале младшие, а затем старшие). Количество информационных битов не более 8, может быть и меньше. За информационными битами может передаваться бит четности (чтобы в пакете битов общее количество единиц было четно). Завершают передачу сигнала один или два стоповых бита, после чего уровень линии передачи снова устанавливается в единицу до прихода следующего стартового бита (рисунок 7.1).

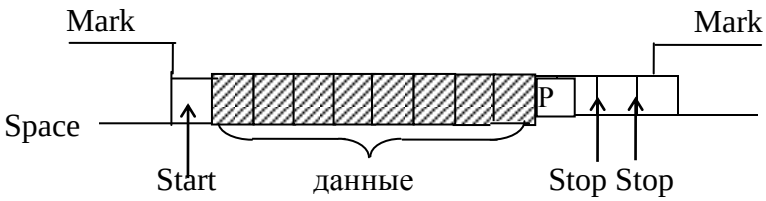


Рисунок 7.1 - Формат асинхронной передачи

Стоповые биты гарантируют определенную выдержку между соседними передачами данных. Старт-бит следующего посланного байта может посылаться в любой момент после окончания стоп-бита, то есть между передачами возможны паузы произвольной длительности. Старт-бит, имеющий всегда строго определенное значение (логический 0), обеспечивает простой механизм синхронизации приемника по сигналу от передатчика.

По получении сигнала о начале приема информации UART-микросхема автоматически осуществляет запись принимаемых битов в буферный регистр приемника с использованием специального сдвигового регистра, собирающего последовательность битов в байт. При передаче данные в буферном регистре данных передатчика поступают в сдвиговый регистр, с помощью которого выделяются отдельные биты, которые затем передаются.

При передаче данных подразумевается, что приемник и передатчик работают на одной скорости обмена. Единицей измерения скорости может быть бит в секунду (BPS, bit per second), бод (Baude), символ в секунду (character per second). BPS учитывает передачу служебных битов, а символ в секунду - нет. Скорость в бодах определяется количеством изменений сигнала на линии, для мыши BPS и боды совпадают, а для модема могут не совпадать за счет кодирования в одном сигнале нескольких бит при применении модуляции сигнала (амплитудная модуляция - кодирование за счет изменения амплитуды; частотная - изменение частоты; фазовая - изменение фазы сигнала; квадратурно-амплитудная - меняется фаза и амплитуда).

Частота передачи данных задается делителем частоты. Делитель частоты передачи (приема) - это число, на которое делится максимально возможная частота 115200 бит/с (частота системных часов), чтобы получить нужную частоту передачи. Например, при подключении мыши через COM-порт нужно установить скорость обмена 1200 бод, т.е. делитель частоты, равный 60h.

У приемника имеется внутренний генератор синхронизации, который генерирует внутренние импульсы синхронизации (стробирующие импульсы, стробы), по которым фиксируются принимаемые биты. В идеале эти стробы располагаются в середине битовых интервалов, что обеспечивает возможность приема данных и при некотором рассогласовании скоростей приемника и передатчика. Чем выше частота передачи, тем больше погрешность

привязки стробов к середине битового интервала. Вообще говоря, желательно, чтобы скорость передачи данных была одинаковой для передатчика и для приемника. Если передатчик и приемник работают с разной частотой, то может возникнуть так называемая ошибка кадрирования.

Обработка получаемых сигналов может быть организована двумя способами:

1) с использованием прерываний,

2) без использования прерываний путем циклического опроса некоторых регистров, характеризующих состояние COM-порта.

В первом случае при инициализации асинхронной последовательной передачи необходимо написать обработчик аппаратного прерывания (0Ch или 0Bh), размаскировать соответствующую линию IRQ в контроллере аппаратных прерываний, разрешить прерывания командной STI и указать в регистрах COM-порта, что мы избрали первый способ обработки сигналов.

Рассмотрим структуру регистров COM-порта и их назначение.

7.1.2 Регистры RS-232C, их назначение и организация доступа

Доступ к 11 регистрам RS-232C осуществляется через 7 портов. Порты будем адресовать не указанием их реального адреса, а через смещение от базового порта (base).

Доступ осуществляется в зависимости от:

1)функции, выполняемой COM-портом (прием или передача);

2) содержимого старшего бита регистра управления линией (доступ через порт base+3). Этот бит играет важную роль, поэтому он имеет свой идентификатор - DLAB;

3) выполнения чтения или записи в порт.

Базовый порт Base имеет двойное назначение в зависимости от бита DLAB. Если DLAB равен 0, то порт Base используется для доступа к регистру данных. Для передачи данных через этот порт записываются передаваемые данные, при приеме данных они считываются из базового порта. Если бит DLAB равен 1, то порт Base используется для доступа к младшему байту делителя частоты. Старший байт делителя при установленном DLAB записывается в регистр через порт Base + 1.

Порт Base + 1 используется либо для доступа к старшему байту делителя частоты (если DLAB = 1), либо для доступа к регистру управления прерываниями.

Формат регистра управления прерываниями:

бит 0 - разрешение/запрещение генерации прерывания при готовности принимаемых данных;

бит 1 - разрешение/запрещение генерации прерывания после передачи байта данных;

бит 2 - разрешает прерывания при получении от модема сигнала прерывания передачи Break (длинная строка нулей) или при возникновении ошибки;

бит 3 - разрешение прерывания после изменения состояния линий CTS, DSR, DSD, RI.

биты 4-7 должны быть равны 0.

При инициализации регистров RS-232 для приема сигналов от мыши нужно записать в регистр управления прерываниями 01h.

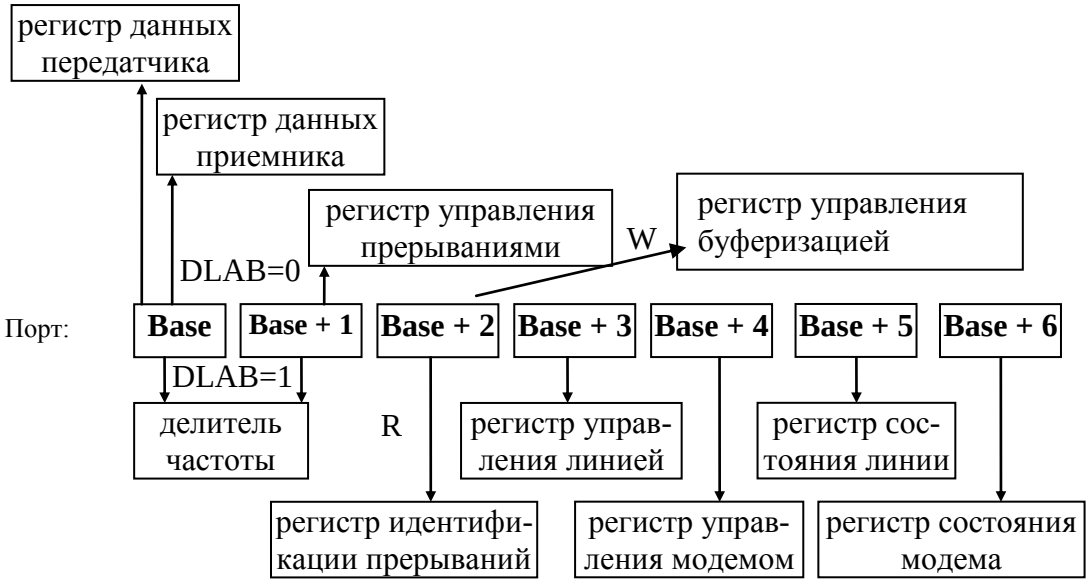


Рисунок 7.2 - Доступ к регистрам RS-232C

Порт Base + 2 используется для доступа к регистру идентификации прерывания (доступ по чтению) или к регистру управления буферизацией (доступ по записи). Обычно без использования буферизации микросхема UART генерирует прерывание каждый раз, когда передается или принимается очередной символ. Использование буферизации позволяет резко сократить количество прерываний, так как оно будет сгенерировано только при приеме или передаче определенного количества символов. Регистр управления буферизацией можно только записывать. Отметим, что бит 0 этого регистра разрешает или запрещает буферизацию, а биты 6 и 7 управляют количеством символов, необходимым для того, чтобы сгенерировать прерывание.

Регистр идентификации прерываний можно только читать через порт Base+2. Считав содержимое этого регистра, можно определить причину прерывания. Структура регистра идентификации прерываний:

бит 0 - если бит равен 1, то нет прерываний, ожидающих обслуживания; 0 - есть прерывание и оно не обработано;

биты 2,1 - Значение **00** - изменилось состояние линий модема (CTS, RI, DSD, DSR). Для сброса данной причины прерывания необходимо выполнить чтение регистра состояния линии. Значение **01** - пуст буфер передатчика, данные переданы, причина прерывания ликвидируется записью новых данных в регистр данных передатчика. Значение **10** - данные приняты и доступны для обработки, причина прерывания ликвидируется после чтения данных из регистра данных. Значение **11** - возникло переполнение приемника, ошибка четности, ошибка формата данных;

биты 3-7 - должны быть равно нулю.

Как правило, устранение причины прерывания выполняется чтением или записью соответствующего регистра. Возможно одновременное возникновение нескольких причин для прерывания, тогда адаптер упорядочивает их по приоритету следующим образом:

1) изменение в регистре состояния модема;
2) готовность данных;
3) пуст регистр передатчика
4) изменение состояния линии.

Порт Base + 3 служит для доступа к управляющему регистру, имеющему следующий формат:

биты 1,0 - определяют длину информационных битов в сигнале (00 - 5 битов, 01 - 6 битов, 10 - 7 битов, 11 - 8 битов);

бит 2 - количество стоповых битов в сигнале (0 - 1 стоп-бит, 1 - 2 стоп-бита);

биты 3-5 - используются при контроле на четность, при отсутствии контроля должны быть равны нулю;

бит 6 - если равен нулю, то служит признаком нормального функционирования адаптера, если установлен в 1, то для модема выдается сигнал Break;

бит 7 - бит DLAB (назначение см. выше).

Например, при подключении через последовательный порт устройств «мышь» длина информационной последовательности бит может быть равна 8 или 7, а количество стоповых бит равно 1, контроль четности не ведется.

Порт Base + 4 используется для доступа к регистру управления модемом. Этот регистр управляет состоянием выходных линий модема - DTR, RTS, а также линий OUT 1 и OUT 2, специфичных для модемов. Структура регистра управления модемом:

бит 0 - сигнал на линии DTR (готовность выходных данных);

бит 1 - сигнал на линии RTS (запрос на передачу);

бит 2 - линия OUT1, для некоторых модемов при установке этого бита происходит его аппаратный сброс, поэтому будем устанавливать его равным нулю;

бит 3 - линия OUT2, если бит установлен в 1, то UART может генерировать прерывание, если бит равен 0 - то нет;

бит 4 - служит для управления запуском диагностического теста;

биты 5-7 - должны быть равны 0.

При инициализации этого регистра для приема сигналов от мыши в порт должно быть записано число 00001011b.

Порт Base + 5 используется для доступа к регистру состояния линии. Регистр состояния линии позволяет определить причину ошибок, которые могут возникнуть при передаче и приеме информации COM-портом. Структура регистра состояния линии:

бит 0 - если бит равен 1, то данные получены и готовы для чтения информации из регистра-приемника;

бит 1 - сигнал об ошибке от переполнения. Был принят новый байт данных, а предыдущий еще не был считан и, отсюда следует, что теперь он потерян. Возможно, программа не успевает обрабатывать информацию.

бит 2 - ошибка четности, сбрасывается после чтения состояния линии;

бит 3 - ошибка синхронизации, возможна при отсутствии стоп-битов или ошибке кадрирования;

бит 4 - обнаружен запрос на прерывание передачи BREAK;

бит 5 - регистр данных передатчика пуст, все передано, можно записывать новый байт для передачи;

бит 6 - сдвиговый регистр передатчика пуст, возможен прием очередного байта;

бит 7 - ошибка тайм-аута. Возникает, если устройство не связано с компьютером. По истечении определенного интервала времени и при отсутствии возможности выполнить требуемую операцию возникает ошибка тайм-аута. Чтение содержимого этого регистра вызывает сброс ошибочных битов состояния, поэтому при инсталляции нужно читать из порта Base + 5.

7.1.3 Инсталлирующая секция в программе с разрешением прерываний от COM-порта

В секции инсталляции драйвера, обрабатывающего прерывания от COM-порта, необходимо выполнить следующие действия:

1. Запретить аппаратные прерывания командой cli.

2. Сохранить адрес старого обработчика прерывания от COM-порта, установить свой обработчик.

3. Перевести RS-232 в неактивное состояние, сбросив в 0 линии DTR и RTS в регистре управления модемом через порт Base + 4.

4. Считать содержимое регистра управления из порта Base + 3. Можно сохранить старую структуру сигнала. В прочитанном байте необходимо установить бит DLAB и вывести байт обратно в порт. Это нужно для получения доступа к делителю частоты. Для установки делителя частоты (например, для мыши, подключенной через COM-порт, это 60h) в порт Base нужно записать 60h, а в порт Base + 1 - 0h. После этого нужно установить свою структуру сигнала через порт Base + 3 (для мыши в этот порт нужно вывести 03h).

5. Считать через порт Base +1 содержимое регистра управления прерываниями, сохранить старое значение этого регистра. Для разрешения генерации прерываний при поступлении данных вывести в порт число 01h.

6. Размаскировать прерывание от COM-порта в регистре маски контроллера прерываний, предварительно считав старую маску через порт 21h, изменив в ней бит, соответствующий нужной линии IRQ и записать маску обратно. Старую маску также можно сохранить.

7. Сохранить старое значение регистра управления модемом, считанное через порт Base+4. Затем последовательно выводим в порт Base + 4 01h (устанавливаем сигнал на линии DTR), 03h (устанавливаем сигнал на линиях DTR RTS), 0Bh (устанавливаем DTR, RTS и разрешаем прерывания). Этот способ рекомендован в соответствующей документации, однако практика показывает, что можно сразу вывести в порт число 0Bh.

8. Очистить регистр состояния линии чтением из порта Base + 5, регистр состояния модема чтением из порта Base + 6, регистр данных чтением из порта Base.

9. Разрешить прерывания командой sti.

Примечание: сохраненные значения регистров COM-порта нужно восстановить при деинсталляции одновременно с восстановлением старого обработчика прерываний от COM-порта.

8.1.4 Пример обработки прерывания от устройства "мышь", подключенного через последовательный порт

Исторически мышь можно подключать к компьютеру через последовательный интерфейс, через интерфейс PS/2, через USB-интерфейс. Подключение мыши через последовательный интерфейс в настоящее время не используется, в большинстве

современных компьютеров мышь подключается через USB. Однако поскольку наиболее просто рассмотреть возможности последовательной передачи данных именно для мыши, то остановимся на обработке ее сигналов более подробно.

Для передачи данных от «последовательной» мыши в компьютер используются стандартные протоколы передачи данных через COM-порт. Когда Вы нажимаете или отпускаете кнопку на мыши или двигаете ее, то микросхема, стоящая в мыши, обрабатывает это событие и посылает в компьютер пакет байтов с информацией о событии. Приход байтов вызывает в компьютере аппаратные прерывания (IRQ3 или IRQ4), обрабатываемые драйвером мыши. Мышь при подключении через COM-порт передает информацию либо при нажатии кнопок, либо при перемещении мыши на величину, большую "микки" (1 микки = 1/200 дюйма). При перемещении мыши перемещается ее курсор, представляющий собой в текстовом режиме прямоугольник, соответствующий знакоместу, а в графическом режиме курсор мыши - это стрелка или другое изображение.

Регулировать перемещение курсора позволяют так называемые пороги чувствительности, определяющие число микки, которые нужно принять для перемещения на 1 пиксел по горизонтали или вертикали. Минимально возможное значение порога чувствительности - 1 микки, однако при таком значении порога курсор будет очень трудно установить точно в заданное место на экране, зато скорость перемещения курсора по экрану высока. Для разрешения этой проблемы используется алгоритм так называемого баллистического курсора. Суть алгоритма заключается в том, что при достижении мышью порога удвоенной скорости каждое перемещение мыши удваивается. При инициализации порог удвоенной скорости составляет 64 микки/сек.

Смещения поступают в мышиных координатах, т.е. в микки. В обработчике прерывания микки пересчитываются в пикселы графического режима или знакоместа текстового режима, чтобы определить место для вывода мышиного курсора.

Структура информационного пакета, передаваемого мышью, различна для мышей различных типов.

1) Mouse Systems (или Mouse Mode) - трехкнопочная мышь, передает 5 байтов в составе пакета. 1 байт - нажатие кнопок; 2, 3 - определяют величины Δx и Δy, на которые увеличиваются координаты курсора мыши по x и по y при ее перемещении; 4, 5 - “добавки” к Δx и Δy, если перемещения большие:

```
байт 1:    1 0  0  0  0 LB MB RB
байт 2:    X7 X6 X5 X4 X3 X2 X1 X0
байт 3:    Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
байт 4:    X7 X6 X5 X4 X3 X2 X1 X0
байт 5:    Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
```

где LB, MB, RB - состояние левой, средней и правой кнопок, при нажатии соответствующей кнопки бит устанавливается в 1. Остальные значащие биты характеризуют относительное перемещение мыши со времени последней посылки информационного пакета. Смещение задается в дополнительном коде в двух частях, которые надо суммировать. При этом положительными считаются перемещения вправо и вверх.

2) Microsoft Mouse - двухкнопочная мышь, передает три байта. 1байт - нажатие и отпускание кнопок и биты 6,7 Δx и Δy, 2, 3 байты - соответствующие биты 0 - 5 Δx и Δy.

```
байт 1:    1 LB RB Y7 Y6 X7 X6
байт 2:    0 X5 X4 X3 X2 X1 X0
байт 3:    0 Y5 Y4 Y3 Y2 Y1 Y0
```

где LB, RB - состояние левой и правой кнопок, при нажатой кнопке бит равен 1. Остальные значащие биты характеризуют перемещение в дополнительном коде со времени последней посылки пакета байтов. При этом перемещения по x и по y считаются положительными, если произошло перемещение вправо (для x) или вниз (для y).

Рассмотрим вычисление новых координат в микки мышиного курсора в предположении, что Δx и Δy уже собраны в один байт. Вычисление новой x-координаты происходит одинаково для Mouse Systems и Microsoft Mouse. Нужно проверить, положительна или отрицательна Δx по значению старшего бита (если 0, то Δx больше 0).

В случае положительного смещения Δx нужно прибавить к старому значению координаты x (в микки), в случае отрицательного смещения у числа Δx нужно изменить знак и отнять от координаты x. Вычисление новой y-координаты в микки для Mouse Systems и Microsoft Mouse различно. Для Microsoft Mouse положительное Δy свидетельствует о перемещении мыши вниз (т.е. $y_{нов} = y_{ст} + \Delta y$), для Mouse Systems положительное Δy указывает на перемещение вверх (т.е. $y_{нов} = y_{ст} - \Delta y$).

Для перехода от координат в микки к координатам текстового режима нужно разделить мышиные координаты на некоторый коэффициент (обычно 8, но его можно менять для регулировки чувствительности мыши). В графическом режиме можно считать, что микки - это 1 или 2 пиксела.

Для организации перемещения мышиного курсора по экрану в текстовом режиме обычно используется изменение цвета фона под символом, где находится курсор. Программно это можно реализовать введением специальной XOR-маски (например, 07Fh), которая используется для восстановления старого значения атрибута при переносе курсора мыши на новое место, так и для изменения атрибута знакоместа, где будет располагаться новый курсор.

```
mov  al,es:[bx+1];читаем байт атрибутов
xor  al,Maska    ;xor байт атрибутов al с маской
mov  es:[bx+1],al;пишем байт обратно.
```

Рассмотрим **общий алгоритм обработчика прерываний** от мыши (int 0Ch или int 0Bh):

1. читаем регистр состояния линии из порта Base+5. Если бит 1 установлен, то ошибка в данных, нужно сбросить регистр данных чтением из порта Base, сбросить в 0 счетчик принимаемых байтов данных и перейти на пункт 4. Если бит 0 регистра состояния линии равен 0, то имеем готовность принятых данных и на пункт 2, иначе сброс данных, счетчика и переход на пункт 4.
2. читаем из порта Base байт ***al***, дополняем байт до слова командой ***cbw*** для удобства обработки, увеличиваем счетчик байтов в пакете.
3. анализируем, какой байт поступил и обрабатываем. При обработке перемещений не забываем затереть старый курсор перед выводом нового.
4. посылаем в контроллер прерываний сигнал ***EOI*** и выходим из обработчика.

При написании обработчика нужно грамотно использовать команды **`sti`** и **`cli`**.

7.2 ПАРАЛЛЕЛЬНЫЙ ИНТЕРФЕЙС

Порт параллельного интерфейса был введен в архитектуру компьютера для подключения принтера, название LPT означает LinePrinTer. В настоящее время через этот порт может подключаться не только матричный принтер, но и струйные, лазерные принтеры, сканер.

7.2.1 Стандартный LPT-порт.

Для управления принтером на низком уровне (без использования int 17h BIOS) необходимо иметь доступ к адресному пространству адаптера параллельного интерфейса, представляющему собой набор регистров, доступ к которым осуществляется через набор портов. Регистры порта адресуются относительного базового адреса порта (для LPT1 - ***378h***; LPT2 - ***278h***). Стандартный параллельный порт (SPP) имеет внешнюю 8-битную шину данных, 5-битную шину сигналов состояния и 4-битную шину управляющих сигналов.

Работа стандартного параллельного порта происходит в соответствии с интерфейсом Centronics, предусматривающим следующие сигналы (или линии):

1) линия строба данных (STROBE), используется для организации побайтного вывода во внутренний буфер принтера и служит для регулировки потока байтов;

2) линии данных - 8 линий;

3) линия Busy - линия занятости, используется при оценке возможности вывода очередного байта;

4) линия PE (Paper End) - сигнал о конце бумаги;

5) линия Select - сигнал о включении принтера;

6) линия Error - сигнал об ошибке принтера;

7) линия Init - инициализация, сигнал аппаратного сброса принтера;

8) линия AutoLF - линия автоматического перевода строки (при получении CR (возврат каретки) принтер выполняет функцию LF);

9) линия ACK (Acknowledge) - линия подтверждения приема байта. По отрицательному перепаду сигнала на этой линии может быть выработан обычно запрос на аппаратное прерывание (IRQ7 или IRQ5). BIOS это прерывание не обслуживает.

10) SLCT IN - выбор принтера.

Стандартный порт имеет три 8-битных регистра, расположенных по соседним адресам, начиная с базового адреса порта BASE.

Через порт BASE организуется доступ к регистру данных, через который передаются как данные для печати, так и команды принтера. Команды бывают однобайтовые и многобайтовые. Многобайтовые команды начинаются с кода Escape и называются Esc-последовательностями. Эти команды уникальны для каждого типа принтера, поэтому их перечень приводить нецелесообразно. Отметим только некоторые возможности, реализуемые для матричных принтеров.

Для установки различных спецификаций, относящихся к формату страницы, стилю шрифта и т.п., на принтер наряду с данными посылаются специальные управляющие коды. Некоторые из кодов - это однобайтные коды из числа первых 32-х кодов из набора ASCII. Они инициируют такие простые действия принтера, как перевод строки или прогон страницы. Однако большинство спецификаций печати устанавливается посылкой Esc-последовательностей, в которых один или более кодовых байтов следуют за символом Esc, ASCII-код которого равен 27. Начальный код Esc информирует о том, что символы, которые следуют за ним, следует интерпретировать как команду, а не как данные. Такие Esc-последовательности обычно не имеют символа-ограничителя, поскольку принтер "знает" длину каждой последовательности. Только в некоторых случаях, когда последовательность может иметь разную длину, требуется ограничивающий символ, в качестве которого всегда используется код ASCII 0. Почти во всех случаях спецификации, установленные этими

кодами, действуют до тех пор, пока они не будут явно отменены. Как только будет получен код, например, печати курсивом, то оно будет осуществляться до тех пор, пока не будет послан код отмены этого режима печати. Если произошла ошибка на принтере и принтер был выключен и включен, то необходимо снова устанавливать все спецификации печати.

Большинство кодов, устанавливающих спецификации принтера, перемешаны с данными, на которые они действуют. Например, данные для слова, которое должно быть выделено жирным шрифтом, должны предваряться Esc-последовательностью, включающей жирный шрифт, и завершаться Esc-последовательностью, выключающей его.

Принтер всегда находится в текстовом режиме, до тех пор, пока он специально не переведен в графический режим. Команда, устанавливающая графический режим, должна сообщать какое число байтов графических данных будет передано (но не больше одной строки) и после того, как это число байтов будет выведено как графическое изображение, принтер вернется в текстовый режим. Обычно за кодом, устанавливающим графический режим следуют 2 байта, указывающие, какое число графических байтов будет передано. За этими двумя байтами должны сразу следовать байты данных, формирующиеся по следующим правилам. Каждый байт - это число, определяемое битами, соответствующими восьми вертикальным точкам одной позиции в строке изображения. Младший бит (1) соответствует низу колонки, а старший бит (128) - верху. Например, чтобы напечатать пирамиду, необходимо послать на печать сначала байт, у которого установлен только нижний бит, затем байт у которого установлены 2 нижних бита и т.д. После восьмого байта необходимо расположить те же байты в обратном порядке. Значение первого байта будет 1, второго - 3 (1+2), третьего - 7 (1+2+4), четвертого - 15 (1+2+4+8) и т.д.

Отметим также такие возможности принтеров, реализуемые с помощью ESC-последовательностей, как изменение интервалов печати между строками, позиционирование печатающей головки, изменение шрифта печати.

Через порт BASE+1 можно получить доступ к регистру состояния (Status Register), имеющему следующую структуру:

бит 3 - сброс в 0 - сигнал об ошибке печати;

бит 4 - 1/0 - принтер ONLINE/OFFLINE (отображение состояния линии Select);

бит 5 - 1 - нет бумаги (отображение состояния линии PaperEnd), 0 - есть;

бит 6 - 0 - готовность к приему следующего символа (отображение состояния линии ACK), подтверждает прием символа;

бит 7 - инверсное отображение состояния линии BUSY, поэтому 1 - разрешает вывод очередного байта и служит сигналом готовности принтера.

Через порт BASE+2 осуществляется доступ к регистру управления. Формат регистра:

бит 0 - 1 - получен сигнал стробирования данных, 0 - нет сигнала стробирования;

бит 1 - 1 - автоматический перевод строки по приему байта возврата каретки;

бит 2 - 0 - сигнал аппаратного сброса принтера (низкий уровень линии Init);

бит 3 - 1 - разрешает работу принтера в соответствии с рассмотренным выше интерфейсом Centronics;

бит 4 - 1 - разрешает прерывания по спаду сигнала на линии Ack и служит для запроса следующего байта.

При инициализации принтера обязательна проверка регистра состояния (online, бумага, готовность), после печати - проверить ошибки.

Общий алгоритм вывода байта на принтер через стандартный LPT-порт:

1. Чтение регистра состояния и проверка битов 6 и 7 (предыдущий байт данных должен быть получен, передан в буфер, и принтер должен быть готов к приему следующего байта). Этот шаг обычно зацикливается до того момента, пока не будут выполнены указанные условия.

2. Байт данных записывается через порт BASE в регистр данных.

3. В регистре управления устанавливается стробирующий сигнал, причем это делается при предварительном считывании регистра управления и изменении только бита, отвечающего за стробирование. При получении стробирующего сигнала байт из регистра данных запишется во внутренний буфер принтера, поэтому необходима организация некоторой задержки для выполнения записи. Линия BUSY устанавливается в 1 и принтер не может принимать другие символы.

4. Снятие стробирующего сигнала.

Стандартный LPT-порт поддерживается BIOS. Базовые адреса LPT-портов хранятся по следующим адресам: 0:0408h - для LPT1, 0:040Ah - для LPT2, 040Ch - для LPT3, 040Eh - для LPT4. Если по указанным адресам хранится 0, это значит, что соответствующий LPT-порт отсутствует. Поиск имеющихся фактически портов BIOS осуществляет путем вывода по базовому адресу тестового байта, после чего осуществляется чтение из порта, прочитанное значение сравнивается с записываемым, если они совпали, то LPT-порт найден. Найденные LPT-порты инициализируются BIOS, в регистре управления формируется и снимается сигнал по линии INIT, затем записывается 0Ch. Для LPT-портов по адресам 0:0478h, 0:0479h, 0:047Ah, 0:047Bh BIOS хранит время тайм-аута. Если символ не удается вывести из-за того, что сигнал на линии BUSY не снимается в течение времени тайм-аута, то стробирующий сигнал не формируется.

Таким образом, вывод одного байта требует нескольких операций ввода/вывода через порты стандартного параллельного порта, что приводит к загрузке процессора при низкой

скорости обмена. Скорость вывода данных через стандартный LPT-порт можно повысить до 100-150 Кбайт/с, что недостаточно для вывода на лазерный принтер.

7.2.2 Расширенные режима работы LPT-порта

Согласно стандарту на параллельный интерфейс IEEE 1284 определено 5 режимов работы LPT-порта, один из них - стандартный вывод, остальные режимы используются для расширения функциональных возможностей LPT-порта. К этим режимам относятся:

1) Nibble Mode - ввод байта в 2 цикла по 4 бита, в этом режиме можно организовать двунаправленный обмен данными. Этот режим работает на всех LPT-портах, но применять его целесообразно только в случае, если данные имеют небольшой объем. Данные при этом передаются по линиям шины состояния.

2) Byte Mode - только для портов, допускающих чтение выходных данных. Передача осуществляется по шине данных.

3) Enhanced Parallel Port (EPP) Mode.

4) Extended Capabitity Port (ECP) Mode.

В режимах 1 и 2 сигналы интерфейса являются программно-управляемыми, в режимах 3 и 4 сигналы интерфейса генерируются аппаратно. Рассмотрим режимы 3 и 4 более подробно.

Для режима **EPP Mode** (улучшенный параллельный порт) характерно наличие двунаправленных циклов обмена компьютера с периферией 2 типов:

а) циклы обмена данными (чтение и запись),

б) адресные циклы, осуществляющие чтение и запись адресной и управляющей информации.

Для организации этих циклов к трем регистрам стандартного порта (SPP) добавлены:

а) регистр адреса EPP (BASE+3),

б) регистр данных EPP (BASE+4,)

в) регистры по BASE+5, BASE+6, BASE+7, используемые для 16 и 32-битных операций ввода/вывода.

Назначение регистров стандартного порта сохранено, однако нужно помнить, что биты 0,1 и 3 регистра управления при записи должны быть равны нулю. Иначе может возникнуть конфликт с генерируемыми аппаратно сигналами на линиях EPP. Некоторые адаптеры имеют специальные средства, блокирующие программную модификацию данных битов.

Чтение и запись регистров данных и адреса расширенного параллельного порта приводит к генерации так называемых связанных циклов обмена данными между процессором и LPT-портом. При записи данные (или адреса) помещаются на выходную шину LPT-порта, выдается стробирующий сигнал для данных (или адреса), ожидается подтверждение об обмене, после получения которого стробирующий сигнал снимается.

Возможны операции с 32-битными данными, при этом регистры по BASE+5, BASE+6, BASE+7 сметаются расширениями регистра данных. Характерной чертой EPP является отсутствие буферизации, обмен осуществляется в реальном времени.

Режим **ECP Mode (Extended Capabitity Port** - порт с расширенными возможностями) используется для связи с принтерами DeskJet моделей 6XX, LaserJet, начиная с 4xx, со сканерами. Как и EPP, ECP обеспечивает двунаправленные циклы чтения и записи данных и циклы чтения и записи командной информации. Командная информация подразделяется на информацию о каналах ECP и информацию о компрессии данных. Компрессия данных при передаче осуществляется по методу RLE (Run Length Encoding), это позволяет сократить время на передачу данных. При передаче растровых изображений используется счетчик компрессии, равный числу повторяющихся байтов без единицы. Так как растровые изображения имеют длинные строки повторяющихся байтов, то коэффициент сжатия при их передаче за счет использования счетчика компрессии может составлять 64:1.

В режиме ECP введено понятие канальной адресации для физического устройства, подключаемого к LPT-порту. Если, например, в SPP-режиме к LPT-порту подключено комбинированное устройство, сочетающее функции принтера и факса, и принтер будет занят печатаньем, а в это время придет факс, то пока не завершится печать, факс не будет обслужен. В ECP-режиме принтеру и факсу соответствуют различные логические каналы LPT-порта, с которыми работает драйвер LPT-порта.

В режиме ECP обмен программы с периферийным устройством осуществляется через специальный буфер, работающий по принципу FIFO. Периферийное устройство обменивается информацией с FIFO-буфером аппаратно с использованием ECP-адаптера, а обмен программы-драйвера с FIFO-буфером может осуществляться как с использованием ДМА, так и через регистры ECP. Протокол EPP позволяет драйверу чередовать циклы прямой и обратной передачи, не фиксируя факт смены направления. В ECP компьютер по специальной линии запрашивается выполнение реверса, после чего выполнение реверса передачи подтверждается периферийным устройством.

Для протокола ECP есть стандарт на регистры и стандарт на режимы работы адаптера LPT-порта. Режим работы указан в битах 7-5 регистра расширенного контроля (доступ через порт BASE+402h).

Перечень режимов и биты:

1. SPP mode - стандартный параллельный порт - 000

2. Byte mode - 001

3. Fast Centronics (однонаправленный с использованием FIFO-буфера и ДМА) - 010

4. ECP mode - 011

5. EPP mode - 100

6. Test mode - тестирование 110

7. Configuration mode - 111 - режим доступа к конфигурационным регистрам.

Каждому режиму ECP соответствует свое подмножество регистров. Переключение режимов осуществляется записью в регистр расширенного контроля. Все регистры делятся на две группы: первая группа доступна через порты со смещением 0-2 от базового адреса, вторая - через порты со смещением 400-402h от базового. Вторая группа адресов не используется традиционными драйверами.

По умолчанию в регистр расширенного контроля записывается номер режима 000 или 001. Из этих режимов можно переключиться в режимы с номерами 010-111, однако переход из режимов 010-111 возможен только в 000 или 001 при условии завершения обмена по DMA и опустошения FIFO-буфера.

Набор регистров LPT-порта с расширенными возможностями приведен в таблице 8.1.

Главным управляющим регистром в ECP-протоколе является ECR, имеющий следующий формат:

биты 5-7 - режим ECP;

бит 4 -1 - запрет прерываний по ошибке, 0 - запрос на прерывание по ошибке;

бит 3 - при установке разрешает обмен по каналу DMA;

бит 2 - запрет сервисных прерываний (по окончании цикла DMA, по заполнению/опустошению FIFO-буфера (если не используется ДМА), по ошибке работы с FIFO-буфером;

бит 1 - установка бита сигнализирует о том, что FIFO-буфер заполнен полностью;

бит 0 - установка означает, что FIFO-буфер опустошен.

Единицы в младших битах означают ошибку работы с FIFO.

Таблица 8.1 - Назначение и адресация регистров в различных режимах работы LPT-порта

Смещение от базового адреса	Имя	Режимы	Назначение
000	DR	000-001	регистр данных
000	AFIFO	011	регистр для командной информации, впоследствии помещаемой в FIF0-буфер
001	SR	все	регистр состояния

002	CR	все	регистр управления
400	SDFIFO	010	регистр данных FIFO-буфера в режиме 010
400	DFIFO	011	регистр данных для обмена с буфером FIFO в режиме 011
400	TFIFO	110	регистр тестирования взаимодействия FIFO и прерываний. Данные попадают в этот регистр программным способом или через DMA. Адаптер отрабатывает операции передачи на максимальной скорости, которую можно определить таким способом
400	CFA	111	регистр конфигурации A
401	CFB	111	регистр конфигурации B
402	ECR	все	регистр расширенного контроля

Рассмотрим использование этих регистров при работе ECP-протокола передачи/приема данных. При передаче в периферийное устройство поток данных и поток команд для периферийного устройства помещается в регистры DFIFO и AFIFO, откуда данные и команды помещаются в буфер. При передаче данных их компрессия по методу RLE выполняется программно. Для передачи подряд более 2 одинаковых байтов в регистр AFIFO записывается счетчик компрессии (младшие 7 бит), а в бит 7 - записывается 0; в регистр DFIFO помещается сам байт. По значениям этих двух байтов осуществляется аппаратная декомпрессия и в FIFO-буфер помещаются декомпрессированные данные. При передаче из периферийного устройства информация в FIFO-буфер из этого устройства помещается аппаратно, а из FIFO-буфера информация попадает в регистр DFIFO.

7.3 USB-ИНТЕРФЕЙС

Последовательный интерфейс, как отмечено выше, позволяет объединить устройства с использованием всего одной или двух пар проводов, однако при этом имеет плохую помехозащищенность и отсутствие гальванической развязки, что не позволяет использовать высокие скорости обмена и «горячее» подключение устройств. Кроме того, увеличение числа устройств, подключаемых к компьютеру, привело к ситуации, когда задача конфигурирования компьютера стала очень сложной из-за наличия практически для каждого устройства своего специализированного разъема, своего протокола обмена, и, как правило, своей линии IRQ. Решением этой проблемы является создание единого и универсального интерфейса для компьютерного «железа».

USB (Universal Serial Bus – универсальная последовательная шина) в настоящее время является промышленным стандартом расширения архитектуры компьютера, ориентированным на подключение различных периферийных устройств.

7.3.1. Физическая и логическая архитектура шины USB

Архитектура шины USB предполагает подключение USB-устройств к компьютеру, который является главным управляющим устройством и называется *хостом*. Подключение устройств к хосту может осуществляться как напрямую, так и через так называемые *хабы* или *концентраторы*, обеспечивающие подключение группы устройств, среди которых также могут быть хабы. Сам компьютер имеет встроенный хаб, который называется корневым.

Физическая архитектура шины USB строится в соответствии со следующими правилами (рисунок 8.3):

- физическое соединение устройств между собой осуществляется по топологии многоярусной звезды, вершиной которой является корневой хаб;

- центром каждой звезды является хаб;

- соединение допускается между хабом и периферийным устройством, способным к обмену данными по шине USB (называется *функцией*), между хабом и другим хабом, между хостом с хабом или с функцией;

- к каждому порту хаба может быть подключено периферийное устройство или другой хаб, при этом допускается до 5 уровней каскадирования хабов, не считая корневого.

Логическая же архитектура USB представляет собой звезду, вершинами которой являются являются так называемые *конечные точки* – хабы и периферийные устройства, с которыми ведется обмен данными.

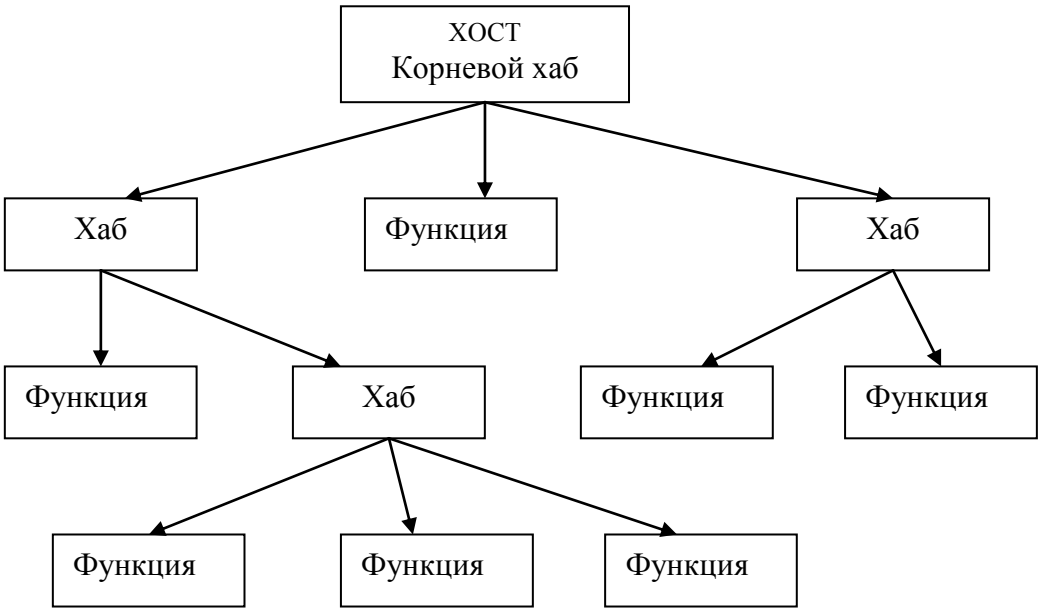


Рисунок 7.3 – Физическая архитектура USB

Рассмотрим составляющие USB более подробно. Шина USB состоит из следующих элементов:

1) **Хост-контроллер**.

Представляет собой главный управляющий контроллер, входящий в состав системного блока компьютера и управляющий работой всех устройств, подключенных к шине USB. На шине USB допускается только 1 хост. Отметим, что системный блок компьютера содержит один или несколько хостов, каждый из которых управляет отдельной шиной USB.

2) **Устройство** представляет собой хаб, функцию или их комбинацию (Compound Device).

3) **Порт** – точка подключения.

4) **Хаб** или **концентратор**.

Это устройство, обеспечивающее дополнительные порты на шине USB. Можно считать, что хаб является вершиной дерева, листья которого представляют собой хабы или функции. При этом происходит обмен данными между одним так называемым восходящим (upstream) портом и несколькими нисходящими (downstream). Хаб умеет распознавать подключение и отключение устройств от своих портов. Каждый из портов может быть запрещен или разрешен, и сконфигурирован на полную или ограниченную скорость обмена.

Хаб выполняет коммутацию сигналов и выдачу питающего напряжения, а также уведомляет хост об изменениях в подключенных устройствах. Для этой цели

5) **Корневой хаб** – хаб, входящий в состав хоста.

6) **Функция**.

Представляет собой периферийное устройство или его отдельный блок, способный к обмену по шине USB. Каждая функция перед использованием должна быть сконфигурирована хостом, ей должна быть выделена полоса в канале и выбраны опции конфигурации.

7) **Логическое устройство** - - это набор конечных точек. В отличие от шин ISA и PCI, где программа работает с устройствами через обращения по физическим адресам ячеек памяти, портов ввода-вывода, прерываниям и каналам DMA, взаимодействие приложений с устройствами USB выполняется только через программный интерфейс, предоставляемый контроллером USB.

7.3.2 Модель передачи данных по шине USB

Каждому логическому устройству USB (как функции или хабу) назначается при подключении свой адрес о 1 до 127, уникальный на данной шине USB. Каждая конечная точка в логическом устройстве имеет свой номер и характеризуется следующими параметрами:

- частота обращения к шине и допустимые задержки обслуживания;
- требуемая полоса пропускания канала;
- максимальные размеры передаваемых и принимаемых пакетов;
- тип и направление передачи.

Точка с номером 0 в устройстве используется для инициалиации, управления устройством и опроса его состояния. Дополнительно к нулевой устройства могут иметь до 15 точек ввода и 15 точек вывода. Именно эти точки и предоставляют возможности обмена данными. Для того чтобы использовать дополнительные точки, их надо сконфигурировать путем назначения согласованного с ними *канала*.

Каналом (pipe) в USB называется модель передачи данных между хост-контроллером и конечной точкой. Существует два типа каналов – потоки и сообщения.

Поток (stream) доставляет данные от одного конца канала к другому, он всегда однонаправленный. Поэтому один и тот же номер конечной точки может использоваться двумя каналами – ввода и вывода. С помощью потока могут быть реализованы передача массивов, изохронный обмен и передачи-прерывания. Передачи массивов данных имеют

самый низкий приоритет, доставка гарантирована, при ошибке выполняется повтор. Время передачи не ограничено. Передачи массивов используются при обмене с принтерами, сканерами, флэш-дисками. Передачи-прерывания – это короткие передачи, которые имеют случайный характер, гарантированную доставку, и ограниченное время доставки. Примером использования передач-прерываний является передача информации от клавиатуры или мыши. Изохронные передачи представляют собой передачи в реальном режиме времени, занимающие до 70% пропускной способности шины. В случае ошибки изохронные данные не повторяются, неверные данные просто игнорируются. Примерами изохронных передач данных являются передачи с видеокамер, цифровых аудиоустройств, CD и DVD.

Сообщения служат для посылки хостом запроса к конечной точке с целью получения информации об ее состоянии.

Все обмены (транзакции) с устройствами USB состоят из 2-3 пакетов. Каждая транзакция планируется и начинается по инициативе контроллера, который посылает пакет-маркер. Этот пакет содержит тип и направление передачи, адрес устройства USB и номер конечной точки. В каждой транзакции возможен обмен только между конечной точкой адресуемого устройства и хостом. Адресуемое устройство после приема пакет-маркера распознает в нем свой адрес и готовится к обмену. Оно передает пакет данных или уведомление об их отсутствии. После успешного приема пакета приемник посылает передатчику так называемый пакет подтверждения или квитирования (handshake packet).

Хост-контроллер организует обмены с устройствами следующим образом. Он циклически (с периодом около 1 мс) формирует *кадры* (frames), которые обслуживают транзакции с имеющимися на шине устройствами. Каждый кадр начинается с посылки маркера SOF (Start Of Frame), который является синхронизирующим сигналов для всех устройств, включая хабы. В конце кадра передается маркер EOF (End Of Frame), на время которого запрещаются передачи по направлению к контроллеру. В промежутке времени между SOF и EOF и происходят транзакции с устройствами, причем в первую очередь происходит посылка сообщений и обслуживаются передачи-прерывания. Оставшееся свободное время кадра заполняется передачами массивов.

Периферийное устройство не может по собственной инициативе выдавать на шину какую-либо информацию и не может самостоятельно посылать запросы.

Архитектура USB предусматривает внутреннюю буферизацию всех устройств, причем чем большую полосу пропускания требует устройство, тем больше должен быть его буфер.

Как было отмечено выше, данные передаются в виде пакетов-маркеров. В интерфейсе USB испольуется несколько разновидностей пакетов:

- пакет-признак (token paket) описывает тип и направление передачи данных, адрес устройства и порядковый номер конечной точки (КТ - адресуемая часть USB-устройства); пакет-признаки бывают нескольких типов: IN, OUT, SOF, SETUP;

- пакет с данными (data packet) содержит передаваемые данные;

- пакет согласования (handshake packet) предназначен для сообщения о результатах пересылки данных; пакеты согасования бывают нескольких типов: ACK, NAK, STALL.

В интерфейсе USB используются несколько типов пересылок информации:

- управляющая пересылка (control transfer) используется для конфигурации устройства, а также для других специфических для конкретного устройства целей;

- потоковая пересылка (bulk transfer) используется для передачи относительно большого объема информации;

- пересылка с прерыванием (iterrupt transfer) испольуется для передачи относительно небольшого объема информации, для которого важна своевременная его пересылка. Имеет ограниченную длительность и повышенный приоритет относительно других типов пересылок;

- изохронная пересылка (isochronous transfer) также называется потоковой пересылкой реального времени. Информация, передаваемая в такой пересылке, требует реального масштаба времени при ее создании, пересылке и приеме.

Потоковые пересылки характеризуются гарантированной безошибочной передачей данных между хостом и функцией посредством обнаружения ошибок при передаче и повторного запроса информации.

Когда хост становится готовым принимать данные от функции, он в фазе передачи пакета-признака посылает функции IN-пакет. В ответ на это функция в фазе передачи данных передает хосту пакет с данными или, если она не может сделать этого, передает NAK- или STALL-пакет. NAK-пакет сообщает о временной неготовности функции передавать данные, а STALL-пакет сообщает о необходимости вмешательства хоста. Если хост успешно получил данные, то он в фазе согласования посылает функции ACK-пакет. В противном случае транзакция завершается.

Когда хост становится готовым передавать данные, он посылает функции OUT-пакет, сопровождаемый пакетом с данными. Если функция успешно получила данные, он отсылает хосту ACK-пакет, в противном случае отсылается NAK- или STALL-пакет.

Управляющие пересылки содержат не менее двух стадий: Setup-стадия и статусная стадия. Между ними может также располагаться стадия передачи данных. Setup-стадия используется для выполнения SETUP-транзакции, в процессе которой пересылается информация в управляющую КТ функции. SETUP-транзакция содержит SETUP-пакет, пакет

с данным и пакет согласования. Если пакет с данными получен функцией успешно, то она отсылает хосту ACK-пакет. В противном случае транзакция завершается.

В стадии передачи данных управляющие пересылки содержат одну или несколько IN- или OUT-транзакций, принцип передачи которых такой же, как и в потоковых пересылках. Все транзакции в стадии передачи данных должны производиться в одном направлении.

В статусной стадии производится последняя транзакция, которая использует те же принципы, что и в потоковых пересылках. Направление этой транзакции противоположно тому, которое использовалось в стадии передачи данных. Статусная стадия служит для сообщения о результате выполнения SETUP-стадии и стадии передачи данных. Статусная информация всегда передается от функции к хосту. При управляющей записи (Control Write Transfer) статусная информация передается в фазе передачи данных статусной стадии транзакции. При управляющем чтении (Control Read Transfer) статусная информация возвращается в фазе согласовании статусной стадии транзакции, после того как хост отправит пакет данных нулевой длины в предыдущей фазе передачи данных.

Пересылки с прерыванием могут содержать IN- или OUT-пересылки. При получении IN-пакета функция может вернуть пакет с данными, NAK-пакет или STALL-пакет. Если у функции нет информации, для которой требуется прерывание, то в фазе передачи данных функция возвращает NAK-пакет. Если работа КТ с прерыванием приостановлена, то функция возвращает STALL-пакет. При необходимости прерывания функция возвращает необходимую информацию в фазе передачи данных. Если хост успешно получил данные, то он посылает ACK-пакет. В противном случае согласующий пакет хостом не посылается.

Изохронные транзакции содержат фазу передачи признака и фазу передачи данных, но не имеют фазы согласования. Хост отсылает IN- или OUT-признак, после чего в фазе передачи данных КТ (для IN-признака) или хост (для OUT-признака) пересылает данные. Изохронные транзакции не поддерживают фазу согласования и повторные посылки данных в случае возникновения ошибок.

В связи с тем, что в интерфейсе USB реализован сложный протокол обмена информацией, в устройстве сопряжения с интерфейсом USB необходим микропроцессорный блок, обеспечивающий поддержку протокола. Поэтому основным вариантом при разработке устройства сопряжения является применение микроконтроллера, который будет обеспечивать поддержку протокола обмена. В настоящее время все основные производители микроконтроллеров выпускают продукцию, имеющую в своем составе блок USB.

7.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каковы различия между синхронным и асинхронным способами передачи информации?
2. Что такое делитель частоты?
3. В каких единицах может измеряться скорость приема-передачи данных?
4. Перечислите способы обработки сигналов, получаемых через COM-порт.
5. Какие причины возникновения прерываний могут быть отслежены через регистр идентификации прерываний?
6. Укажите, какие регистры COM-порта необходимо модифицировать в программе, где предусмотрена собственная обработка прерываний по линии IRQ4?
7. В каких случаях мышь посылает информационный пакет байтов?
8. Поясните структуру пакета байтов для мыши.
9. Какие различия в обработке перемещений мыши для двух- и трехкнопочной мыши Вам известны?
10. Какие отличия для режимов работы LPT-порта Вы знаете?

7.5 УПРАЖНЕНИЯ

Задание. Мышь начинает бегать по экрану только после щелчка по левой кнопке. При щелчке по правой мышь прекращает движение. Выгрузка программы - при одновременном нажатии на все кнопки.

```
code segment   'code'    ; определение  кодового сегмента
     assume          cs:code,ds:code,es:code
org  100h            ;резервируем 256 байт для PSP
Begin:    jmp Start ;прыгаем через данные
BASE equ  3F8h       ;базовый адрес порта COM1
Old0Co dw ?          ;смещение старого обработчика 0Ch
Old0Cs dw ?     ;сегмент старого обработчика 0Ch
Save0 dw ?      ;старое значение регистра данных
Save1 db ?      ;старое значение регистра управления прерываниями
Save3 db ?      ;старое значение регистра управления линией
Save4 db ?      ;старое значение регистра управления модемом
Maska db ?      ;старое значение маски контролера прерываний
X dw 40         ;текущая координата X
Y dw 12         ;текущая координата Y
Fix db 1        ;флаг фиксации курсора
Fbyte db ?      ;первый байт пакета данных
OfsX db ?       ;смещение по X
```

```
OfsY db ?       ;смещение по Y
Count db ?      ;счетчик принятых байт в пакете
Buttons db 0    ;количество нажатых клавиш
;Стирает или восстанавливает курсор
XorPut    proc
     push es
     pusha
; Вычисляем адрес байта видеопамяти в котором находится атрибут
; символа с координатами (X,Y) по формуле:
; адрес=0B800h:((Y*80)+X)*2+1
     mov  ax,0b800h ;в ax:=адрес начала видеопамяти
     mov  es,ax     ;es:=ax
     mov  ax,Y      ;ax:=Y
     mov  bx,160    ;bx:=80*2 (ширина экрана*2 байта)
     mul  bx        ;ax:=Y*160
     mov  bx,X      ;bx:=X
     shl  bx,1      ;bx:= bx *2
     add  ax,bx     ;ax:=ax + bx
     inc  ax        ;ax увеличим на 1
     mov  bx,ax     ;bx:=ax
     mov  dl,byte ptr es:[bx] ;читаем атрибут по вычисл. адресу
     xor  dl,52     ;накладываем на него XOR-маску
     mov  byte ptr es:[bx],dl ;измененный атрибут в видеопамять
     popa
     pop  es
     ret
XorPut         endp
; Обработчик 0Ch
Int_0Ch   proc far
     sti            ;запрещаем аппаратные прерывания
     push ds
     pushf
     pusha
     push cs
     pop  ds        ;ds:=cs
     mov  dx,BASE;в dx адрес базового порта для доступа к данным
     in   al,dx     ;читаем очередной байт из регистра данных
     test al,1000000b;если 6-ой бит равен 1,то это 1 байт пакета
     jnz  FirstByte ;это первый байт !
     cmp  Count,1   ;первый байт принят ?
     je   N2        ;да, принимаем второй.
     jmp  N3        ;иначе третий.
FirstByte:          ;обработка первого принятого байта
     mov  Count,1   ;установим счетчик принятых байт
```

```
     mov  FByte,al  ;сохраним первый принятый байт пакета
     mov  ah,al     ;ah:=al
     and  al,00000011b ;устанавливаем старшие биты в OfsX
     shl  al,6
     mov  OfsX,al   ;OfsX:=al
     and  ah,00001100b;устанавливаем старшие биты в OfsY
     shl  ah,4
     mov  OfsY,ah   ;OfsY:=ah
     jmp  ExInt     ;будем принимать остальные байты пакета
N2:                 ;обработка второго принятого байта
     mov  ah,OfsX   ;ah:=OfsX
     or   ah,al     ;устанавливаем младшие биты OfsX
     mov  OfsX,ah   ;OfsX:=ah
     mov  Count,2   ;изменяем счетчик принятых байт
     jmp  ExInt     ;будем принимать остальные байты пакета
N3:                 ;обработка третьего принятого байта
     mov  ah,OfsY   ;ah:=Ofs
     or   ah,al     ;устанавливаем младшие биты OfsY
     mov  OfsY,ah   ;OfsY:=ah
;Наконец-то прочитали весь пакет, теперь его обработаем
     call XorPut    ;уберем курсор
     cmp  fix,1     ;курсор зафиксирован ?
     je   but       ;да ! Переходим на анализ нажатых кнопок
     mov  al,OfsX   ;al:=OfsX
     cbw            ;расширяем al до слова
     add  X,ax      ;X=X+OfsX
     mov  al,OfsY   ;al:=OfsY
     cbw            ;расширяем al до слова
     add  Y,ax      ;Y=Y+OfsY
; Проверяем X и Y на предельные значения
     cmp  X,79      ;X ушел за правую границу экрана ?
     jle  NoR       ;нет !
     mov  X,79      ;да ! X:=79 (правая граница экрана)
     jmp  AY        ;уходим на проверку Y
NoR: cmp  X,0       ;X ушел за левую границу экрана ?
     jns  AY        ;нет !
     mov  X,0       ;ушел ! X:=0
AY:  cmp  Y,24      ;Y ушел за нижнюю границу экрана ?
     jle  NoD       ;нет !
     mov  Y,24      ;да ! Y:=24
     jmp  But       ;идем на обработку кнопок
NoD: cmp  Y,0       ;Y ушел за верхнюю границу экрана ?
     jns  But       ;нет ! Идем на обработку кнопок
     mov  Y,0       ;да ! Y:=0
```

```
But:                ;начинаем обработку нажатых кнопок
     mov  al,Fbyte  ;берем первый байт пакета
     mov  Buttons,0 ;сбрасываем счетчик количества нажатых кнопок
LB:  test al,0100000b    ;левая кнопка нажата ?
     jz   RB          ;нет ! Смотрим правую.
     mov  fix,0       ;да ! Освобождаем мышь
     inc  Buttons     ;Buttons увеличиваем на 1
RB:  test al,0010000b;правая кнопка нажата ?
     jz   Analiz      ;нет ! Идем дальше
     mov  fix,1       ;если нажата правая кнопка
     inc  Buttons     ;увеличим Buttons на 1
Analiz:   cmp  Buttons,2 ;были нажаты обе кнопки ?
     jne  NoPress     ;нет !
     call XorPut      ;стираем указатель
     call UnInstall ;выгружаемся
NoPress:
     call XorPut      ;стираем указатель
ExInt: mov al,20h     ;завершение аппаратного прерывания
     out  20h,al
     popa
     popf
     pop  ds
     iret
Int_0Ch   endp
; Процедура восстановления старых значений портов и старого
; обработчика
UnInstall PROC
     pusha
     cli              ;запрещаем аппаратные прерывания
     mov  dx,BASE+4 ;в dx адрес порта для доступа к р-ру
; управления модемом
     mov  al,Save4    ;в al =сохраненное значение регистра
     out  dx,al       ;восстанавливаем исходное значение регистра
     mov  al,Maska    ;в al - сохраненное значение маски
     out  21h,al      ;восстанавливаем исходное значение маски
     mov  dx,BASE+1 ;в dx - адрес порта для доступа к регистру
; управления прерываниями
     mov  al,Save1    ;в al - сохраненное значение регистра
     out  dx,al       ;восстанавливаем исходное значение регистра
     mov  dx,BASE+3 ;в dx адрес порта для доступа к регистру
;управления линией
     mov  al,Save3    ;в al - сохраненное значение регистра
     or   al,80h      ;установим DLAB = 1 для доступа
     out  dx,al       ;к делителю частоты
```

```
     mov  dx,BASE    ;в dx - адрес базового порта
     mov  ax,Save0   ;в ax - сохраненный делитель частоты
     out  dx,ax      ;восстанавливаем значение делителя частоты
     mov  dx,BASE+3 ;в dx адрес порта для доступа к регистру
; управления линией
     mov  al,Save3   ;в al - сохраненное значение регистра
     out  dx,al      ;восстанавливаем исходное значение регистра
     push ds
     mov  ax,cs:Old0Cs;в ax - сегментный адрес старой ПОП
     mov  ds,ax      ;ds:= ax
     mov  dx,cs:Old0Co   ;в dx - смещение старого обработчика
     mov  ax,250Ch   ;установим старый обработчик прерывания 0Ch
     int  21h        ;восстанавливаем старый обработчик
     pop  ds
     sti             ;разрешаем аппаратные прерывания
     popa
     ret
UnInstall ENDP
Start:    cli        ;запрещаем аппаратные прерывания
     mov  ax,350Ch   ;ah=35h для выдачи адреса ПОП 0Ch
     int  21h        ;получаем адрес вектора прерывания 0Ch
     mov  Old0Cs,es ;сохраняем сегм. адрес старого обработчика
     mov  Old0Co,bx ;сохраняем смещение старого обработчика
     lea  dx,Int_0Ch;в dx адрес нового обработчика
     mov  ax,250Ch   ;ah=25h для установки адреса обработчика
; прерывания 0Ch
     int  21h        ;установим свой обработчик
; Начинаем инициализацию мышки
     mov  dx,BASE+3 ;в dx адрес порта для доступа к регистру
; управления линией
     in   al,dx      ;читаем содержимое регистра
     mov  Save3,al   ;сохраняем его
     or   al,80h     ;установим DLAB = 1 для доступа
     out  dx,al      ;к делителю частоты
     mov  dx,BASE    ;в dx - адрес базового порта
     in   ax,dx      ;читаем регистр младшего байта делителя
     mov  Save0,ax   ;частоты и сохраняем его
     mov  ax,60h     ;делитель частоты=115200/1200=60h - 1200 бод
     out  dx,ax      ;устанавливаем делитель частоты
     mov  dx,BASE+3 ;в dx адрес порта для доступа к регистру
; управления линией
     mov  al,00000010b   ;установим структуру сигнала
     out  dx,al      ;7 информационных бит,1 стоп-бит
     mov  dx,BASE+1 ;в dx адрес порта для доступа к регистру
```

```
; управления  прерываниями
     in   al,dx      ;читаем содержимое регистра
     mov  Save1,al   ;сохраняем его
     mov  al,00000001b ;разрешаем генерацию прерывания по
     out  dx,al      ;готовности принимаемых данных
     in   al,21h     ;читаем маску контролера прерываний
     mov  Maska,al   ;сохраняем маску
     and  al,11101111b   ;размаскируем линию IRQ4
     out  21h,al     ;меняем маску
     mov  dx,BASE+4 ;в dx адрес порта для доступа к регистру
; управления модемом
     in   al,dx      ;читаем содержимое регистра
     mov  Save4,al   ;сохраняем его
     mov  al,00000001b   ;установим сигнал на линии DTR
     out  dx,al      ;заносим в регистр
     mov  al,00000011b   ;установим сигнал на линиях DTR и RTS
     out  dx,al          ; заносим в регистр
     mov  al,00001011b ;установим сигнал на линиях DTR, RTS и OUT2
     out  dx,al      ;заносим в регистр
     sti             ;разрешаем аппаратные прерывания
     int  27h        ;остаемся резидентными
code ends
end  begin
```

Варианты заданий:

1. Реализовать смену цвета мышиного курсора при нажатии на левую кнопку.
2. Реализовать циклическое перемещение мышиного курсора по экрану.
3. При нажатии на левую кнопку мышь "замирает на месте", а при нажатии на правую мыши возвращается свобода перемещения.
4. При движении мыши за ней остается "след", то есть значение атрибута в старом знакоместе не восстанавливается.
5. Реализовать движение мыши только в верхней половине экрана.
6. Реализовать движение мыши только в верхней строке экрана.
7. Реализовать перемещение мыши только по краям экрана.
8. Щелчок по правой кнопке мыши переводит ее в особый режим перемещения - только по горизонтали.
9. Мышь может перемещаться только при нажатой правой кнопке.
10. Двойной щелчок по левой кнопке меняет цвет курсора мыши.
11. По двойному щелчку по левой кнопке мышь скачет по углам экрана. Никакие другие перемещения не разрешены.

12. По двойному щелчку по правой кнопке мышь начинает двигаться только по вертикали.
13. Реализовать смену цвета курсора по двойному щелчку по левой кнопке.
14. Реализовать перемещение курсора мыши в графическом режиме.
15. При нажатии левой кнопки мыши координаты x и y мыши фиксируются, после чего щелчок по правой кнопке всегда выводит курсор мыши в точке с запомненными координатами.

7.6 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Имеется несколько утверждений о последовательной передаче данных:
   1) При последовательной передаче данные передаются по единственной линии;
   2) Асинхронная передача байта означает передачу в соответствии с определенным протоколом передачи, позволяющем синхронизировать передающее и принимающее устройства;
   3) Синхронная и асинхронная передачи представляют собой разновидность одного протокола передачи данных;
   4) Скорость передачи обычно измеряется в бодах, представляющих собой количество символов, передаваемых в одну секунду.

   Из этих утверждений правильными являются: а) 2,3,4; б) 3,4; в) 1,3; г) 1,2.
2. Делитель частоты представляет собой:

   а) число, модуль которого пропорционален скорости передачи;

   б) число, на которое нужно разделить максимально возможную скорость передачи, чтобы получить в результате скорость передачи для данного устройства;

   в) число, которое делится на минимально возможную скорость передачи, результат - скорость передачи для данного устройства.
3. Информационный пакет байтов, передаваемый мышью, включает в себя:

   а) информацию о перемещениях мыши и нажатии кнопок;

   б) только информацию о нажатии кнопок, данные о перемещении извлекаются из регистра состояния линии COM-порта;

   в) только данные о перемещении мыши, причем передаются относительные перемещения по x и по y в дополнительном коде;

   г) список номеров регистров COM-порта, содержащих данные о перемещении мыши.
4. При параллельной передаче данных с использованием стандартного параллельного порта стробирующий сигнал необходим для того, чтобы:

а) приостановить вывод байтов в случае возникновения ошибок при выводе;

б) различать между собой команды для принтера и данные, выводимые в порт данных;

в) организовать побайтный вывод во внутренний буфер принтера и регулировать поток байтов;

г) выделить в потоке выводимых байтов те из них, которые служат для организации перевода строки.

Ответы : 1 - г; 2 - б; 3 - а; 4 - в.

ЛИТЕРАТУРА

1. Агуров П.В. Интерфейсы USB. Практика использования и программирования. – СПб.: БХВ-Петербург, 2004. – 576 с.
2. Гук М.. Аппаратные средства IBM PC. Энциклопедия. 3-е издание– СПб: Питер, 2008. – 1072 с., ил.
3. Гук М. Шины PCI, USB и FireWire. Энциклопедия. – СПб: Питер, 2005. – 540 с.
4. Ирвин К. Язык Ассемблера для процессоров Intel. – М.: Вильямс, 2002. – 616 с.
5. Несвижский В. Программирование аппаратных средств в Windows. – СПб.: БХВ-Петербург, 2008. – 528 с.
6. Пирогов В. Ассемблер для Windows. - СПб.: БХВ-Петербург, 2007. – 896 с.
7. Рудаков П.И., Финогенов К.Г. Язык Ассемблера: уроки программирования – М: ДИАЛОГ-МИФИ, 2001. – 640 с., ил.
8. Таненбаум Э. Архитектура компьютера. - СПб.: Питер, 2010. -844 с.
9. Таненбаум Э. Современные операционные системы. – СПб.: Питер, 2010. -1120 с.
10. Хамахер К., Вранешич З., Заки С. Организация ЭВМ. – СПб: Питер, Киев. – 2003.-848 с.
11. Юров В., Хорошенко С. Ассемблер. Учебный курс. – СПб: ПитерКом, 1999. – 672 с, ил.

ПРИЛОЖЕНИЕ А – ПРЕДУПРЕЖДАЮЩИЕ СООБЩЕНИЯ И СООБЩЕНИЯ ОБ ОШИБКАХ

В данном приложении описаны основные сообщения, выдаваемые транслятором TASM. Эта информация позволит вам меньше ломать голову над правильностью перевода текста сообщений с английского языка и поможет понять их смысл в контексте вашей программы.

Сообщения об ошибках

32-bit segment not allowed without .386

32-битовые флаги без директивы .386 не допускаются.

Argument needs type override

Требуется явно указать тип операнда. Требуется явно указать размер (тип) выражения, так как транслятор не может сделать этого исходя только из контекста (см. урок 5). Отметим лишь, что такого рода ошибки исправляются с помощью оператора PTR, позволяющего сообщить транслятору истинный размер операнда.

Argument to operation or instruction has illegal size

Операнд операции или команды имеет недопустимый размер.

Arithmetic overflow

Арифметическое переполнение. Потеря значащих цифр при вычислении значения выражения.

ASSUME must be segment register

В директиве ASSUME должен быть указан сегментный регистр.

Bad keyword in SEGMENT statement

Неверное ключевое слово в операторе SEGMENT. Один из параметров директивы SEGMENT: тип выравнивания, тип объединения или тип сегмента, — имеет недопустимое значение.

Can't add relative quantities

Нельзя складывать относительные адреса.

Can't address with currently ASSUMEd segment registers

Невозможна адресация из текущих, установленных директивой ASSUME, сегментных регистров. В выражении содержится ссылка на переменную, для доступа к которой не специфицирован сегментный регистр.

Can't convert to pointer

Невозможно преобразование в указатель.

Can't emulate 8087 instruction

Невозможна эмуляция команд сопроцессора 8087.

Can't make variable public

Переменная не может быть объявлена как PUBLIC. Скорее всего, это вызвано тем, что данная переменная была уже где-то ранее объявлена таким образом, что уже не может быть определена как общая (PUBLIC).

Can't override ES segment

Нельзя переопределить сегмент es. Это сообщение характерно для операций типа цепочечных. В некоторых из них нельзя переопределять местоположение сегментной части адреса операнда.

Can't subtract dissimilar relative quantities

Недопустимое вычитание относительных адресов. Выражение содержит операцию вычитания двух адресов, которая для данных адресов является недопустимой. К примеру, это может случиться, если адреса находятся в разных сегментах.

Can't use macro name in expression

Недопустимо использование имени макрокоманды в качестве операнда выражения.

Can't use this outside macro

Использование данного оператора недопустимо вне макроопределения.

Code or data emission to undeclared segment

Не объявлен сегмент для кода или данных. Это может случиться, если предложение программы, генерирующее код или данные, не принадлежит ни одному из сегментов, объявленных директивами SEGMENT.

Constant assumed to mean Immediate const

Константа интерпретируется как непосредственная.

Constant too large

Слишком большая константа. Константа превышает максимально допустимую для данного режима величину. Например, числа, большие Offffh, можно использовать, если только директивой . 386/. 386P или .486/.486P разрешены команды процессора i386 или i486 соответственно.

CS not correctly assumed

Некорректное значение в регистре cs.

CS override in protected mode

Переопределение регистра cs в защищенном режиме. Это предупреждающее сообщение выдается, если в командной строке указан параметр /P.

CS unreachable from current segment

cs недостижим из текущего сегмента. При определении метки кода с помощью двоеточия (:) или с помощью директив LABEL или PROC сегментный регистр не указывает на текущий кодовый сегмент или группу, содержащую текущий кодовый сегмент.

Declaration needs name

В директиве объявления не указано имя.

Directive ignored in Turbo Pascal model

В режиме TPASCAL директива игнорируется.

Directive not allowed inside structure definition

Недопустимая директива внутри определения структуры.

Duplicate dummy arguments

Недопустимо использование одинаковых имен для формальных параметров

Expecting METHOD keyword

Требуется ключевое слово METHOD.

Expecting offset quantity

Требуется указать величину смещения.

Expecting offset or pointer quantity

Требуется указать смещение или указатель.

Expecting pointer type

Операнд должен быть указателем. Означает, что операндом текущей команды должен быть адрес памяти.

Expecting record field name

Требуется имя поля записи. Инструкция SETFIELD или GETFIELD использована без последующего имени поля.

Expecting register ID

Требуется идентификатор регистра.

Expecting scalar type

Операнд должен быть константой.

Expecting segment or group quantity

Должно быть указано имя сегмента или группы.

Extra characters on line

Лишние символы в строке.

Forward reference needs override

Ошибка при использовании умолчания для ссылки вперед.

Global type doesn't match symbol type

Тип, указанный в директиве GLOBAL, не совпадает с действительным типом имени идентификатора.

ID not member of structure

Идентификатор не является полем структуры.

Illegal forward reference

Недопустимая ссылка вперед.

Illegal immediate

Недопустим непосредственный операнд.

Illegal indexing mode

Недопустимый режим индексации.

Illegal instruction

Недопустимая команда.

Illegal instruction for currently selected processor(s)

Недопустимая команда для выбранного в настоящий момент процессора.

Illegal local argument

Недопустимый локальный параметр.

Illegal local symbol prefix

Недопустимый префикс для локальных имен идентификаторов.

Illegal macro argument

Недопустимый параметр макрокоманды.

Illegal memory reference

Недопустима ссылка на память.

Illegal number

Недопустимое число.

Illegal origin address

Недопустимый начальный адрес.

Illegal override in structure

Недопустимое переопределение в структуре.

Illegal override register

Недопустимое переопределение регистра.

Illegal radix

Недопустимое основание системы счисления. В директиве , RADIX в качестве основания системы счисления указано недопустимое число. Основанием системы счисления могут быть только числа 2,8, 10 и 16. Эти числа интерпретируются как десятичные независимо от текущей системы счисления.

Illegal register for instruction

Недопустимый регистр в инструкции. В качестве операнда инструкции SETFIELD и GETFIELD использован недопустимый регистр.

Illegal register multiplier

Недопустимо указание множителя для регистра.

Illegal segment address

Недопустимый сегментный адрес.

Illegal use of constant

Недопустимо использование константы.

Illegal use of register

Недопустимо использование регистра.

Illegal use of segment register

Недопустимо использование сегментного регистра.

Illegal USES register

В директиве USES указан недопустимый регистр.

Illegal version ID

Недопустимый идентификатор версии.

Illegal warning ID

Недопустимый идентификатор предупреждающего сообщения.

Instruction can be compacted with override

Возможно сокращение длины команды, если явно указать тип имени. Из-за наличия ссылки вперед на имя идентификатора объектный код содержит дополнительные команды NOP. Этим самым транслятор резервирует место для размещения адреса идентификатора. При необходимости код можно сократить, убрав ссылку вперед либо явно указав тип символического имени.

Invalid model type

Недопустимая модель памяти.

Invalid operand(s) to instruction

Недопустимый операнд (операнды) для данной команды.

Labels can't start with numeric characters

Метки не могут начинаться с цифровых символов.

Line too long — truncated

Строка слишком длинная, и поэтому производится усечение.

Location counter overflow

Переполнение счетчика адреса.

Method call requires object name

В вызове метода необходимо имя объекта.

Missing argument list

Отсутствует список аргументов.

Missing argument or <

Отсутствует аргумент либо не указана угловая скобка <.

Missing argument size variable

Отсутствует переменная для размера блока параметров.

Missing COMM ID

Отсутствует идентификатор в директиве COMM.

Missing dummy argument

Отсутствует формальный параметр.

Missing end quote

Отсутствует закрывающая кавычка.

Missing macro ID

Отсутствует идентификатор макрокоманды.

Missing module name

Отсутствует имя модуля.

Missing or illegal type specifier

Отсутствует или неверно указан спецификатор типа.

Missing table member ID

Пропущен идентификатор элемента таблицы.

Missing term in list

Отсутствует член в списке параметров.

Missing text macro

Отсутствует текстовая макрокоманда.

Model must be specified first

Сначала должна быть указана модель памяти.

Module is pass-dependant — compatibility pass was done

Модуль зависит от прохода. Выполнен проход, обеспечивающий совместимость с MASM.

Name must come first
Имя должно быть указано первым.
Near jump or call to different CS
Адресат ближнего перехода или вызова находится в другом кодовом сегменте.
Need address or register
Требуется указать адрес или регистр.
Need colon
Требуется двоеточие.
Need expression
Требуется указать выражение.
Need file name after INCLUDE
В директиве INCLUDE должно быть указано имя файла.
Need left parenthesis
Отсутствует левая круглая скобка.
Need method name
Требуется имя метода.
Need pointer expression
Требуется выражение-указатель.
Need quoted string
Требуется указать строку в кавычках.
Need register in expression
В выражении требуется указать имя регистра.
Need right angle bracket
Отсутствует правая угловая скобка.
Need right curly bracket
Требуется правая фигурная скобка.
Need right parenthesis
Отсутствует правая круглая скобка.
Need right square bracket
Отсутствует правая квадратная скобка.
Need stack argument
Не указан стековый параметр в команде арифметики с плавающей запятой.
Need structure member name
Не указано имя поля структуры.
Not expecting group or segment quantity
Использование имени группы или сегмента недопустимо.
One non-null field allowed per union expansion
При расширении объединения допускается указывать только одно поле непустым.
Only one startup sequence allowed
Допускается только одна директива генерации кода инициализации.
Open conditional
Открытый условный блок. После завершающей программу директивы END обнаружен незакрытый, условно ассемблируемый блок, открытый одной из директив IFxxx.
Open procedure
Открытая процедура. После завершающей программу директивы END обнаружен незакрытый директивой ENDP блок описания процедуры, открытый где-то в программе директивой PROC.
Open segment
Открытый сегмент. После завершающей программу директивы END обнаружен незакрытый директивой ENDS сегмент, открытый где-то в программе директивой SEGMENT.
Open structure definition

Не указан конец определения структуры (директива ENDS).

Operand types do not match

Не совпадают типы операндов. Тип одного из операндов команды не совпадает с типом другого операнда либо не является типом, допустимым для данной команды.

Operation illegal for static table member

Для статического элемента таблицы операция не допускается.

Pass-dependant construction encountered

Обнаружена конструкция, зависящая от прохода. Данную ошибку можно исправить, убрав ссылки вперед либо указав нужное число проходов транслятора в опции командной строки */m.*

Pointer expression needs brackets

Адресное выражение должно быть заключено в квадратные скобки.

Positive count expecting

Счетчик должен быть положительным.

Record field too large

Слишком длинное поле в записи.

Record member not found

Не найден статический элемент записи.

Recursive definition not allowed for EQU

Рекурсивное определение недопустимо в директиве EQU.

Register must be AL or AX

Допустимо указание только регистра a1 или ax.

Register must be DX

Допустимо указание только регистра dx.

Relative jump out of range by __ bytes

Адрес назначения условного перехода превышает допустимый предел на __ байт.

Relative quantity illegal

Недопустимый относительный адрес. Ссылка на адрес памяти не может быть разрешена на этапе ассемблирования.

Reserved word used as symbol

Зарезервированное слово используется в качестве имени идентификатора.

Rotate count must be constant or CL

Счетчик в командах сдвига должен быть указан с помощью константы или регистра с 1.

Rotate count out of range

Недопустимое значение счетчика сдвига.

Segment alignment not strict enough

Выравнивание сегмента недостаточно точное.

Segment attributes illegally redefined

Недопустимое переопределение атрибутов сегмента. Суть здесь в том, что пользователь может повторно открывать уже определенный ранее сегмент. Но при этом атрибуты этого сегмента должны иметь те же самые значения либо вообще быть опущены (тогда будут взяты прежние значения).

Segment name is superfluous

Имя сегмента игнорируется.

String too long

Слишком длинная строка. Длина указанной в кавычках строки превышает 255 символов.

Symbol already defined:__

Имя идентификатора уже определено.

Symbol already different kind

Имя идентификатора уже объявлено с другим типом.

Symbol has no width or mask
Имя идентификатора не может быть использовано в операциях WIDTH и MASK.
Symbol is not a segment or already part of a group
Имя идентификатора не является именем сегмента, или оно уже определено в группе.
Text macro expansion exceeds maximum line length
Расширение текстовой макрокоманды превышает максимально допустимую длину.
Too few operands to instruction
В команде не хватает операндов.
Too many errors or warnings
Слишком много ошибок или предупреждений. Число сообщений об ошибках превысило максимально возможное число 100.
Too many initial values
Слишком много начальных значений.
Too many register multipliers in expression
В выражении содержится слишком много множителей для регистров.
Too many registers in expression
В выражении указано слишком много регистров.
Too many USES registers
Слишком много регистров в директиве USES.
Trailling null value assumed
Предполагается конечное пустое значение.
Undefined symbol
Идентификатор не определен.
Unexpected end of file (no END directive)
Неожиданный конец файла (нет директивы END).
Unknown character
Неизвестный символ.
Unmatched ENDP:_
Непарная директива ENDP;_.
Unmatched ENDS:_
Непарная директива ENDS:_.
User-generated error
Ошибка, сгенерированная пользователем. Сообщение выдается в результате выполнения одной из директив генерирования ошибки.
USES has no effect without language
USES игнорируется без спецификации языка.
Value out of range
Значение константы превышает допустимое значение.

Сообщения о фатальных ошибках

Кроме вышерассмотренных ошибок TASM формирует еще один тип — сообщения о *фатальных* ошибках. Их особенность в том, что при их возникновении TASM выдает соответствующее сообщение и немедленно прекращает ассемблирование исходного файла.

Bad switch
Неверный параметр-переключатель командной строки.
Can't find @file __
Не найден файл подсказок __.
Can't locate file __
Не обнаружен файл __. При выдаче этого сообщения нужно проверить, правильно ли указаны в имени файла имя диска и путь к файлу, заданному в директиве INCLUDE.

Error writing to listing file

Ошибка при записи в файл листинга. Возможно, просто исчерпано место на диске.

Error writing to object file

Ошибка при записи в объектный файл. Возможно, просто исчерпано место на диске.

File not found

Не найден файл. В командной строке указано имя несуществующего исходного файла.

File was changed or deleted while assembly in progress

Файл был изменен или уничтожен в процессе ассемблирования.

Insufficient memory to process command line

Не хватает памяти для обработки командной строки.

Internal error

Внутренняя ошибка.

Invalid command line

Недопустимая командная строка.

Invalid number after _

Недопустимый номер после _.

Out of hash space

Не хватает памяти под хеш-таблицы. Для каждого имени идентификатора в программе транслятор формирует один элемент таблицы идентификаторов. Эта таблица рассчитана на 16 384 имен идентификаторов. При необходимости это число можно увеличить, используя параметр командной строки /kh.

Out of memory

Не хватает памяти. Для ассемблирования пользовательского файла недостаточно свободной памяти.

Out of string space

Не хватает памяти под строки. Здесь имеется в виду оперативная память для хранения строк, представляющих собой имена идентификаторов, имена файлов, информацию для разрешения ссылок вперед, текстов макрокоманд. Допускается максимум 512 Кбайт памяти.

Too many errors found

Обнаружено слишком много ошибок. Трансляция прекращена, так как в исходном файле содержится слишком много ошибок.

Unexpected end of file (no END directive)

Неожиданный конец файла (отсутствует директива END).

Приложение Б – Scan-коды клавиш

Скан-код		Клавиша	Скан-код		Клавиша	Скан-код		Клавиша
01h	1	**Esc**	1Dh	29	**Ctrl**	39h	57	**Пробел**
02h	2	**1 !**	1Eh	30	**A**	3Ah	58	**Caps Lock**
03h	3	**2 @**	1Fh	31	**S**	3Bh	59	**F1**
04h	4	**3 #**	20h	32	**D**	3Ch	60	**F2**
05h	5	**4 $**	21h	33	**F**	3Dh	61	**F3**
06h	6	**5 %**	22h	34	**G**	3Eh	61	**F4**
07h	7	**6 ^**	23h	35	**H**	3Fh	63	**F5**
08h	8	**7 &**	24h	36	**J**	40h	64	**F6**
09h	9	**8 ***	25h	37	**K**	41h	65	**F7**
0Ah	10	**9 (**	26h	38	**L**	42h	66	**F8**
0Bh	11	**0)**	27h	39	**; :**	43h	67	**F9**
0Ch	12	**- _**	28h	40	**‘ «**	44h	68	**F10**
0Dh	13	**= +**	29h	41	**` ~**	45h	69	**Num Lock**
0Eh	14	**BackSpace**	2Ah	42	**левый Shift**	46h	70	**Scroll Lock**
0Fh	15	**Tab**	2Bh	43	**\ |**	47h	71	**Home [7]**
10h	16	**Q**	2Ch	44	**Z**	48h	72	**стр. вверх[8]**
11h	17	**W**	2Dh	45	**X**	49h	73	**Pg Up [9]**
12h	18	**E**	2Eh	46	**C**	4Ah	74	**-**
13h	19	**R**	2Fh	47	**V**	4Bh	75	**<- [4]**
14h	20	**T**	30h	48	**B**	4Ch	76	**[5]**
15h	21	**Y**	31h	49	**N**	4Dh	77	**-> [6]**
16h	22	**U**	32h	50	**M**	4Eh	78	**+**
17h	23	**I**	33h	51	**, <**	4Fh	79	**End [1]**
18h	24	**O**	34h	52	**. >**	50h	80	**стр. вниз [2]**
19h	25	**P**	35h	53	**/ ?**	51h	81	**PgDn [3]**
1Ah	26	**[{**	36h	54	**правый Shift**	52h	82	**Ins [0]**
1Bh	27	**] }**	37h	55	**PrtScr***	53h	83	**Del [.]**
1Ch	28	**Enter**	38h	56	**Alt**	54h	84	**SysRq**

Скан-коды Windows-клавиатуры (по умолчанию):

LeftWindows – E0 5B

RightWindows – E0 5C

Application – E0 5D

ОГЛАВЛЕНИЕ