Министерство сельского хозяйства РФ

Кубанский государственный
аграрный университет

Кафедра Системного анализа и обработки информации

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Методические указания к лабораторным работам по дисциплине
«Системное программное обеспечение»

для студентов третьего курса направления подготовки 09.03.02
«ИФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ»
всех форм обучения

Краснодар, 2019

Работу подготовили по решению методической комиссии факультета прикладной информатики и кафедры Системного анализа и обработки информации (протокол №_____ от _____ 2019 г.)
Мурлин А.Г., Иванова Е.А.,

Системное программное обеспечение.

Методические указания к лабораторным работам по дисциплине «Системное программное обеспечение» для студентов третьего курса направления подготовки 09.03.02 «ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ» всех форм обучения/ Кубан. гос. аграрн. ун-т., Сост. А.Г. Мурлин, Е.А. Иванова, 2019, 184 с.

Составлены в соответствии с рабочей программой курса “Системное программное обеспечение” для студентов третьего курса направления подготовки 09.03.02

Содержат описание лабораторных работ, методические указания к их выполнению, требования к оформлению отчета, приложение с примерами программ и пр.

Ил. 25. Библиогр.: 11 назв.

Рецензенты: проф. В.И. Лойко (КубГАУ),
проф. Л.А. Максименко (КубГТУ).

СОДЕРЖАНИЕ

Лабораторная работа №1 Инструментальные средства разработки программного обеспечения на языке Ассемблера для платформы IBM PC с ОС MS DOS

1 Цель работы*:*

Получить основные навыки использования пакетов MASM и TASM при программировании на языке Ассемблера. Исследовать работу и научиться использовать команды пересылки, загрузки и операций с флагами. Познакомиться на практике с режимами адресации.

2 Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя по вариантам;
- написать программу, ввести программу, отладить и решить
- ее на ЭВМ;
- оформить отчет.

3 Теоретические основы

4.1 Основные инструментальные средства

Пакет MASM удобное средство для создания программ на языке Ассемблера для реального и защищённого режимов процессоров семейства X86.

Основные инструментальные средства, входящие в состав пакета:

а) pwb - интегрированная среда, включающая в себя редактор текстов программ и позволяющая компилировать модули и запускать программы не выходя из неё;

б) cv - интегрированная среда предназначенная для отладки и дизассемблирования программ;

в) masm - компилятор с языка Ассемблера;

г) link - компоновщик объектных модулей в исполняемую программу;

д) ml - компилятор и компоновщик в одном лице.

Компилятор masm рекомендуется запускать с ключом /Zi, а компоновщик - с ключом /Co; при этом в объектные модули будет записана отладочная информация.

Аналогичные опции рекомендуется установить и в среде pwb.

Пакет TASM не имеет единой интегрированной среды, но по популярности не уступает пакету MASM.

Основные инструментальные средства, входящие в состав пакета:

а) tasm - компилятор с языка Ассемблера;

б) tlink - компоновщик объектных модулей в исполняемую программу;

в) td - интегрированная среда предназначенная для отладки и дизассемблирования программ.

Учитывая отсутсвие интегрированной среды в пакете TASM, рекомендуется составить пакетные файлы для компиляции, компоновки и отладки, например так (предполагается, что пакет TASM инсталлирован в каталог C:\TASM, исходный текст программы записан в файл main.asm, а соответствующие пакетные файлы находятся в том же каталоге, что и main.asm):

а) компиляция и компоновка (файл COMPILE.BAT):

```
@echo off
cls
c:\tasm\bin\tasm /zi main.asm
if errorlevel 1 goto ERROR
c:\tasm\bin\tlink /v main.obj
if errorlevel 1 goto ERROR
exit
:ERROR
echo ERROR!
```

б) отладка (файл DEBUG.BAT):

```
@echo off
cls
if not exist main.exe echo main.exe не существует!
if not exist main.exe goto EXIT
c:\tasm\bin\td main.exe
:EXIT
```

Создание первого проекта в Microsoft Macro Assembler

Запустите PWB.exe (не забудьте сначала выполнить New-vars.bat из каталога с PWB.exe)

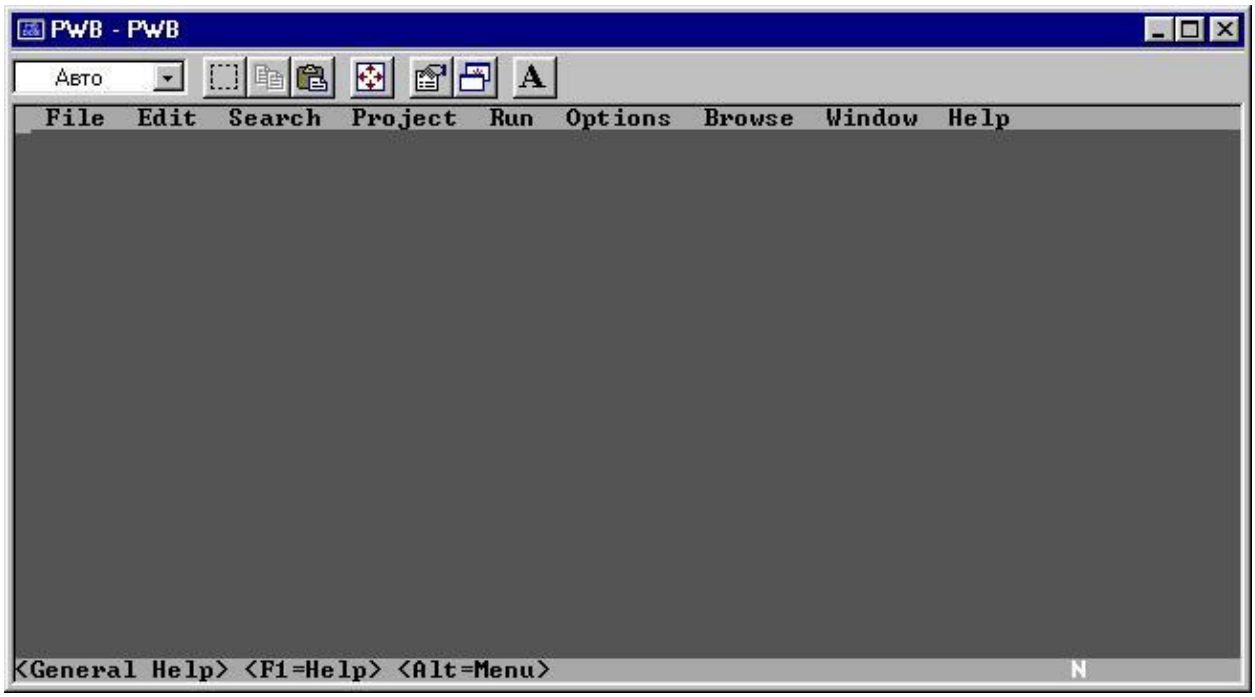

Рисунок 1 – Терминал

Выполните следующую команду меню:
Project -> New Project…, где введёте имя вашего проекта.
Например: Hello.

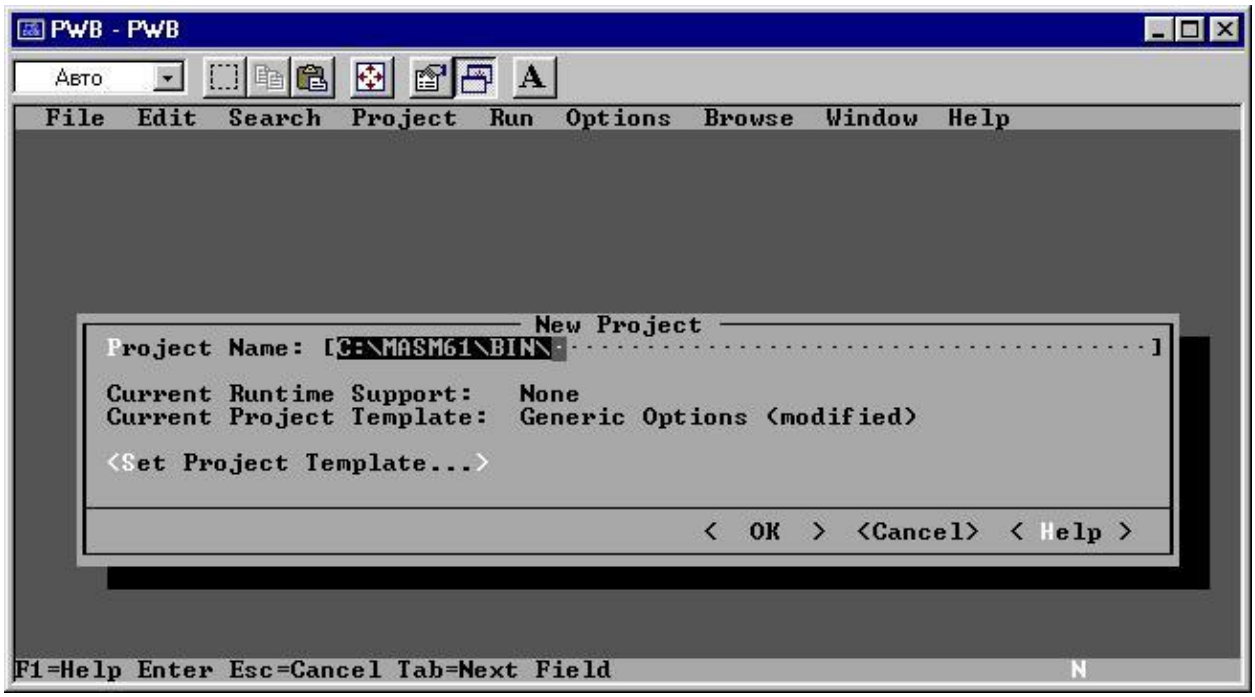

Рисунок 2 – Создание нового проекта

File -> New для создания в вашем проекте файла для исходного текста.

Введите код программы и выполните:
Project -> Compile File: Hello.asm;
Project -> Build: Hello.exe;
Run -> Execute: Hello.exe;
Run -> Debug: Hello.exe

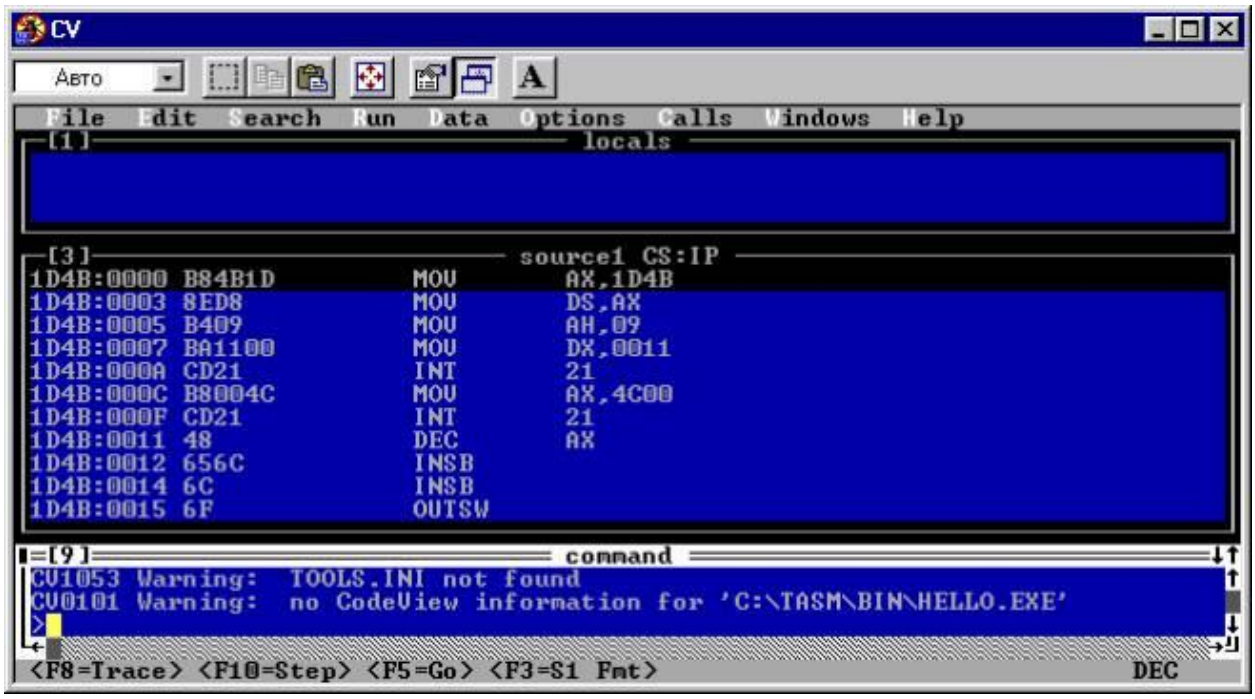

Рисунок 3 – Просмотр регистров

Для того чтобы увидеть регистры, воспользуйтесь командой меню Windows -> Register или комбинацией "горячих" клавиш Alt+7.

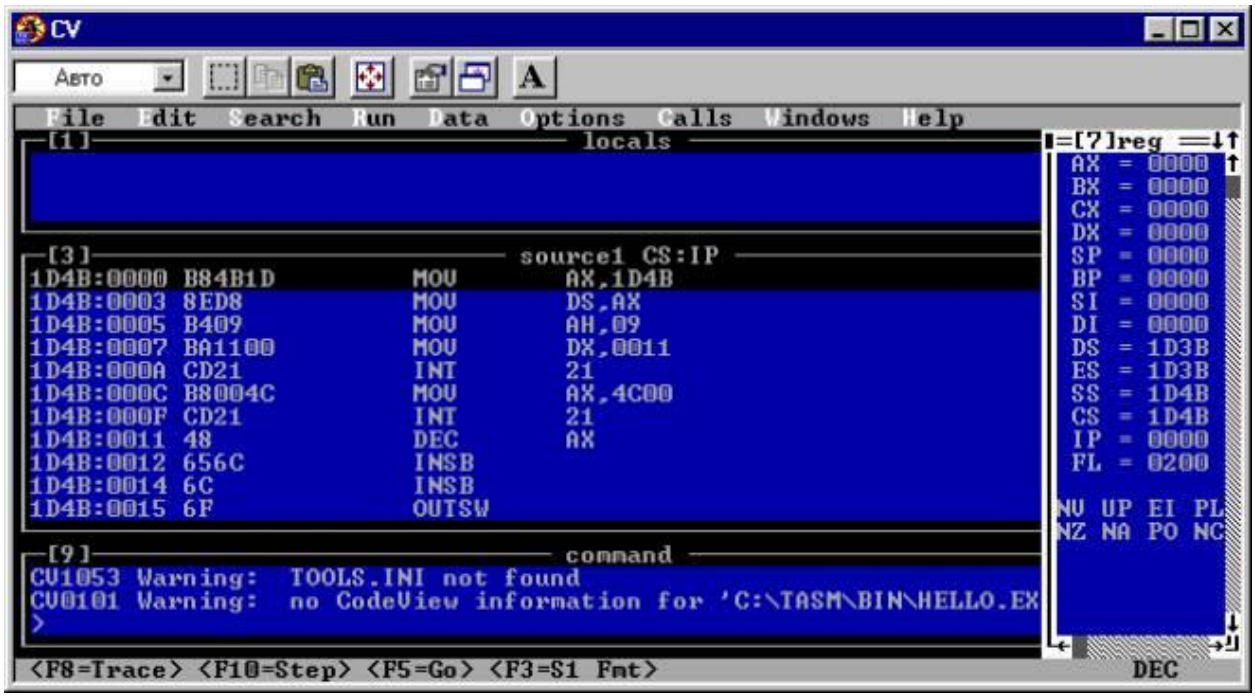

Рисунок 4 – Просмотр содержимого памяти

Для того чтобы увидеть содержимое памяти воспользуйтесь командой меню Windows -> Memory 1 или Windows -> Memory 2 или комбинацией "горячих" клавиш Alt+5 или Alt+6 соответственно.

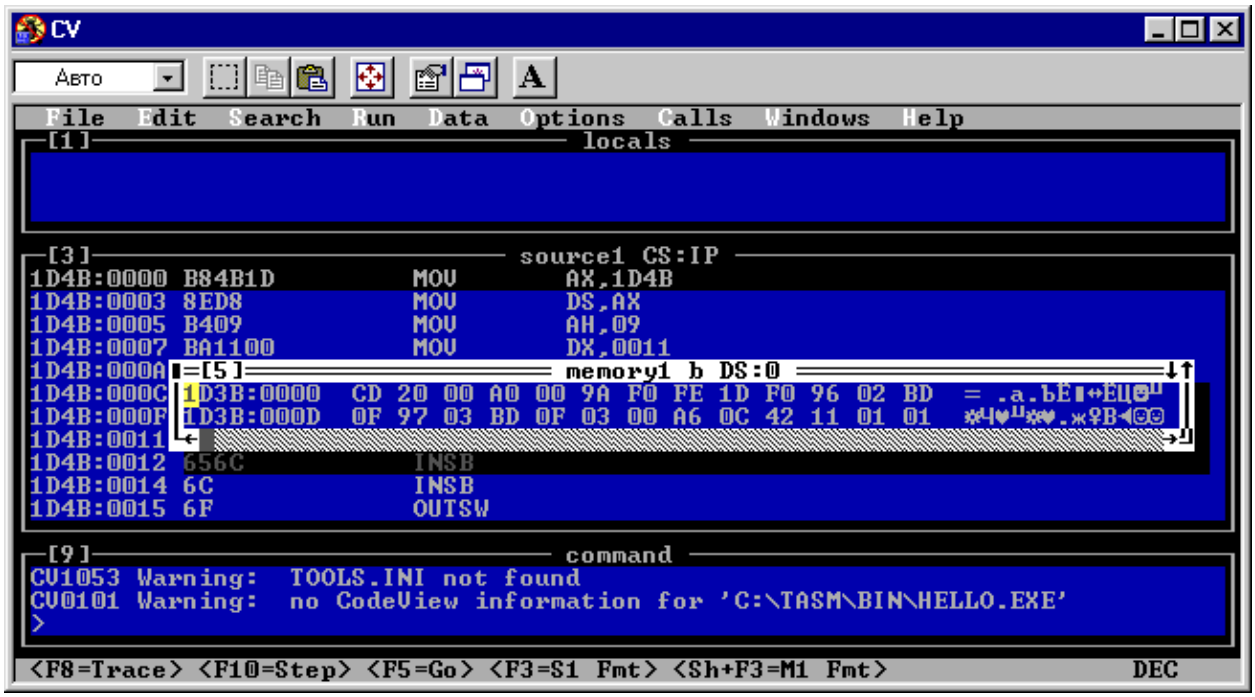

Рисунок 5 – Содержимое памяти

Далее для отладки используют трассировку (F8) - пошаговое выполнение программы, проверяя значения регистров, флагов и переменных.

При ассемблировании программы вы можете использовать в одной командной строке несколько опций, разделяя их пробелами, что не является обязательным. Ниже приводится несколько примеров:

```
tasm /h
tasm - l -c ff
tasm  /l  /c  ff
tasm -zi ff
tasm -l -iC:\INCLUDES  ff
```

Вместо ассемблирования первая команда просто выводит список опций командной строки. Чтобы распечатать его, используйте команду tasm /h > prn. Вторая команда создает файл листинга с перекрестными номерами строк(#10, #25 и т.п.) в конце. Третья команда, делая то же самое и просто показывает, что при определении опций можно использовать косую черту вместо короткой черты. Четвертая команда добавляет в FF.OBJ информацию для Turbo Debugger. Последняя команда создает файл листинга и определяет путь включаемых (дополнительно используемых) файлов. Включаемые файлы - это отдельные текстовые файлы, которые по вашему желанию Turbo Assembler вставляет в программу. Программа 2.1 не использует включаемые файлы, следовательно, эта команда не приведет к каким-либо практическим результатам. Turbo Linker также имеет различные опции командной строки, определяемые аналогичным образом, за исключением того, что некоторые ранние версии TLINK требуют использования опций с косой чертой (/m), а не дефиса *(-m)*. Некоторые версии компоновщиков допускают использование, как косой черты, так и дефиса, но при наборе нескольких опций с косой чертой в одной строке они должны разделяться пробелами. Ниже приводится несколько примеров применения опций в Turbo Linker:

```
tlink  -v   ff
tlink  /v   ff
tlink   -m -l ff tlink
/m  /l ff tlink  -x  ff
tlink  /x  ff
```

Опции -v или /v в первых двух строках подготавливают FF.EXE да использования его с Turbo Debugger. Следующие строки определяют две опции, которые приводят к созданию расширенного файла отображений (FF.MAP) и добавляют в этот файл вспомогательную информацию о номерах строк (/l). Опции /x и - x запрещают Turbo Linker создавать

файл отображений, что приводит к небольшой экономии дискового пространства и времени компоновки. Применяйте эти команды, если нет необходимости в файле отображений, который показывает организацию памяти программы и обычно используется отладчиками либо как часть программной документации.

Опции Tasm:

/a Алфавитное упорядочение сегмента
/b Не имеет никакого эффекта.
/c Перекрестная ссылка в распечатке файла.
/d Определяет символ.
/e Генерирует команды эмулятора с плавающей точкой.
/h,/? Отображает экран справки.
/i Устанавливает путь для включаемого файла.
/j Определяет директиву запуска ассемблера.
/kh Максимальное число позволенных символов
ks Максимальный размер пространства под строку Турбо Ассемблера.
/l Генерирует файл распечатки.
/la Показывает код интерфейса высокого уровня в распечатке файла.
/m # Позволить # многократным проходам решать вперед ссылки
/ml Обрабатывает все символы как чувствительные к регистру.
/mu Преобразовывает символы в верхний регистр
/mx Делает общие(public) и внешние(external) символы(глобальные) чувствительные к регистру .
/mv# Установить максимальное значение длины для символов
/n Подавляет таблицу символов в распечатке файла.
/o Вывод tlink-совместимые объекты с допускаемой оверлейной поддержкой.
/oi Вывод совместимых с IBM компоновщиком объектов IBM линейно - выполнимый компоновщик.
/op Вывод Pharlap совместимые объекты
/os Вывод tlink-совместимые объекты без оверлейных программ.
/p Проверяет на чистоту код защищенного режима.
/q Удаляет все записи из .OBJ, в которых нет необходимости для компонования.
/r Генерирует действительные с плавающей запятой команды
/s Последовательное упорядочение(заказывание) сегмента. (Значение по умолчанию)
/t Подавляет сообщения об успешной трансляции.
/v Опция совместимость.

/w Генерация предупреждающих сообщений
/x Включает неправильные условные выражения в распечатку.
x/z Отображает исходные линии наряду с сообщениями об ошибках.
/zd Допускает информацию номера строки в .OBJ
/zi Допускает информацию отладки в объектном файле.

Листинг 1

```
assume CS:CodeSg, DS:CodeSg
CodeSg segment 'CODE'

begin:    mov AX, CodeSg
  mov DS, AX
  ;
  mov AH, 09h
  mov DX, offset message
  int 21h
  ;
  mov AX, 4C00h
  int 21h
  ;
  message db 'Hello World!', '$'
  ;
CodeSg ends

end begin
```

4.2 Организация программы

4.2.1 Сегменты

Каждая программа, написанная на любом языке программирования, состоит из одного или нескольких сегментов. Обычно область памяти, в которой находятся команды, называют сегментом кода, область памяти с данными — сегментом данных и область памяти, отведенную под стек, — сегментом стека. Разумеется, ассемблер позволяет изменять устройство программы как угодно — помещать данные в сегмент кода, разносить код на множество сегментов, помещать стек в один сегмент с данными или вообще использовать один сегмент для всего.

Сегмент программы описывается директивами SEGMENT и ENDS.

```
имя_сегмента   segment readonly выравн. тип разряд 'класс'
                ...
имя_сегмента   ends
```

Имя сегмента — метка, которая будет использоваться для получения сегментного адреса, а также для комбинирования сегментов в группы.

Все пять операндов директивы SEGMENT необязательны.

READONLY. Если этот операнд присутствует, MASM выдаст сообщение об ошибке на все команды, выполняющие запись в данный сегмент. Другие ассемблеры этот операнд игнорируют.

Выравнивание. Указывает ассемблеру и компоновщику, с какого адреса может начинаться сегмент. Значения этого операнда:

BYTE — с любого адреса;

WORD — с четного адреса;

DWORD — с адреса, кратного 4;

PARA — с адреса, кратного 16 (граница параграфа);

PAGE — с адреса, кратного 256.

По умолчанию используется выравнивание по границе параграфа.

Тип. Выбирает один из возможных типов комбинирования сегментов:

- тип PUBLIC (иногда используется синоним MEMORY) означает, что все такие сегменты с одинаковым именем, но разными классами будут объединены в один;
- тип STACK — то же самое, что и PUBLIC, но должен использоваться для сегментов стека, потому что при загрузке программы сегмент, полученный объединением всех сегментов типа STACK, будет использоваться как стек;
- сегменты типа COMMON с одинаковым именем также объединяются в один, но не последовательно, а по одному и тому же адресу, следовательно, длина суммарного сегмента будет равна не сумме длин объединяемых сегментов, как в случае PUBLIC и STACK, а длине максимального. Таким способом иногда можно формировать оверлейные программы;
- тип AT — выражение указывает, что сегмент должен располагаться по фиксированному абсолютному адресу в памяти. Результат выражения, использующегося в качестве операнда для AT, равен этому адресу, деленному на 16. Например: segment at 40h — сегмент, начинающийся по абсолютному адресу 0400h. Такие сегменты обычно содержат только метки, указывающие на области памяти, которые могут потребоваться программе;

PRIVATE (значение по умолчанию) — сегмент такого типа не объединяется с другими сегментами.

Разрядность. Этот операнд может принимать значения USE16 и USE32. Размер сегмента, описанного как USE16, не может превышать 64 Кб, и все команды и адреса в этом сегменте считаются 16-битными. В этих сегментах все равно можно применять команды, использующие 32-битные регистры или ссылающиеся на данные в 32-битных сегментах, но они будут использовать префикс изменения разрядности операнда или адреса и окажутся длиннее и медленнее. Сегменты USE32 могут занимать до 4 Гб, и все команды и адреса в них по умолчанию 32-битные. Если разрядность сегмента не указана, по умолчанию используется USE16 при условии, что перед директивой .MODEL не применялась директива задания допустимого набора команд .386 или старше.

Класс сегмента — это любая метка, взятая в одинарные кавычки. Все сегменты с одинаковым классом, даже сегменты типа PRIVATE, будут расположены в исполняемом файле непосредственно друг за другом.

Для обращения к любому сегменту следует сначала загрузить его сегментный адрес (или селектор в защищенном режиме) в какой-нибудь сегментный регистр. Если в программе определено много сегментов, удобно объединить несколько сегментов в группу, адресуемую с помощью одного сегментного регистра:

имя_группы group имя_сегмента...

Операнды этой директивы — список имен сегментов (или выражений, использующих оператор SEG), которые объединяются в группу. Имя группы теперь можно применять вместо имен сегментов для получения сегментного адреса и для директивы ASSUME.

assume регистр:связь...

Директива ASSUME указывает ассемблеру, с каким сегментом или группой сегментов связан тот или иной сегментный регистр. В качестве операнда «связь» могут использоваться имена сегментов, имена групп, выражения с оператором SEG или слово «NOTHING», означающее отмену действия предыдущей ASSUME для данного регистра. Эта директива не изменяет значений сегментных регистров, а только позволяет ассемблеру проверять допустимость ссылок и самостоятельно вставлять при необходимости префиксы переопределения сегментов, если они необходимы.

Перечисленные директивы удобны для создания больших программ на ассемблере, состоящих из разнообразных модулей и содержащих множество сегментов. В повседневном программировании обычно ис-

пользуется ограниченный набор простых вариантов организации программы, известных как модели памяти.

Листинг 2

```
CodeSg segment 'CODE'
assume CS:CodeSg, DS:DataSg, SS:StackSg
begin:    mov AX, DataSg
  mov DS, AX
  ;
  ; добавить код сюда
  ;
  mov AX, 4C00h
  int 21h
CodeSg ends
;-----------------------------------
DataSg segment 'DATA'
  ;
  ; описать данные здесь
  ;
DataSg ends
;-----------------------------------
StackSg segment stack 'STACK'
  db 256 dup(0)
StackSg ends
;-----------------------------------
end begin
```

4.3 Модели памяти и упрощенные директивы определения сегментов

Модели памяти задаются директивой .MODEL

.model модель, язык, модификатор

где модель — одно из следующих слов:

TINY — код, данные и стек размещаются в одном и том же сегменте размером до 64 Кб. Эта модель памяти чаще всего используется при написании на ассемблере небольших программ;

SMALL — код размещается в одном сегменте, а данные и стек — в другом (для их описания могут применяться разные сегменты, но объединенные в одну группу). Эту модель памяти также удобно использовать для создания программ на ассемблере;

COMPACT — код размещается в одном сегменте, а для хранения данных могут использоваться несколько сегментов, так что для обраще-

ния к данным требуется указывать сегмент и смещение (данные дальнего типа);

MEDIUM — код размещается в нескольких сегментах, а все данные — в одном, поэтому для доступа к данным используется только смещение, а вызовы подпрограмм применяют команды дальнего вызова процедуры;

LARGE и HUGE — и код, и данные могут занимать несколько сегментов;

FLAT — то же, что и TINY, но используются 32-битные сегменты, так что максимальный размер сегмента, содержащего и данные, и код, и стек, — 4 Мб.

Язык — необязательный операнд, принимающий значения C, PASCAL, BASIC, FORTRAN, SYSCALL и STDCALL. Если он указан, подразумевается, что процедуры рассчитаны на вызов из программ на соответствующем языке высокого уровня, следовательно, если указан язык C, все имена ассемблерных процедур, объявленных как PUBLIC, будут изменены так, чтобы начинаться с символа подчеркивания, как это принято в C.

Модификатор — необязательный операнд, принимающий значения NEARSTACK (по умолчанию) или FARSTACK. Во втором случае сегмент стека не будет объединяться в одну группу с сегментами данных.

После того как модель памяти установлена, вступают в силу упрощенные директивы определения сегментов, объединяющие действия директив SEGMENT и ASSUME. Кроме того, сегменты, объявленные упрощенными директивами, не требуется закрывать директивой ENDS — они закрываются автоматически, как только ассемблер обнаруживает новую директиву определения сегмента или конец программы.

Директива .CODE описывает основной сегмент кода

```
.code имя_сегмента
```

эквивалентно

```
_TEXT    segment word public 'CODE'
```

для моделей TINY, SMALL и COMPACT и

```
name_TEXT    segment word public 'CODE'
```

для моделей MEDIUM, HUGE и LARGE (name — имя модуля, в котором описан данный сегмент). В этих моделях директива .CODE также допускает необязательный операнд — имя определяемого сегмента, но все сегменты кода, описанные так в одном и том же модуле, объединяются в один сегмент с именем NAME_TEXT.

```
.stack размер
```

Директива .STACK описывает сегмент стека и эквивалентна директиве

STACK segment para public ’stack’

Необязательный параметр указывает размер стека. По умолчанию он равен 1 Кб.

.data

Описывает обычный сегмент данных и соответствует директиве

_DATA segment word public ’DATA’

.data?

Описывает сегмент неинициализированных данных:

_BSS segment word public ’BSS’

Этот сегмент обычно не включается в программу, а располагается за концом памяти, так что все описанные в нем переменные на момент загрузки программы имеют неопределенные значения.

.const

Описывает сегмент неизменяемых данных:

CONST segment word public ’CONST’

В некоторых операционных системах этот сегмент будет загружен так, что попытка записи в него может привести к ошибке.

.fardata имя_сегмента

Сегмент дальних данных:

имя_сегмента segment para private ’FAR_DATA’

Доступ к данным, описанным в этом сегменте, потребует загрузки сегментного регистра. Если не указан операнд, в качестве имени сегмента используется FAR_DATA.

.fardata? имя_сегмента

Сегмент дальних неинициализированных данных:

имя_сегмента segment para private ’FAR_BSS’

Как и в случае с FARDATA, доступ к данным из этого сегмента потребует загрузки сегментного регистра. Если имя сегмента не указано, используется FAR_BSS.

Во всех моделях памяти сегменты, представленные директивами .DATA, .DATA?, .CONST, .FARDATA и .FARDATA?, а также сегмент, описанный директивой .STACK, если не был указан модификатор FARSTACK, и сегмент .CODE в модели TINY автоматически объединяются в группу с именем FLAT — для модели памяти FLAT или DGROUP — для всех остальных моделей. При этом сегментный регистр

DS (и SS, если не было FARSTACK, и CS в модели TINY) настраивается на всю эту группу, как если бы была выполнена команда ASSUME.

4.4 Способы адресации

Большинство команд процессора вызывается с аргументами, которые в ассемблере принято называть *операндами*. Например: команда сложения содержимого регистра с числом требует задания двух операндов — содержимого регистра и числа. Далее рассмотрены все существующие способы задания адреса хранения операндов — способы адресации.

4.4.1 Регистровая адресация

Операнды могут располагаться в любых регистрах общего назначения и сегментных регистрах. В этом случае в тексте программы указывается название соответствующего регистра, например команда, копирующая в регистр AX содержимое регистра BX, записывается как

```
mov     ax,bx
```

4.4.2 Непосредственная адресация

Некоторые команды (все арифметические команды, кроме деления) позволяют указывать один из операндов непосредственно в тексте программы, например команда

```
mov     ax,2
```

помещает в регистр AX число 2.

4.4.3 Прямая адресация

Если известен адрес операнда, располагающегося в памяти, можно использовать этот адрес. Если операнд — слово, находящееся в сегменте, на который указывает ES, со смещением от начала сегмента 0001, то команда

```
mov     ax,es:0001
```

поместит это слово в регистр AX. В реальных программах обычно для задания статических переменных используют директивы определения данных (глава 3.3), которые позволяют ссылаться на статические переменные не по адресу, а по имени. Тогда, если в сегменте, указанном в ES, была описана переменная word_var размером в слово, можно записать ту же команду как

```
mov     ax,es:word_var
```

В таком случае ассемблер сам заменит слово «word_var» на соответствующий адрес. Если селектор сегмента данных находится в DS, имя сегментного регистра при прямой адресации можно не указывать, DS используется по умолчанию. Прямая адресация иногда называется адресацией по смещению.

Адресация отличается для реального и защищенного режимов. В реальном режиме (так же как и в режиме V86) смещение всегда 16-битное, это значит, что ни непосредственно указанное смещение, ни результат сложения содержимого разных регистров в более сложных методах адресации не могут превышать границ слова. При программировании для Windows, для DOS4G, PMODE и в других ситуациях, когда программа будет запускаться в защищенном режиме, смещение не может превышать границ двойного слова.

4.4.4 Косвенная адресация

По аналогии с регистровыми и непосредственными операндами адрес операнда в памяти также можно не указывать непосредственно, а хранить в любом регистре. До 80386 для этого можно было использовать только BX, SI, DI и BP, но потом эти ограничения были сняты и адрес операнда разрешили считывать также и из EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP (но не из AX, CX, DX или SP напрямую — надо использовать EAX, ECX, EDX, ESP соответственно или предварительно скопировать смещение в BX, SI, DI или BP). Например, следующая команда помещает в регистр AX слово из ячейки памяти, селектор сегмента которой находится в DS, а смещение — в BX:

```
mov     ax,[bx]
```

Как и в случае прямой адресации, DS используется по умолчанию, но не во всех случаях: если смещение берут из регистров ESP, EBP или BP, то в качестве сегментного регистра используется SS. В реальном режиме можно свободно пользоваться всеми 32-битными регистрами, надо только следить, чтобы их содержимое не превышало границ 16-битного слова.

4.4.5 Адресация по базе со сдвигом

Теперь скомбинируем два предыдущих метода адресации: следующая команда

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

помещает в регистр AX слово, находящееся в сегменте, указанном в DS, со смещением на 2 большим, чем число, находящееся в BX. Так как слово занимает ровно два байта, эта команда поместила в AX слово,

непосредственно следующее за тем, которое есть в предыдущем примере. Такая форма адресации используется в тех случаях, когда в регистре находится адрес начала структуры данных, а доступ надо осуществить к какому-нибудь элементу этой структуры. Другое важное применение адресации по базе со сдвигом — доступ из подпрограммы к параметрам, переданным в стеке, используя регистр BP (EBP) в качестве базы и номер параметра в качестве смещения, что детально разобрано в параграфе 5.2. Другие допустимые формы записи этого способа адресации:

```
mov    ax,[bp]+2
mov    ax,2[bp]
```

До 80386 в качестве базового регистра можно было использовать только BX, BP, SI или DI и сдвиг мог быть только байтом или словом (со знаком). Начиная с 80386 и старше, процессоры Intel позволяют дополнительно использовать EAX, EBX, ECX, EDX, EBP, ESP, ESI и EDI, так же как и для обычной косвенной адресации. С помощью этого метода можно организовывать доступ к одномерным массивам байт: смещение соответствует адресу начала массива, а число в регистре — индексу элемента массива, который надо считать. Очевидно, что, если массив состоит не из байт, а из слов, придется умножать базовый регистр на два, а если из двойных слов — на четыре. Для этого предусмотрен следующий специальный метод адресации.

4.4.6 Косвенная адресация с масштабированием

Этот метод адресации полностью идентичен предыдущему, за исключением того, что с его помощью можно прочитать элемент массива слов, двойных слов или учетверенных слов, просто поместив номер элемента в регистр:

```
mov    ax,[esi*2]+2
```

Множитель, который может быть равен 1, 2, 4 или 8, соответствует размеру элемента массива — байту, слову, двойному слову, учетверенному слову соответственно. Из регистров в этом варианте адресации можно использовать только EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, но не SI, DI, BP или SP, которые можно было использовать в предыдущих вариантах.

4.4.7Адресация по базе с индексированием

В этом методе адресации смещение операнда в памяти вычисляется как сумма чисел, содержащихся в двух регистрах, и смещения, если оно указано. Все следующие команды — это разные формы записи одного и того же действия:

```
mov     ax,[bx+si+2]
mov     ax,[bx][si]+2
mov     ax,[bx+2][si]
mov     ax,[bx][si+2]
mov     ax,2[bx][si]
```

В регистр AX помещается слово из ячейки памяти со смещением, равным сумме чисел, содержащихся в BX и SI, и числа 2. Из шестнадцатибитных регистров так можно складывать только BX + SI, BX + DI, BP + SI и BP + DI, а из 32-битных — все восемь регистров общего назначения. Так же как и для прямой адресации, вместо непосредственного указания числа можно использовать имя переменной, заданной одной из директив определения данных. Так можно прочитать, например, число из двумерного массива: если задана таблица 10x10 байт, 2 — смещение ее начала от начала сегмента данных (на практике будет использоваться имя этой таблицы), BX = 20, а SI = 7, приведенные команды прочитают слово, состоящее из седьмого и восьмого байт третьей строки. Если таблица состоит не из одиночных байт, а из слов или двойных слов, удобнее использовать следующую, наиболее полную форму адресации.

Смещение может быть байтом или двойным словом. Если ESP или EBP используются в роли базового регистра, селектор сегмента операнда берется по умолчанию из регистра SS, во всех остальных случаях — из DS.

Примеры использования режимов адресации:

Пример 1 (регистровая адресация):

```
mov AX, BX ; копируются данные из BX в AX
mov DS, AX ; сегментный регистр DS инициализируется числом из AX
```

Пример 2 (непосредственная адресация):

```
; счетчик проинициализировать числом 10:
mov CX, 10
; нельзя инициализировать сегментный регистр непосредственным значением:
mov DS, 0000h ; недопустимо !
; можно сделать так:
mov AX, 0000h
mov DS, AX
```

Пример 3 (прямая адресация):

```
; скопировать в BX слово по адресу 0000:0000h:
mov AX, 0000h
mov ES, AX
mov BX, ES:0000h
; если в сегменте, на который указывает DS, есть переменная с именем X и
; размера слово, то значение хранящееся в DX можно скопировать в X так:
mov X, DX
```

Пример 4 (косвенная адресация):

```
; пусть в сегменте, на который указывает DS, первым объявлено слово X,
; а следом за ним - слово Y; тогда их значения можно скопировать так:
mov BX, 0000h
mov AX, [BX]
mov BX, 0002h
mov CX, [BX]
```

Пример 5 (адресация по базе со сдвигом):

```
; следующие команды из примера 4 можно заменить на одну:
; mov BX, 0002h
; mov CX, [BX]
mov CX, [BX]+2
```

Пример 6 (адресация по базе с индексированием):

```
; пусть объявлен массив слов X; тогда получить значения
; первого, третьего и пятого элемента можно так:
mov BX, offset X
mov SI, 0000h
mov AX, [BX][SI] ; 1
add SI, 4
mov AX, [BX][SI] ; 3
add SI, 4
mov AX, [BX][SI] ; 5
```

Ниже приведены примеры использования команд push/pop:

Пример 7: Временное хранение данных.

```
push    eax     ; сохраняем текущее значение eax в стеке
...             ; выполняем какие нибудь действия,
...             ; использующие eax
pop     eax     ; восстанавливаем прежнее значение eax
```

Пример 8: Копирование содержимого одного сегментного регистра в другой.

```
push    ds
pop     es
```

Пример 9: Выделение старшей части расширенного регистра данных (с помещением значения в другой регистр).

```
push    eax     ; помещаем значение eax в стек (4 байта)
pop     ax      ; извлекаем младшую часть (2 байта)
pop     bx      ; извлекаем старшую часть (2 байта)
```

5.5 Структура программы

Программа на языке ассемблера состоит из строк, имеющих следующий вид:

метка команда/директива операнды ; комментарий

Причем все эти поля необязательны. Метка может быть любой комбинацией букв английского алфавита, цифр и символов _, $, @, ?, но цифра не может быть первым символом метки, а символы $ и ? иногда имеют специальные значения и обычно не рекомендуются к использованию. Большие и маленькие буквы по умолчанию не различаются, но различие можно включить, задав ту или иную опцию в командной строке ассемблера. Во втором поле, поле команды, может располагаться команда процессора, которая транслируется в исполняемый код, или директива, которая не приводит к появлению нового кода, а управляет работой самого ассемблера. В поле операндов располагаются требуемые командой или директивой операнды (то есть нельзя указать операнды и не указать команду или директиву). И наконец, в поле комментариев, начало которого отмечается символом ; (точка с запятой), можно написать все что угодно — текст от символа «;» до конца строки не анализируется ассемблером.

Для облегчения читаемости ассемблерных текстов принято, что метка начинается на первой позиции в строке, команда — на 17-й (две табуляции), операнды — на 25-й (три табуляции) и комментарии — на 41-й или 49-й. Если строка состоит только из комментария, его начинают с первой позиции.

Если метка располагается перед командой процессора, сразу после нее всегда ставится символ «:» (двоеточие), который указывает ассемблеру, что надо создать переменную с этим именем, содержащую адрес текущей команды:

```
some_loop:
    lodsw           ; считать слово из строки,
    cmp     ax,7    ; если это 7 - выйти из цикла
    loopne  some_loop
```

Когда метка стоит перед директивой ассемблера, она обычно оказывается одним из операндов этой директивы и двоеточие не ставится. Рассмотрим директивы, работающие напрямую с метками и их значениями, — LABEL, EQU и =.

метка label тип

Директива LABEL определяет метку и задает ее тип. Тип может быть одним из: BYTE (байт), WORD (слово), DWORD (двойное слово), FWORD (6 байт), QWORD (учетверенное слово), TBYTE (10 байт), NEAR (ближняя метка), FAR (дальняя метка). Метка получает значение, равное адресу следующей команды или следующих данных, и тип, указанный явно. В зависимости от типа команда

```
mov метка,0
```

запишет в память байт (слово, двойное слово и т.д.), заполненный нулями, а команда

```
call метка
```

выполнит ближний или дальний вызов подпрограммы.

С помощью директивы LABEL удобно организовывать доступ к одним и тем же данным, как к байтам, так и к словам, определив перед данными две метки с разными типами.

метка equ выражение

Директива EQU присваивает метке значение, которое определяется как результат целочисленного выражения в правой части. Результатом этого выражения может быть целое число, адрес или любая строка символов:

```
truth    equ     1
message1 equ     'Try again$'
var2     equ     4[si]
         cmp     ax,truth     ; cmp ax,1
         db      message1     ; db 'Try again$'
         mov     ax,var2      ; mov ax, 4[si]
```

Директива EQU чаще всего используется с целью введения параметров, общих для всей программы, аналогично команде #define препроцессора языка C.

метка = выражение

Директива = эквивалентна EQU, но определяемая ею метка может принимать только целочисленные значения. Кроме того, метка, указанная этой директивой, может быть переопределена.

Каждый ассемблер предлагает целый набор специальных предопределенных меток — это может быть текущая дата (@date или ??date), тип процессора (@cpu) или имя того или иного сегмента программы, но

единственная предопределенная метка, поддерживаемая всеми рассматриваемыми нами ассемблерами, — $. Она всегда соответствует текущему адресу. Например, команда

```
jmp     $
```

выполняет безусловный переход на саму себя, так что создается вечный цикл из одной команды.

5.6 Директивы распределения памяти

5.6.1 Псевдокоманды определения переменных

Псевдокоманда — это директива ассемблера, которая приводит к включению данных или кода в программу, хотя сама она никакой команде процессора не соответствует. Псевдокоманды определения переменных указывают ассемблеру, что в соответствующем месте программы располагается переменная, определяют тип переменной (байт, слово, вещественное число и т.д.), задают ее начальное значение и ставят в соответствие переменной метку, которая будет использоваться для обращения к этим данным. Псевдокоманды определения данных записываются в общем виде следующим образом:

имя_переменной d* значение

где D* — одна из нижеприведенных псевдокоманд:

DB — определить байт;

DW — определить слово (2 байта);

DD — определить двойное слово (4 байта);

DF — определить 6 байт (адрес в формате 16-битный селектор: 32-битное смещение);

DQ — определить учетверенное слово (8 байт);

DT — определить 10 байт (80-битные типы данных, используемые FPU).

Поле значения может содержать одно или несколько чисел, строк символов (взятых в одиночные или двойные кавычки), операторов ? и DUP, разделенных запятыми. Все установленные таким образом данные окажутся в выходном файле, а имя переменной будет соответствовать адресу первого из указанных значений. Например, набор директив

```
text_string    db   'Hello world!'
number         dw   7
table          db   1,2,3,4,5,6,7,8,9,0Ah,0Bh,0Ch,0Dh,0Eh,0Fh
float_number   dd   3.5e7
```

заполняет данными 33 байта. Первые 12 байт содержат ASCII-коды символов строки «Hello world!», и переменная text_string указывает на первую букву в этой строке, так что команда

```
mov     al,text_string
```

считает в регистр AL число 48h (код латинской буквы H). Если вместо точного значения указан знак ?, переменная считается неинициализированной и ее значение на момент запуска программы может оказаться любым. Если нужно заполнить участок памяти повторяющимися данными, используется специальный оператор DUP, имеющий формат счетчик DUP (значение). Например, вот такое определение:

```
table_512w     dw   512 dup(?)
```

создает массив из 512 неинициализированных слов, на первое из которых указывает переменная table_512w. В качестве аргумента в операторе DUP могут выступать несколько значений, разделенных запятыми, и даже дополнительные вложенные операторы DUP.

6 .Задание для самостоятельного выполнения

1. Изучить теоретический материал.
2. Ознакомиться с пакетами MASM и TASM, воспользовавшись программой из работы.
3. Приведённые примеры скомпилировать и просмотреть в отладчике.

7. Контрольные вопросы

1. Что такое макроопределения и как их использовать в языке ассемблера?
2. Какие модели памяти используются при написании программ на языке Ассемблера, чем они отличаются?
3. Какова структура программы на языке Ассемблера?
4. Как происходит получение выполняемого кода программы?
5. Какую опцию необходимо применять при ассемблировании программы, для получения возможности ее отлаживать?
6. Как распечатать листинг программы?
7. Как правильно заканчивать программу на языке ассемблера?
8. Какие типы данных позволяет создать директива DB?

Лабораторная работа №2 Режимы адресации. Команды передачи данных

1 Цель работы

Изучить сегментную организацию памяти. Исследовать работу и научиться использовать команды передачи данных. Познакомиться на практике с режимами адресации.

2 Теоретические основы

Большинство команд процессора вызывается с аргументами, которые в ассемблере принято называть *операндами*. Например: команда сложения содержимого регистра с числом требует задания двух операндов — содержимого регистра и числа. Далее рассмотрены все существующие способы задания адреса хранения операндов — способы адресации.

2.1 Регистровая адресация

В зависимости от того откуда и куда передаются данные и каким способом, различаются несколько режимов адресации:

- регистровая;
- непосредственная;
- прямая;
- косвенная;
- по базе со смещением;
- по базе с индексированием;
- стековая.

Этот вид адресации предполагает передачу данных из одного регистра в другой.

Пример 2.1 Регистровая адресация

```
mov EAX, EBX ; копируются данные из EBX в EAX
mov DX, BX ; копируются данные из DX в BX
mov AL, CH ; копируются данные из AL в CH
xchg AL, AH ; обмен значений регистров AL и AH
```

2.2 Непосредственная адресация

При этом виде адресации происходит загрузка в регистр константы непосредственно записанной в команде.

Пример 2.2 Непосредственная адресация

```
; счетчик проинициализировать числом 10
mov ECX, 10
; Поместить в EAX число 10h (16)
mov EAX, 10h
; загрузка адресов в индексный регистр
lea ESI, X
; загрузка адресов в регистр EBX
lea EBX, Y
```

2.3 Прямая адресация

Здесь предполагается передача данных из ячейки памяти в другую ячейку памяти, или из ячейки памяти в регистр, или из регистра в ячейку памяти.

Пример 2.3 Прямая адресация

```
; скопировать в BX слово по адресу ES:00404000
; (Необходимо иметь доступ к данным, расположеннм по этому адресу!
; В противном сучае обращение вызовет исключение.)
mov BX, ES:00404000h
; Если в сегменте, на который указывает DS, есть переменная с именем X
; размером двойное слово, то значение хранящееся в EDX можно сохранить
; в X так:
mov [X], EDX
; или так:
mov X, EDX
; Поместить данные из переменной в регистр:
mov EAX, Z
```

2.4 Косвенная адресация

Этот вид адресации предполагает использование базового регистра BX/EBX, используемого в качестве указателя.

Пример 2.4 Косвенная адресация

```
; пусть в сегменте, на который указывает DS, первым объявлено слово p,
; а следом за ним - слово q; тогда их значения можно скопировать так:
lea EBX, [p] ; Получаем эффективный адрес переменной
mov AX, [EBX] ; Копируем данные в регистр из яейки с адресом равным EBX
add EBX, 000000002h ; Перемещаемся к следующей ячейке (к переменной q)
mov CX, [EBX] ; Копируем данные в регистр из яейки с адресом равным EBX
```

2.5 Адресация по базе со сдвигом

Теперь скомбинируем два предыдущих метода адресации: следующая команда

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

помещает в регистр AX слово, находящееся в сегменте, указанном в DS, со смещением на 2 большим, чем число, находящееся в BX. Так как слово занимает ровно два байта, эта команда поместила в AX слово, непосредственно следующее за тем, которое есть в предыдущем примере. Такая форма адресации используется в тех случаях, когда в регистре находится адрес начала структуры данных, а доступ надо осуществить к какому-нибудь элементу этой структуры. Другое важное применение адресации по базе со сдвигом — доступ из подпрограммы к параметрам, переданным в стеке, используя регистр BP (EBP) в качестве базы и номер параметра в качестве смещения, что детально разобрано в параграфе 5.2. Другие допустимые формы записи этого способа адресации:

```
mov     ax,[bp]+2
mov     ax,2[bp]
```

До 80386 в качестве базового регистра можно было использовать только BX, BP, SI или DI и сдвиг мог быть только байтом или словом (со знаком). Начиная с 80386 и старше, процессоры Intel позволяют дополнительно использовать EAX, EBX, ECX, EDX, EBP, ESP, ESI и EDI, так же как и для обычной косвенной адресации. С помощью этого метода можно организовывать доступ к одномерным массивам байт: смещение соответствует адресу начала массива, а число в регистре — индексу элемента массива, который надо считать. Очевидно, что, если массив состоит не из байт, а из слов, придется умножать базовый регистр на два, а если из двойных слов — на четыре. Для этого предусмотрен следующий специальный метод адресации.

```
; следующие команды из примера 4 можно заменить на одну:
; lea EBX, [p]
; add EBX, 000000002h
; mov CX, [EBX]
mov CX, [EBX + 000000002h]
```

2.6 Косвенная адресация с масштабированием

Этот метод адресации полностью идентичен предыдущему, за исключением того, что с его помощью можно прочитать элемент массива слов, двойных слов или учетверенных слов, просто поместив номер элемента в регистр:

```
mov     ax,[esi*2]+2
```

Множитель, который может быть равен 1, 2, 4 или 8, соответствует размеру элемента массива — байту, слову, двойному слову, учетверенному слову соответственно. Из регистров в этом варианте адресации можно использовать только EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, но не SI, DI, BP или SP, которые можно было использовать в предыдущих вариантах.

2.7Адресация по базе с индексированием

В этом методе адресации смещение операнда в памяти вычисляется как сумма чисел, содержащихся в двух регистрах, и смещения, если оно указано. Все следующие команды — это разные формы записи одного и того же действия:

```
mov    ax,[bx+si+2]
mov    ax,[bx][si]+2
mov    ax,[bx+2][si]
mov    ax,[bx][si+2]
mov    ax,2[bx][si]
```

В регистр AX помещается слово из ячейки памяти со смещением, равным сумме чисел, содержащихся в BX и SI, и числа 2. Из шестнадцатибитных регистров так можно складывать только BX + SI, BX + DI, BP + SI и BP + DI, а из 32-битных — все восемь регистров общего назначения. Так же как и для прямой адресации, вместо непосредственного указания числа можно использовать имя переменной, заданной одной из директив определения данных. Так можно прочитать, например, число из двумерного массива: если задана таблица 10x10 байт, 2 — смещение ее начала от начала сегмента данных (на практике будет использоваться имя этой таблицы), BX = 20, а SI = 7, приведенные команды прочитают слово, состоящее из седьмого и восьмого байт третьей строки. Если таблица состоит не из одиночных байт, а из слов или двойных слов, удобнее использовать следующую, наиболее полную форму адресации.

```
; Пусть объявлен массив слов array; тогда получить значения
; первого, третьего и пятого элемента можно так:
mov EBX, offset array
mov ESI, 00000000h
mov AX, [EBX][ESI] ; 1
add ESI, 4
mov AX, [EBX][ESI] ; 3
add ESI, 4
mov AX, [EBX][ESI] ; 5
```

Смещение может быть байтом или двойным словом. Если ESP или EBP используются в роли базового регистра, селектор сегмента операнда берется по умолчанию из регистра SS, во всех остальных случаях — из DS.

Примеры использования режимов адресации:

Пример 1 (регистровая адресация):
mov AX, BX ; копируются данные из BX в AX
mov DS, AX ; сегментный регистр DS инициализируется числом из AX

Пример 2 (непосредственная адресация):
; счетчик проинициализировать числом 10:
mov CX, 10
; нельзя инициализировать сегментный регистр непосредственным значением:
mov DS, 0000h ; недопустимо !
; можно сделать так:
mov AX, 0000h
mov DS, AX

Пример 3 (прямая адресация):
; скопировать в BX слово по адресу 0000:0000h:
mov AX, 0000h
mov ES, AX
mov BX, ES:0000h
; если в сегменте, на который указывает DS, есть переменная с именем X и
; размера слово, то значение хранящееся в DX можно скопировать в X так:
mov X, DX

Пример 4 (косвенная адресация):
; пусть в сегменте, на который указывает DS, первым объявлено слово X,
; а следом за ним - слово Y; тогда их значения можно скопировать так:

```
mov BX, 0000h
mov AX, [BX]
mov BX, 0002h
mov CX, [BX]
```

Пример 5 (адресация по базе со сдвигом):

```
; следующие команды из примера 4 можно заменить на одну:
; mov BX, 0002h
; mov CX, [BX]
mov CX, [BX]+2
```

Пример 6 (адресация по базе с индексированием):

```
; пусть объявлен массив слов X; тогда получить значения
; первого, третьего и пятого элемента можно так:
mov BX, offset X
mov SI, 0000h
mov AX, [BX][SI] ; 1
add SI, 4
mov AX, [BX][SI] ; 3
add SI, 4
mov AX, [BX][SI] ; 5
```

Ниже приведены примеры использования команд push/pop:

Пример 7: Временное хранение данных.

```
push    eax     ; сохраняем текущее значение eax в стеке
...             ; выполняем какие нибудь действия,
...             ; использующие eax
pop     eax     ; восстанавливаем прежнее значение eax
```

Пример 8: Копирование содержимого одного сегментного регистра в другой.

```
push    ds
pop     es
```

Пример 9: Выделение старшей части расширенного регистра данных (с помещением значения в другой регистр).

```
push    eax     ; помещаем значение eax в стек (4 байта)
pop     ax      ; извлекаем младшую часть (2 байта)
pop     bx      ; извлекаем старшую часть (2 байта)
```

2.8 Стековая адресация

Стековая адресация предназначена для передачи данных из стека и в стек. Для этого предназначены команды push и pop, адресующие ячейки памяти посредством регистра SP, и команды mov, использующие регистр BP/EBP.

Пример 10 Временное хранение данных

```
push EAX ; сохраняем текущее значение EAX в стеке
... ; выполняем какие нибудь действия,
... ; использующие EAX
pop EAX ; восстанавливаем прежнее значение EAX
```

Пример 11 Копирование содержимого одного сегментного регистра в другой

```
push DS
pop ES
```

Пример 12 Выделение старшей части расширенного регистра данных (с помещением значения в другой регистр)

```
push EAX ; помещаем значение eax в стек (4 байта)
pop AX ; извлекаем младшую часть (2 байта)
pop BX ; извлекаем старшую часть (2 байта)
```

Пример 13 Извлечение данных из произвольной ячейки стека

```
mov EBP, ESP
mov AX, [EBP + 2]
```

В общем виде общую схему адресации (объединяющую все виды адресации) можно представить как сумму следующих значений:
Эффективный адрес = База + (Масштаб*Индекс) + Смещение

2.8 Пересылка данных

MOV приемник, источник

Базовая команда пересылки данных. Копирует содержимое источника в приемник, источник не изменяется. Команда MOV действует аналогично операторам присваивания из языков высокого уровня, то есть команда

```
mov ax,bx
```

эквивалентна выражению

```
ax = bx;
```

языка C, за исключением того, что команда ассемблера позволяет работать не только с переменными в памяти, но и со всеми регистрами процессора.

В качестве источника для MOV могут использоваться: число (непосредственный операнд), регистр общего назначения, сегментный регистр или переменная (то есть операнд, находящийся в памяти). В качестве приемника — регистр общего назначения, сегментный регистр (кроме CS) или переменная. Оба операнда должны быть одного и того же размера — байт, слово или двойное слово.

Нельзя выполнять пересылку данных с помощью MOV из одной переменной в другую, из одного сегментного регистра в другой и нельзя помещать в сегментный регистр непосредственный операнд — эти операции выполняют двумя командами MOV (из сегментного регистра в обычный и уже из него в другой сегментный) или парой команд PUSH/POP.

2.8.1 Условная пересылка данных

CMOVcc приемник, источник

Это набор команд, которые копируют содержимое источника в приемник, если удовлетворяется то или иное условие (см. табл.). Источником может быть регистр общего назначения или переменная, а приемником — только регистр. Условие, которое должно удовлетворяться, — просто равенство нулю или единице тех или иных флагов из регистра FLAGS, но, если использовать команды CMOVcc сразу после команды CMP (сравнение) с теми же операндами, условия приобретают особый смысл, например:

```
cmp     ax,bx     ; сравнить ax и bx
cmovl   ax,bx     ; если ax < bx, скопировать bx в ax
```

Слова «выше» и «ниже» в таблице относятся к сравнению чисел без знака, слова «больше» и «меньше» учитывают знак.

Таблица. *Разновидности команды CMOVcc*

Код команды	**Реальное условие**	**Условие для CMP**
CMOVA CMOVNBE	CF = 0 и ZF = 0	если выше если не ниже или равно
CMOVAE CMOVNB CMOVNC	CF = 0	если выше или равно если не ниже если нет переноса
CMOVB CMOVNAE CMOVC	CF = 1	если ниже если не выше или равно если перенос
CMOVBE CMOVNA	CF = 1 и ZF = 1	если ниже или равно если не выше
CMOVE CMOVZ	ZF = 1	если равно если ноль
CMOVG CMOVNLE	ZF = 0 и SF = OF	если больше если не меньше или равно
CMOVGE CMOVNL	SF = OF	если больше или равно если не меньше
CMOVL CMOVNGE	SF <> OF	если меньше если не больше или равно
CMOVLE CMOVNG	ZF = 1 и SF <> OF	если меньше или равно если не больше
CMOVNE CMOVNZ	ZF = 0	если не равно если не ноль
CMOVNO	OF = 0	если нет переполнения
CMOVO	OF = 1	если есть переполнение
CMOVNP CMOVPO	PF = 0	если нет четности если нечетное
CMOVP CMOVPE	PF = 1	если есть четность если четное
CMOVNS	SF = 0	если нет знака
CMOVS	SF = 1	если есть знак

2.8.2 Обмен операндов между собой

XCHG операнд1, операнд2

Содержимое операнда 2 копируется в операнд 1, а старое содержимое операнда 1 — в операнд 2. XCHG можно выполнять над двумя регистрами или над регистром и переменной.

```
xchg    eax,ebx    ; то же, что три команды на языке C:
                   ; temp = eax; eax = ebx; ebx = temp;
```

2.8.3 Поместить данные в стек

PUSH источник

Помещает содержимое источника в стек. Источником может быть регистр, сегментный регистр, непосредственный операнд или переменная. Фактически эта команда копирует содержимое источника в память по адресу SS:[ESP] и уменьшает ESP на размер источника в байтах (2 или 4). Команда PUSH практически всегда используется в паре с POP (считать данные из стека). Так, например, чтобы скопировать содержимое одного сегментного регистра в другой (что нельзя выполнить одной командой MOV), можно использовать такую последовательность команд:

```
push    cs
pop     ds    ; теперь DS указывает на тот же сегмент, что и CS
```

Другое частое применение команд PUSH/POP — временное хранение переменных, например:

```
push    eax   ; сохраняет текущее значение EAX
...           ; здесь располагаются какие-нибудь команды,
              ; которые используют EAX, например CMPXCHG
pop     eax   ; восстанавливает старое значение EAX
```

Начиная с 80286, команда PUSH ESP (или SP) помещает в стек значение ESP до того, как эта же команда его уменьшит, в то время как на 8086 SP помещался в стек уже уменьшенным на два.

2.8.4 Считать данные из стека

POP приемник

Помещает в приемник слово или двойное слово, находящееся в вершине стека, увеличивая ESP на 2 или 4 соответственно. POP выполняет

действие, полностью обратное PUSH. Приемником может быть регистр общего назначения, сегментный регистр, кроме CS (чтобы загрузить CS из стека, надо воспользоваться командой RET), или переменная. Если в роли приемника выступает операнд, использующий ESP для косвенной адресации, команда POP вычисляет адрес операнда уже после того, как она увеличивает ESP.

2.8.5 Вычисление эффективного адреса

LEA приемник, источник

Вычисляет эффективный адрес источника (переменная) и помещает его в приемник (регистр). С помощью LEA можно вычислить адрес переменной, которая описана сложным методом адресации, например по базе с индексированием. Если адрес 32-битный, а регистр-приемник 16-битный, старшая половина вычисленного адреса теряется, если наоборот, приемник 32-битный, а адресация 16-битная, то вычисленное смещение дополняется нулями.

Команду LEA часто используют для быстрых арифметических вычислений, например умножения:

```
lea     bx,[ebx+ebx*4] ; BX=EBX*5
```

или сложения:

```
lea     ebx,[eax+12]; EBX=EAX+12
```

(эти команды меньше, чем соответствующие MOV и ADD, и не изменяют флаги)

3. Задание для самостоятельного выполнения

1) Протестировать все примеры, приведенные в разделе "Краткая теория".
2) Оформить отчёт.

Лабораторная работа №3 Арифметические команды целочисленного устройства микропроцессора

1 Цель работы

Исследовать с помощью отладчика работу арифметических команд. Научиться использовать арифметические команды целочисленного устройства для вычисления простых выражений.

2 Теоретические основы

2.1 Следует различать две категории арифметических команд целочисленного устройства

- команды двоичной арифметики;
- команды двоично-десятичной арифметики.

2.1.1 Команды двоичной арифметики

- команды сложения add и adc; inc; xadd;
- команды вычитания sub и sbb; dec;
- команды умножения mul и imul;
- команды деления div и idiv;
- команда изменения знака neg.

Команда	**Назначение**
add приемник, источник	сложение
adc приемник, источник	сложение с переносом
xadd приемник, источник	обменять между собой и сложить
sub приемник, источник	вычитание
sbb приемник, источник	вычитание с заемом
imul источник	умножение чисел со знаком
imul приемник, источник	умножение чисел со знаком
imul приемник, источник1, источник2	умножение чисел со знаком
mul источник	умножение чисел без знака
idiv источник	целочисленное деление со знаком
div источник	целочисленное деление без знака
inc приемник	увеличение на 1 (инкремент)
dec приемник	уменьшение на 1 (декремент)

neg приемник изменение знака

2.1.2 Форматы арифметических команд

команда add:

add reg/mem, imm; размер 8/16/32 бита
add reg, reg/mem; размер 8/16/32 бита
add mem/reg, reg; размер 8/16/32 бита
add reg/mem16, imm8; получатель 16 бит, источник 8 бит
add reg/mem32, imm8; получатель 32 бит, источник 8 бит

Команда выполняет арифметическое сложение приемника и источника, помещает сумму в приемник, не изменяя содержимое источника. Приемник может быть регистром или переменной, источник может быть числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Команда ADD никак не различает числа со знаком и без знака, но, употребляя значения флагов CF (перенос при сложении чисел без знака), OF (перенос при сложении чисел со знаком) и SF (знак результата), можно использовать ее и для тех, и для других.

•команда adc:

adc reg/mem, imm; размер 8/16/32 бита
adc reg, reg/mem; размер 8/16/32 бита
adc mem/reg, reg; размер 8/16/32 бита
adc reg/mem16, imm8; получатель 16 бит, источник 8 бит
adc reg/mem32, imm8; получатель 32 бит, источник 8 бит

Эта команда во всем аналогична ADD, кроме того, что она выполняет арифметическое сложение приемника, источника и флага CF. Пара команд ADD/ADC используется для сложения чисел повышенной точности. Сложим, например, два 64-битных целых числа: пусть одно из них находится в паре регистров EDX:EAX (младшее двойное слово (биты 0 – 31) — в EAX и старшее (биты 32 – 63) — в EDX), а другое — в паре регистров EBX:ECX Если при сложении младших двойных слов произошел перенос из старшего разряда (флаг CF = 1), то он будет учтен следующей командой ADC.

• команда xadd:

xadd reg/mem, reg; размер 8/16/32 бита

Выполняет сложение, помещает содержимое приемника в источник, — сумму операндов — в приемник. Источник всегда регистр, приемник может быть регистром и переменной.

• команда sub:

sub reg/mem, imm; размер 8/16/32 бита
sub reg, reg/mem; размер 8/16/32 бита
sub mem/reg, reg; размер 8/16/32 бита
sub reg/mem16, imm8; получатель 16 бит, источник 8 бит
sub reg/mem32, imm8; получатель 32 бит, источник 8 бит

Вычитает источник из приемника и помещает разность в приемник. Приемник может быть регистром или переменной, источник может быть числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Точно так же, как и команда ADD, SUB не делает различий между числами со знаком и без знака, но флаги позволяют использовать ее как для тех, так и для других.

• **команда sbb:**

sbb reg/mem, imm; размер 8/16/32 бита
sbb reg, reg/mem; размер 8/16/32 бита
sbb mem/reg, reg; размер 8/16/32 бита
sbb reg/mem16, imm8; получатель 16 бит, источник 8 бит
sbb reg/mem32, imm8; получатель 32 бит, источник 8 бит

Эта команда во всем аналогична SUB, кроме того, что она вычитает из приемника значение источника и дополнительно вычитает значение флага CF. Так, можно использовать эту команду для вычитания 64-битных чисел в EDX:EAX и EBX:ECX аналогично ADD/ADC:

```
sub     eax,ecx
sbb     edx,ebx
```

Если при вычитании младших двойных слов произошел заем, то он будет учтен при вычитании старших.

• **команда imul:**

imul reg/mem; размер 8/16/32 бита
imul reg16, imm8; произведение 16 бит
imul reg16, imm16; произведение 16 бит
imul reg32, imm8; произведение 32 бита
imul reg32, imm32; произведение 32 бита
imul reg16, reg/mem16; произведение 16 бит
imul reg32, reg/mem32; произведение 32 бита
imul reg16, reg/mem16, imm8/16; произведение 16 бит
imul reg32, reg/mem32, imm8/32; произведение 32 бит

Эта команда имеет три формы, различающиеся числом операндов:

IMUL источник: источник (регистр или переменная) умножается на AL, AX или EAX (в зависимости от размера операнда), и результат располагается в AX, DX:AX или EDX:EAX соответственно.

IMUL приемник,источник: источник (число, регистр или переменная) умножается на приемник (регистр), и результат заносится в приемник.

IMUL приемник,источник1,источник2: источник 1 (регистр или переменная) умножается на источник 2 (число), и результат заносится в приемник (регистр).

Во всех трех вариантах считается, что результат может занимать в два раза больше места, чем размер источника. В первом случае приемник автоматически оказывается достаточно большим, но во втором и третьем случаях могут произойти переполнение и потеря старших бит результата. Флаги OF и CF будут равны единице, если это произошло, и нулю, если результат умножения поместился целиком в приемник (во втором и третьем случаях) или в младшую половину приемника (в первом случае).

Значения флагов SF, ZF, AF и PF после команды IMUL не определены.

• **команда mul:**

mul reg/mem; размер 8/16/32 бита

Выполняет умножение содержимого источника (регистр или переменная) и регистра AL, AX, EAX (в зависимости от размера источника) и помещает результат в AX, DX:AX, EDX:EAX соответственно. Если старшая половина результата (AH, DX, EDX) содержит только нули (результат целиком поместился в младшую половину), флаги CF и OF устанавливаются в 0, иначе — в 1. Значение остальных флагов (SF, ZF, AF и PF) не определено.

• **команда idiv:**

idiv reg/mem; размер 8/16/32 бита

Выполняет целочисленное деление со знаком AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL, AX или EAX, а остаток — в AH, DX или EDX соответственно. Результат всегда округляется в сторону нуля, знак остатка всегда совпадает со знаком делимого, абсолютное значение остатка всегда меньше абсолютного значения делителя. Значения флагов CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0 — в реальном.

• **команда div:**

div reg/mem; размер 8/16/32 бита

Выполняет целочисленное деление без знака AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL, AX или EAX, а остаток — в AH, DX или EDX соответственно. Результат всегда округляется в сторону нуля, абсолютное значение остатка всегда меньше абсолютного значения делителя. Значения флагов CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0 — в реальном.

• **команда inc:**

inc reg/mem; размер 8/16/32 бита

Увеличивает приемник (регистр или переменная) на 1. Единственное отличие этой команды от ADD приемник,1 состоит в том, что флаг CF не затрагивается. Остальные арифметические флаги (OF, SF, ZF, AF, PF) устанавливаются в соответствии с результатом сложения.

• **команда dec:**

dec reg/mem; размер 8/16/32 бита

Уменьшает приемник (регистр или переменная) на 1. Единственное отличие этой команды от SUB приемник,1 состоит в том, что флаг CF не затрагивается. Остальные арифметические флаги (OF, SF, ZF, AF, PF) устанавливаются в соответствии с результатом вычитания.

• **команда neg:**

neg reg/mem; размер 8/16/32 бита

Выполняет над числом, содержащимся в приемнике (регистр или переменная), операцию дополнения до двух. Эта операция эквивалентна обращению знака операнда, если рассматривать его как число со знаком. Если приемник равен нулю, флаг CF устанавливается в 0, иначе — в 1. Остальные флаги (OF, SF, ZF, AF, PF) устанавливаются в соответствии с результатом операции.

2.1.3 Двоично-десятичный формат целых чисел

• упакованный: XXXX XXXX ;

• неупакованный: 0000 XXXX .

Четыре бита XXXX представляют десятичную цифру 0-9.

Все обычные операции над такими числами приводят к неправильным результатам.

2.1.4 Команды десятичной арифметики

1. Команда BCD-коррекции после сложения DAA;

Пример: 19h + 1h = 1Ah; после DAA: 20h.

Если эта команда выполняется сразу после ADD (ADC, INC или XADD) и в регистре AL находится сумма двух упакованных двоично-десятичных чисел, то в результате в AL записывается упакованное двоично-десятичное число, которое должно было быть результатом сложения. Например, если AL содержит число 19h, последовательность команд

```
inc  al
daa
```

приведет к тому, что в AL окажется 20h (а не 1Ah, как было бы после INC).

DAA выполняет следующие действия:

Если младшие четыре бита AL больше 9 или флаг AF = 1, то AL увеличивается на 6, CF устанавливается, если при этом сложении произошел перенос, и AF устанавливается в 1. Иначе AF = 0. Если теперь старшие четыре бита AL больше 9 или флаг CF = 1, то AL увеличивается на 60h и CF устанавливается в 1. Иначе CF = 0.

2. Команда BCD-коррекции после вычитания DAS;

Пример: 20h - 1h = 1Fh; после DAS: 19h.

Если эта команда выполняется сразу после SUB (SBB или DEC) и в регистре AL находится разность двух упакованных двоично-десятичных чисел, то в результате в AL записывается упакованное двоично-десятичное число, которое должно было быть результатом вычитания. Например, если AL содержит число 20h, последовательность команд

```
dec  al
das
```

приведет к тому, что в AL окажется 19h (а не 1Fh, как было бы после DEC).

DAS выполняет следующие действия:

Если младшие четыре бита AL больше 9 или флаг AF = 1, то AL уменьшается на 6, CF устанавливается, если при этом вычитании произошел заем, и AF устанавливается в 1. Иначе AF = 0. Если теперь старшие четыре бита AL больше 9 или флаг CF = 1, то AL уменьшается на 60h и CF устанавливается в 1. Иначе CF = 0.

3. Команда ASCII-коррекции после сложения AAA;

Пример: 05h + 06h = 0Bh; после AAA: 0101h.

Корректирует сумму двух неупакованных двоично-десятичных чисел в AL. Если коррекция приводит к десятичному переносу, AH увеличивается на 1. Эта команда имеет смысл сразу после команды сложения двух таких чисел. Например, если при сложении 05 и 06 в AX окажется число 000Bh, то команда AAA скорректирует его в 0101h (неупакованное десятичное 11). Флаги CF и OF устанавливаются в 1, если произошел перенос из AL в AH, иначе они равны нулю. Значения флагов OF, SF, ZF и PF не определены.

4. Команда ASCII-коррекции после вычитания AAS;

Корректирует разность двух неупакованных двоично-десятичных чисел в AL сразу после команды SUB или SBB. Если коррекция приводит к займу, AH уменьшается на 1. Флаги CF и OF устанавливаются в 1, если произошел заем из AL в AH, и в ноль — в противном случае. Значения флагов OF, SF, ZF и PF не определены.

5. Команда ASCII-коррекции после умножения AAM;

Пример: mov AL, 5; mov BL, 5; mul BL; aam; 0019h -> 0205h

Корректирует результат умножения неупакованных двоично-десятичных чисел, находящийся в AX после выполнения команды MUL, преобразовывая полученный результат в пару неупакованных двоично-десятичных чисел (в AH и AL). Например:

```
mov     al,5
mov     bl,5     ; умножить 5 на 5
mul     bl       ; результат в AX - 0019h
aam              ; теперь AX содержит 0205h
```

AAM устанавливает флаги SF, ZF и PF в соответствии с результатом и оставляет OF, AF и CF неопределенными.

Код команды AAM — D4h 0Ah, где 0Ah — основание системы счисления, по отношению к которой выполняется коррекция. Этот байт можно заменить на любое другое число (кроме нуля), и AAM преобразует AX к двум неупакованным цифрам любой системы счисления. Такая обобщенная форма AAM работает на всех процессорах (начиная с 8086), но появляется в документации Intel только с процессоров Pentium. Фактически действие, которое выполняет AAM, — целочисленное деление AL на 0Ah (или любое другое число в общем случае), частное помещается в AL, и остаток — в AH, так что эту команду часто используют для быстрого деления в высокооптимизированных алгоритмах.

6. Команда ASCII-коррекции перед делением AAD;

Пример: mov AX, 0205h; mov BL, 5; aad; AX = 19h; div BL; AX = 05h

Выполняет коррекцию неупакованного двоично-десятичного числа, находящегося в регистре AX, так, чтобы последующее деление привело к корректному десятичному результату. Например, разделим десятичное 25 на 5:

```
mov     ax,0205h  ; 25 в неупакованном формате
mov     bl,5
aad               ; теперь в AX находится 19h
div     bl        ; AX = 0005
```

Флаги SF, ZF и PF устанавливаются в соответствии с результатом, OF, AF и CF не определены.

AAD используется с любой системой счисления: ее код — D5h 0Ah, и второй байт можно заменить на любое другое число. Действие AAD состоит в том, что содержимое регистра AH умножается на второй байт команды (0Ah по умолчанию) и складывается с AL, после чего AH обнуляется, так что AAD можно использовать для быстрого умножения на любое число.

2.1.5 Арифметические команды воздействуют на следующие флаги

• CF(перенос);
• PF(чётность);
• AX(вспомогательный перенос);
• ZF(нуль);
• SF(знак);
• OF(переполнение).

Ситуация переноса\заема (CF) возникает, когда результат сложения\вычитания не помещается в приёмнике.

Пример:

```
  FF = 11111111 = 255
+FF = 11111111 = 255
--------------------
1FE = 111111110 = 510
        ^перенос
```

PF -> 1, когда результат арифметической операции имеет чётное число 1 в младших 8-ми битах.

Ситуация вспомогательного переноса\заема (AX) возникает при переносе из 3-го бита в 4-й.

Полезна в операциях десятичной арифметике.

ZF -> 1, когда результат арифметической операции 0.
SF -> 1, когда 1 в старшем разряде результата (результат отрицательный).
Ситуация переполнения (OF) возникает, когда результат умножения не помещается в приёмнике.

2.2 Пример

Пример программы, вычисляющей следующее арифметическое выражение:

$$Z = - \frac{X2 + XY - 2}{(Y + 1)3}$$

где X и Y - двойное слово.

```
CodeSg segment 'CODE'
        assume CS:CodeSg, DS:DataSg, SS:StackSg
begin:
        mov   AX, DataSg
        mov   DS, AX
        mov   BX, X       ; X -> BX ( BX=10=Ah )
        mov   AX, BX      ; X -> AX ( AX=10=Ah )
        mul   AX          ; AX = X^2 ( AX=100=64h )
        mov   CX, AX      ; X^2 -> CX ( CX=100=64h )
        mov   AX, BX      ; X -> AX ( AX=10=Ah )
        mov   BX, Y       ; Y -> BX ( BX=3 )
        mul   BX          ; AX = X*Y ( AX=30=1Eh )
        sub   AX, 2       ; AX = X*Y - 2 ( AX=28=1Ch )
        add   AX, CX      ; AX = (X*Y - 2) + X^2 ( AX=128=80h )
        mov   CX, AX      ; (X*Y - 2) + X^2 -> CX ( CX=128=80h )
        inc   BX          ; Y + 1 ( BX=4 )
        mov   AX, BX      ; (Y+1) -> AX ( AX=4 )
        mul   AX          ; AX = (Y+1)^2 ( AX=16=10h )
        mul   BX          ; AX = (Y+1)^2 * (Y+1) ( AX=64=40h )
        mov   BX, AX      ; (Y+1)^3 -> BX ( BX=64=40h )
        mov   AX, CX      ; (X*Y - 2) + X^2 -> AX ( AX=128=80h )
        mov   DX, 0       ; 0 -> DX ( DX=0 )
        div   BX          ; AX = AX / BX ( AX=2 )
```

```
        neg     AX              ; -AX ( AX=-2=FFFEh )
        mov     Z, AX           ; AX -> Z
        mov     AX, 4C00h
        int     21h
CodeSg ends
;-----------------------------------
DataSg segment 'DATA'
X       DW      10
Y       DW      3
Z       DW      ?
DataSg ends
;-----------------------------------
StackSg segment stack 'STACK'
        db      256 dup(0)
StackSg ends
;-----------------------------------
      end       begin
```

3 Задание для самостоятельного выполнения

Выполните 5 (пять) упражнений из ниже приведенного списка, выбирая по следующему принципу: пусть номер студента с списке группы N, тогда выполняются упражнения с номерами N, N+1, N+2, N+3, N+4. Написать программу, вычисляющую Z для заданных X и Y двумя способами (с помощью команд двоичной и двоично-десятичной арифметики).

1. Z = (X^2 + 2*Y - 45) / (X^3);
2. Z = 1 / Y + X^3 - 32;
3. Z = (3 + X/Y) / (X-Y+1);
4. Z = X / (X - Y + X*Y);
5. Z = (4 - (X+3)/(Y-1))*(-XY);
6. Z = ((X+1)/Y - 1)*2X;
7. Z = Y*(2-(Y+1)/X);
8. Z = (XY - 1)/(X+Y);
9. Z = X^3 + Y -1;
10. Z = (XY + 1)/ X^2;
11. Z = (X+Y)/(X-Y);
12. Z = - 1/X^3 + 3;
13. Z = X - Y/X +1;

14.Z = ((X+Y)/Y^2 - 1)*X;
15.Z = (X-Y)/(XY+1);
16.Z = - X/Y+Y^2 +3;
17.Z = Y^2 + XY + X/Y;
18.Z = (1 + X * Y)/2;
19.Z = -(1-Y)/(1+X);
20.Z = -X*(1-XY);
21.Z = Y+X/Y-1;
22.Z = 5/XY+X^3;
23.Z = -X + Y^3 - 1;
24.Z = X^3 / (X-Y);
25.Z = X^3 -2X^2*Y+1;
26.Z = -3X + Y^2 +1;
27.Z = -(X/Y +1)/Y^2;
28.Z = 1+X^2/3Y;
29.Z = Y-X/3+1;
30.Z = (XY)^3 +1/Y.

4 Контрольные вопросы

1. Какова двоичная форма представления целых чисел.
2. Что такое дополнительный код.
3. Какова шестнадцатеричная форма представления целых чисел.
4. Как представляются BCD и ASCII-форматы целых чисел;
5. Как объявляются поля данных размера DB, DW, DD.
6. Каковы форматы, предназначение и алгоритмы работы арифметических команд целочисленного устройства: add, adc, xadd, sub, sbb, imul, mul, idiv, div, inc, dec, neg; aaa, aas, aam, aad; or, xchg.

Лабораторная работа №4 Разработка программ обработки прерываний для режима реального адреса

1 Цель работы

научиться составлять программы обработки прерываний.

2 Порядок выполнения работы:

- -ознакомиться с описанием лабораторной работы;
- -получить задание у преподавателя по вариантам;
- -написать программу, ввести программу, отладить и решить ее на ЭВМ;
- -оформить отчет.

3 Теоретические основы

При получении сигнала на прерывание процессор заканчивает выполнение очередной команды: содержимое CS, IP и регистра флагов сохраняет в стеке, в IP, CS помещается адрес подпрограммы обработки данного прерывания и выполняет ее, затем восстанавливает из стека CS, IP и регистр флагов и продолжает выполнение прерванной программы.

Двойное слово, в котором находится адрес начала программы обработки прерывания, называется вектором прерывания. Всего допустимо иметь 256 различных номеров прерываний. Адрес вектора каждого прерывания находится по его номеру, умноженному на 4, так как для хранения двойного слова адреса требуется 4 байта. Векторы прерывания занимают первые 256*4=1024 байта памяти. Для определения адреса подпрограммы обработки прерывания умножьте номер прерывания на 4. В первых двух байтах полученного адреса размещено значение IP, а в следующих двух значение CS подпрограммы обработки данного прерывания. Если написать свою подпрограмму обработки прерываний и с помощью функций, описанных ниже, записать ее адрес в вектор прерывания, то при вызове данного прерывания его обработка будет проходить по новой подпрограмме. Старый вектор прерывания обычно запоминается, чтобы восстановить его значение после завершения программы. Каждая подпрограмма обработки прерывания завершается командой IRET, которая похожа на команду RET, но дополнительно восстанавливает из стека регистр флагов. Каждая подпрограмма обработки прерывания должна со-

хранять значения всех используемых ею регистров процессора аналогично обычной подпрограмме.

Прерывания могут быть вложенными, т.е. одно прерывание может вызывать другое внутри себя. Например, с каждым отсчетом таймера происходит аппаратное прерывание по вектору 08h. Однако оно, в свою очередь, всегда вызывает прерывание 1Ch. Это прерывание предназначено для программиста. Изначально система делает его указывающим на адрес в ПЗУ, по которому расположена лишь команда возврата из прерывания IRET. Программист может присоединиться к этому вектору с тем, чтобы процессор по каждому тику таймера (18,2 тика в сек.) выполнял некоторую дополнительную работу.

MS DOS предоставляет функции 35h и 25h прерывания 21h для чтения и установки вектора прерывания. Функция 35h выполняет чтение адреса подпрограммы обработки прерывания.

При вызове:

AH=35h

AL – номер прерывания

Возвращает:

ES:BX – указатель на подпрограмму прерывания, BX содержит смещение, ES сегмент подпрограммы.

```
mov     ah, 35h
mov     al, 21h
int     21h
; в результате bx = 0206h, es=0AAEh
```

Функция 25h устанавливает новый вектор прерывания.

При вызове:

AH=25h

AL – номер прерывания

DS:DX – указатель на программу обработки прерывания.

```
mov     ax, cod             ; cod -  имя сегмента программы обработки
mov     ds, ax          ;  прерывания
lea     dx, name        ; name – имя подпрограммы обработки прерывания
mov     ah, 25h             ;  номер функции
mov     al, 08h             ; номер прерывания
int     21h
```

Пример выполнения работы:

Задание: написать программу, выводящую в текущее положение курсора символ «@», следующий символ «@» выводить в позицию левее или правее, выше или ниже текущего положения, вывод осуществлять непрерывно с некоторой задержкой. Направление вывода определяется, нажатием клавиш «8, 2, 6, 4» на клавиатуре, нажатием клавиши «0» работа программы завершается, клавишей «5» меняет цвет выводимого символа с желтого на зеленый и наоборот, нажатие остальных клавиш игнорируется.

Текст программы:

```
data    segment
direct  db    1       ; направление перемещения
exit    db    0       ; признак окончания работы (если не 0)
old_cs  dw    ?       ; для хранения «старого» вектора прерываний
old_ip  dw    ?       ; с номером 1Ch
pos1    dw    3840    ; позиция для вывода символа (вначале левый
                        нижний угол)
sum     db    “@”     ; символ, выводимый на экран
atr     db    14      ; атрибут символа (желтый)
atr2    db    10      ; атрибут символа (зеленый)
data    ends
code    segment
        assume          cs:code, ds:data
new_1cproc  far     ; подпрограмма обработки прерываний 1Ch
        push  ax
        push  bx
        push  cx
        push  dx
        push  ds
        mov   ax, data
        mov   ds, ax
        push  es            ; проверка буфера клавиатуры
        mov   ax, 40h
        mov   es, ax        ; es на сегмент области данных BIOS
        mov   ax, es:[1Ch] ; в ax указатель «хвоста» буфера клавиатуры
        mov   bx, es:[1Ah] ; в bx указатель «головы»
        cmp   bx, ax        ; буфер пуст?
        jne   n5            ; нет – на n5
        pop   es            ; да – вернуться в основную программу
        jmp   back
```

```
n5:     mov   al, es:[bx]    ; извлечь символ из буфера
        mov   es:[1Ch], bx ; указатель «хвоста» на «голову»
        pop   es
        cmp   al, 30h              ; нажата ли клавиша «0» ?
        jnz   n1             ; нет – идем дальше
        mov   exit, 1        ; да – устанавливаем признак конца
        jmp   back           ; работы основной программы и выход
n1:     cmp   al, 35h              ; клавиша «5» ?
        jne   n6             ; нет – идем дальше
        mov   dl, atr        ; да – обмениваем значения атрибутов
        mov   dh, atr2
        mov   atr, dh
        mov   atr2, dl
        jmp   back           ; выход в основную программу
n6:     cmp   al, 34h              ; влево
        jz    n2
        cmp   al, 36h              ; вправо
        jz    n3
        cmp   al, 38h              ; вверх
        jz    n4
        cmp   al, 32h              ; вниз
        jnz   back           ; неиспользуемая клавиша - выход
        mov   direct, 4
        jmp   back
n2:     mov   direct, 2
        jmp   back
n3:     mov   direct, 3
        jmp   back
  n4:   mov   direct, 1
  back:       pop   ds
            pop dx
            pop cx
             pop        bx
             pop        ax
             iret
new_1cendp
cls     proc  near           ; подпрограмма очистки экрана
        push  cx
        push  ax
        push  si
```

```
            xor   si, si
            mov   ah, 7          ; атрибут
            mov   al,' '         ; символ, которым заполняется экран
            mov   cx, 2000       ; кол-во слов видеопамяти под весь  экран
      c:    mov   es:[si], ax          ; занести символ и атрибут в видеопа-
мять
            add   si, 2          ; перейти к следующему слову видеопамя-
ти
            loop  c
            pop   bx
            pop   ax
            pop   cx
            ret
      cls   endp
      delay proc  near           ; подпрограмма задержки
            push  cx
            mov   cx, 80000
      d:    nop
            loop  d
            pop   cx
            ret
      delay endp
      out_sym     proc  near           ; подпрограмма вывода символа с за-
данным атрибутом
            push  ax
            push  bx
            mov   al, sym              ; в al - символ
            mov   ah, atr        ; в ah - атрибут
            mov   bx, pos1       ; в bx – позиция видеоОЗУ для вывода
символа
            call  delay          ; задержка перед выводом символа
            mov   es:[bx], ax    ; помещение символа и атрибута в ви-
деоОЗУ
            pop   bx
            pop   ax
            ret
      out_sym     endp
      start:      mov   ax, data       ; основная программа
            mov   ds, ax           ; чтение вектора прерывания
            mov   ah, 35         ; функция получения вектора
```

```
        mov   al, 1Ch              ; номер вектора
        int   21h          ; сегмент – в es, смещение в bx
        mov   old_ip, bx
        mov   old_cs, es       ; установка вектора прерывания
        push  ds
        mov   dx, offset new_1c
        mov   ax, seg      new_1c
        mov   ds, ax
        mov   ah, 25h
        mov   al, 1Ch
        int   21h
        pop   ds
        mov   ax,0b800h    ; в es- сегментный адрес
        mov   es, ax       ; видеоОЗУ
        call  cls
l1:     cmp   exit, 0      ; проверка признака завершения программы
        jn    quit         ; если не «0» – конец программы
        cmp   direct, 1    ; вверх
        jz    l2
        cmp   direct, 2    ; влево
        jz    l3
        cmp   direct, 3    ; вправо
        jz    l4
        mov   ax, pos1     ; вниз
        add   ax, 160              ; считаем новую позицию для вывода
        cmp   ax,3999      ; правее правой нижней границы экрана ?
        jg    next         ; да – возвращаемся в цикл
        mov   pos1,ax      ; нет – записываем в pos1 новую позицию
        call  out_sym      ; и выводим символ в этой позиции
        jmp   l1           ; назад в бесконечный цикл
l2:     mov   ax, pos1
        sub   ax, 160
        jl    next
        mov   pos1, ax
        call  out_sym
        jmp   l1
l3:     mov   ax, pos1
        sub   ax2, 160
        jl    next
```

```
        mov   pos1, ax
        call  out_sym
        jmp   l1
l4:     mov   ax, pos1
        add   ax, 2
        cmp   ax, 3999
        jg    next
        mov   pos1, ax
        call  out_sym
next:     jmp     l1
quit:     call    cls
        push  ds
        mov   dx, old_ip          ; подготовка к восстановлению
        mov   ax,  old_cs
        mov   ds, ax              ; подготовка к восстановлению
        mov   ah, 25h                   ; функция установки вектора
        mov   al, 1Ch                   ; номер вектора
        int   21h
        pop   ds
        mov   ax, 4C00h
        int   21h
code    ends
        end   start
```

4 Задание для самостоятельного выполнения

1. Выводить последовательно цифры от 0 до 9 в одно место экрана. При вводе с клавиатуры какой-либо цифры менять темп вывода. Значение задержки между выводом очередного символа определять следующим образом: введенную цифру преобразовать в соответствующее ей двоичное число, умножить на 2^9, прибавить большее число в пределах 384. (Использовать вектор 1Ch)
2. Выводить в одно место экрана поочередно код пробела и код какого-либо символа. Задержка между выводом каждого символа определяется нажатием одной из цифровых клавиш. (Использовать вектор 1Ch.)
3. Выводить введенный символ до тех пор, пока не будет введен новый, меняя при этом его атрибут циклически от 1 до 15. (Ис-

пользовать вектор 1Ch)

4. Условие задачи из рассмотренного примера, но кроме того, темп вывода меняется в зависимости от нажатой клавиши.
5. В программе имеются два циклических счетчика от 0 до 23 и от 0 до 79, определяющие позицию символа на экране. По нажатию какой-либо клавиши на экран в положение, указанное счетчиками, выводится символ #.
6. В программе имеется циклический счетчик от 1 до 6. При нажатии на любую клавишу содержимое счетчика преобразуется в ASCII код и выводится в определенное место экрана, после чего счетчик продолжат считать.
7. Подсчитать за какое время процессор выполняет 1000 команд mov r1,r2; add r1, r2; inc r; mul r.
8. Очистить экран. Вывести несколько строк произвольного текста (атрибут 14). Перехватить прерывание печати экрана (int 5h). По этому прерыванию атрибут всех строк на экране должен циклически меняться от 1 до 15 (одно прерывание вызывает однократное изменение атрибута).
9. Реализовать программу, ежесекундно выводящую в правом углу экрана системное время «чч:мм:сс».
10. Вывести на экран несколько строк, содержащие лишь заглавные латинские буквы. Каждые 10 сек. латиница должна сменяться кириллицей и наоборот.
11. Написать программу, перехватывающую прерывания от системного таймера, поступающие каждые 18,2 с, периодически выводящую на экран какую-либо информацию.
12. Прочитать из КМОП-микросхемы текущее время, прибавить к нему заданный интервал (например 5 с) и установить будильник на полученное время. В программе обработки прерывания будильника вывести на экран символ.
13. Написать тестовую программу, периодически выводящую строку символов на экран (период 2-3 с). Предусмотреть завершение программы после вывода 5-6 строк. Включить в текст программы собственный обработчик прерываний по <Ctrl+C>. В качестве функций обработчика можно принять вывод на экран сообщения о перехвате <Ctrl+C>, изменение состояния счётчика цикла.
14. Установить транзитный обработчик прерываний от клавиатуры, включив его в текст прикладной программы. Обработчик должен перехватывать код "горячей клавиши" (например F10) с

целью выполнения некоторых аварийных действий.

15. В программе имеются два циклических счетчика от 0 до 23 и от 0 до 79, определяющие позицию символа на экране. По нажатию какой-либо клавиши на экран в положение, указанное счетчиками, выводится символ &.
16. В программе имеется циклический счетчик от 1 до 6. При нажатии на любую клавишу содержимое счетчика преобразуется в ASCII код и выводится в определенное место экрана, после чего счетчик продолжат считать.
17. Подсчитать за какое время процессор выполняет 2000 команд mov r1,r2; add r1, r2; inc r; mul r.
18. Очистить экран. Вывести несколько строк произвольного текста (атрибут 14). Перехватить прерывание печати экрана (int 5h). По этому прерыванию атрибут всех строк на экране должен циклически меняться от 1 до 15 (одно прерывание вызывает однократное изменение атрибута).
19. Реализовать программу, ежесекундно выводящую в левом углу экрана системное время «чч:мм:сс».
20. Вывести на экран несколько строк, содержащие лишь заглавные латинские буквы. Каждые 20 сек. латиница должна сменяться кириллицей и наоборот.

5 Контрольные вопросы

1. Чем процедура обработки прерывания отличается от обычной процедуры?
2. Что называется вектором прерываний?
3. Какое количество векторов прерываний допускается?
4. Как вычисляется адрес вектора прерываний?
5. Для чего используется прерывание 1Ch?
6. Каково назначение функции 35h прерывания DOS int 21h?
7. Каково назначение функции 25h прерывания DOS int 21h?

Лабораторная работа №5 Разработка резидентных программ обработки прерываний

1 Цель и порядок работы

Изучить вопросы, связанные с разработкой резидентных программ обработки прерываний.

2 Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя по вариантам;
- написать программу, ввести программу, отладить и решить ее на ЭВМ;
- оформить отчет.
-

3 Теоретические основы

При разработке реальных резидентных программ и обработчиков аппаратных прерываний возникает целый ряд проблем. На рисунке 5.1 изображена топологическая карта, где левая окружность обозначает обработчики аппаратных прерываний, а правая – резидентные программы. Тогда фигура в центре, образованная пересечением этих окружностей, относится к резидентным обработчикам, полумесяц слева – к транзитным обработчикам, а полумесяц справа – к резидентным программам, не обрабатывающим аппаратные прерывания и, следовательно, активизируемым синхронно. Если обработчик аппаратного прерывания обслуживает не стандартную для компьютера аппаратуру (например, измерительную или управляющую), он должен представлять собой законченную программу, включающую в себя все программные элементы, необходимые для обеспечения работоспособности обслуживаемого устройства. Если, однако, прикладной обработчик прерываний пишется для стандартной аппаратуры (клавиатуры, таймера, КМОП-микросхемы, дисков и проч.), целесообразно в программу прикладного обработчика включить лишь необходимые дополнения к системному алгоритму обслуживания устройства. В этом случае прикладной обработчик надо сцепить с системным, организовав, в зависимости от конкретных требований, прикладную обработку либо до, либо после системной (а иногда и до, и после). Для обработчиков аппаратных прерываний характерен асинхронный способ активизации: аппаратное прерывание может возникнуть в любой момент времени, и состояние программы, в частности, содержимое регистров на момент прерывания никогда не бывает известно. Про-

цессор, выполняя процедуру прерывания, сохраняет только регистры CS:IP и флаги; все остальные регистры, если они используются в программе обработчика, необходимо при входе в обработчик сохранить, а перед выходом восстановить. Обычно сохранение осуществляется в стеке. Если обработчик является транзитным, т.е. включен в качестве составного элемента в текущую транзитную программу, используется, естественно, ее стек, общий для всей программы и для обработчика. Сложности возрастают, если обработчик является резидентным.

В силу асинхронного способа активизации обработчик аппаратного прерывания может получить управление в тот момент, когда в основной программе выполняется запрошенная ею функция DOS или BIOS. И DOS, и BIOS состоят из нереентерабельных программ; нельзя, прервав выполнение какой-то функции DOS, вызвать ту же или другую функцию – это неминуемо приведет к разрушению системы. Поэтому без принятия специальных мер в обработчике аппаратного прерывания недопустимо обращаться к функциям DOS или BIOS. Отмеченная проблема является важнейшей из всех перечисленных.

Ситуация усугубляется, если обработчик прерываний является резидентным, т.е. не входит в состав какой-либо программы. В этом случае он существует независимо от транзитных программ и может получить управление, прервав выполнение произвольной транзитной (или даже другой резидентной) программы. При переключении на программу обработчика текущим стеком остается стек прерванной программы. Активное использование стека в обработчике может привести к переполнению стека и разрушению текущей программы. В таком обработчике необходимо предусмотреть собственный стек, а также процедуры переключения стека сразу после входа в обработчик и перед выходом из него.

Рисунок 5.1 – Проблемы, возникающие при разработке обработчиков аппаратных прерываний и резидентных программ.

Даже если преодолеть трудности, возникающие из-за нереентерабельности DOS, в резидентом обработчике аппаратных прерываний нельзя обращаться к услугам файловой системы. Это связано с тем, что при открытии файла DOS использует для связи с системной областью файлов (SFT) таблицу файлов задания, расположенную в PSP текущей программы. Поскольку при переходе на программу обработчика текущей для системы остается прерванная программа, файлы, открываемые в обработчике, будут использовать ее PSP (или, например, PSP программы COMMAND.COM, если в памяти нет запущенных транзитных программ). Ясно, что завершение текущей транзитной программы и запуск вместо нее другой разрушит цепочку связи с SFT и лишит обработчик возможности работать с нужным ему файлом. Таким образом, в обработчике, обращающемся к файловым функциям DOS, необходимо выполнять переключение текущей программы, для чего в DOS предусмотрены соответствующие средства.

Многие резидентные программы должны активизироваться нажатием некоторой клавиши или комбинации клавиш. Обычно такую клавишу

называют "горячей". Таким образом, программа сама по себе не имеющая отношения к аппаратным прерываниям, должна, тем не менее, перехватывать прерывания от клавиатуры и вылавливать из потока нажимаемых клавиш свои горячие клавиши, чтобы активизироваться или, наоборот, "свернуться" и перейти в пассивное состояние. Такого рода программы обычно широко используют системные средства для чтения файлов, вывода на экран и ввода с клавиатуры; всеми этими средствами нельзя воспользоваться, находясь "внутри" программы обработки аппаратного прерывания от клавиатуры.

В любой резидентной программе должны предусматриваться средства проверки на повторную установку. Такую проверку можно осуществить разными методами, однако наиболее распространенным является использование мультиплексорного прерывания 2Fh.

Другой важный вопрос – выгрузка ненужной более резидентной программы из памяти. Резидентные программы, совсем не связанные с аппаратными прерываниями, используются относительно редко. В таких программах должны быть предусмотрены средства их синхронной активизации и передачи (в одну или обе стороны) параметров.

– 3.1 Использование средств BIOS в обработчиках аппаратных прерываний

И DOS, и BIOS являются нереентерабельными программами, однако их нереентерабельность вызывается разными причинами и проявляется по-разному. Рассмотрим, например, прерывание 13h BIOS, закрепленное за программами управления дисками. Программный запрос на выполнение функции 02h прерывания 13h должен привести к чтению в программу указанной группы секторов жесткого или гибкого дисков. Этот запрос реализуется программами BIOS с помощью целого ряда команд, направляемых в контроллер диска: инициализации, поиска требуемой дорожки, чтения состояния контроллера, собственно чтения. При этом многие команды выполняются путем многократного обращения к контроллеру: сначала в него посылается код команды, затем – требуемые для выполнения этой команды параметры.

Если в обработчике аппаратного прерывания, активизированного в момент выполнения текущей программой функции дискового прерывания 13h, в свою очередь, вызывается какая-то функция прерывания 13h, то произойдет неминуемое нарушение работы программы, так как после выполнения части команд первого запроса (например, поиска дорожки) начнут выполняться команды второго запроса (например, тоже поиск, но другой дорожки).

В то же время прерывание дисковой функции BIOS само по себе не страшно. Если в обработчике при этом вызываются какие-то другие прерывания BIOS, например int 10h для управления экраном или int 16h для ввода с клавиатуры, нарушения работы не произойдет. Таким образом, прерывания BIOS нереентерабельны только по отношению к самим себе – нельзя, прервав выполнение int 13h, вызвать в обработчике то же прерывание int 13h или, прервав работу с экраном на уровне BIOS через прерывание 10h, вызвать в обработчике функцию того же прерывания int 10h.

Таким образом, вызов функций BIOS возможен в обработчике только в те моменты времени, когда прерываемая программа не выполняет функции того прерывания, к которому мы хотим обратиться в обработчике. Для того, что бы программа обработчика знала, не выполняется ли сейчас интересующее нас прерывание, в состав резидентного обработчика включаются секции перехвата всех прерываний BIOS, которые предполагается использовать в обработчике. Входные адреса этих секций заносятся в соответствующие векторы прерываний, а сами программы перехватчиков строятся так, что часть своей работы они выполняют перед системным обработчиком данного прерывания, а часть – после. Так, перехватчик прерывания BIOS, представленный в примере 5.1, перед передачей управления BIOS устанавливает флаг "занятости" данного прерывания, а после возврата из BIOS сбрасывает этот флаг. Основная же программа обработчика, перед тем, как вызывать функции BIOS, анализирует состояние этого флага и обращается к BIOS только в том случае, если флаг сброшен, т.е. данное прерывание BIOS "свободно".

Пример 5.1 – Процедура перехватчика прерывания BIOS

```
capt_13h proc       ;Процедура перехватчика int 13h
  flag_13h    db 0 ;Флаг занятости прерывания 13h
  old_13h     dd 0 ;Ячейка для хранения исходного
new 13h:            ;Точка входа в перехватчик
  inc  cs:flag_13h  ;Текущая программа вызвала int 13h, установим флаг
занятости
  pushf             ;Вызовем системный обработчик
  call cs:old_13h   ;прерывания 13h с возвратом
  dec  cs:flag_13h  ;Системная обработка int 13h закончилась, сбросим
флаг занятости
  iret              ;Назад в текущую программу
capt_13h end        ;Конец процедуры перехватчика int
```

Перехватчики прерываний BIOS, включенные в программу обработчика аппаратного прерывания, позволяют выяснить, свободна ли в настоящий момент BIOS. Если данное прерывание BIOS сейчас не выполняется, обработчик может свободно вызывать любые функции этого прерывания. Если BIOS занята, обработчик должен установить флаг незавершенности и вернуть управление в прерванную программу (фактически – в программу BIOS). Теперь для того, чтобы обработчик завершил свою работу, его надо вызывать повторно до тех пор, пока он не обнаружит, что BIOS свободна. Это даст возможность обработчику вызвать требуемую функцию BIOS и, сбросив флаг незавершенности, окончательно вернуть управление в прерванную программу. Для таких повторных вызовов обычно используются прерывания от таймера, поступающие 18,2 раза в секунду.

– 3.2 Использование средств DOS в обработчиках аппаратных прерываний

Функции DOS, как и прерывания BIOS, нереентерабельны, однако нереентерабельность DOS носит иной характер. Среди системных областей DOS имеется весьма важная структура, которая называется областью текущих данных (Swappable Data Area (SDA)). Это довольно большая область объемом около 2 Кбайт, адрес которой можно получить с помощью недокументированной функции DOS 5B06h. Функция возвращает в регистрах DS:SI адрес SDA, а в регистре CX ее размер. Наиболее интересные для прикладного программиста поля SDA представлены в таблице 5.1.

Флаг критической ошибки ErrorMode устанавливается DOS, если зафиксировано состояние критической ошибки, которое может возникнуть по разным причинам, в большинстве своем связанным с дисковыми операциями: попытка записи на защищенный или отсутствующий диск, на диске не найден требуемый сектор, ошибка в контрольной сумме данных в секторе и т.д. Обнаружив такую ситуацию, DOS устанавливает флаг ErrorMode, записывая в этот байт SDA 1, и выполняет команду int 24h. В этом векторе хранится адрес системного обработчика критической ошибки, который выводит на экран аварийное сообщение и вводит ответную команду пользователя.

Таблица 5.1 – Некоторые поля области текущих данных

Смещение	Число байтов	Назначение
00h	1	Флаг критической ошибки (флаг ErrorMode)
01h	1	Флаг занятости DOS (флаг InDOS)
02h	1	Дисковод, на котором зафиксирована последняя кри-
03h	1	тическая ошибка, или FFh
04h	2	Устройство, на котором зафиксирована последняя
06h	1	ошибка
07h	1	Код расширенной ошибки для последней ошибки
08h	4	Предлагаемое действие для исправления последней
0Ch	4	ошибки
10h	2	Класс последней ошибки
12h	2	Содержимое ES:DI при последней ошибке
14h	2	Адрес текущей области дисковой передачи DTA
16h	1	Сегментный адрес текущего PSP (ID текущей про-
17h	1	граммы)
30h	1	Ячейка для хранения SP при вызове int 23h
31h	1	Код возврата завершившегося процесса
32h	2	Текущий дисковод
34h	2	Расширенный флаг BREAK
36h	1	День месяца
264h	4	Месяц
2D0h	4	Год-1980
336h	330	Номер дня от 01.01.1960
480h	384	День недели (0=воскресенье, 1=понедельник,...
600h	384	Прошлый кадр стека
		Позапрошлый кадр стека
		Вспомогательный стек (для функций 0lh...0Ch при наличии критической ошибки)
		Дисковый стек (для функций 00h, 0Dh...6Ch)
		Стек ввода-вывода (для функций 0lh...0Ch)

Флаг занятости DOS, обычно называемый флагом InDOS ("внутри DOS"), устанавливается диспетчером DOS сразу после анализа им номера вызванной функции DOS, и сбрасывается перед возвратом из DOS в прикладную программу. Таким образом, значение этого флага, равное 1, говорит о том, что выполняется программа DOS. Функции DOS в прикладной программе можно вызывать, только флаг InDOS сброшен.

Сегментный адрес текущего PSP (смещение 10h), является важнейшей системной переменной. В этом поле записывается адрес PSP (идентификатор ID) той программы, которую DOS считает текущей. Именно эта программа будет завершена, если выдать запрос на выполнение функции 4Ch (независимо от того, какая программа выдала этот запрос). Далее, при создании или открытии файлов используется таблица файлов задания JFT текущей программы, опять же независимо от того, какая программа реально выдала запрос на работу с файлом. Понятие текущей программы приобретает особую важность при разработке обработчиков аппаратных прерываний. Действительно, при вызове обработчика изменяются только значения CS:IP, текущей же остается прерванная программа. Поэтому, например, дескрипторы файлов, открытых в обработчике, находятся не в PSP обработчика, а в PSP прерванной программы, и при завершении этой программы работа обработчика (если он, будучи резидентным, остался в памяти) будет нарушена.

Адрес текущей области дисковой передачи (Disk Transfer Area (DTA)) важен в тех случаях, когда в обработчике аппаратного прерывания вызываются функции DOS, использующие эту область. Раньше, когда для работы с файлами использовались блоки управления файлами (FCB), с помощью DTA осуществлялись все файловые операции; в настоящее время эта область нужна только при выполнении функций поиска файлов.

В области текущих данных находятся и все три системных стека – стек ввода-вывода, который используется DOS при выполнении функций ввода-вывода из диапазона 0lh...0Ch, дисковый стек для всех остальных функций DOS и вспомогательный стек, на который DOS переключается в том случае, если в процессе обработки критической ошибки возникла необходимость обратиться к функциям ввода-вывода. Наличие внутренних стеков повышает надежность работы DOS, так как устраняется возможность переполнения стека прикладной программы при выполнении запросов к DOS, которые в своем большинстве активно используют стек для вызова системных подпрограмм, сохранения текущих значений и передачи параметров.

Диспетчер DOS, получив управление в результате выполнения команды int 21h, совершает целый ряд операций, из которых, прежде всего надо отметить следующие:

– сохранение регистров задачи на стеке задачи;
– инкремент InDOS, который при первом (не вложенном) вызове DOS становится равен 1;
– перенос прошлого кадр стека (SS:SP) в ячейку для позапрошлого кадра;
– сохраняет в ячейке для прошлого кадра стека SS:SP прерванной задачи;
– заносит в SS:SP кадр одного из системных стеков в соответствии с вызванной функцией;
– вызывает по номеру функции требуемую программу DOS.

После завершения вызванной функции диспетчер выполняет описанные шаги в обратном порядке: восстанавливает кадр стека задачи и кадр прошлого стека, декрементирует флаг InDOS, который опять становится равен 0, восстанавливает из стека задачи ее регистры и выполняет команду iret возврата в прерванную задачу.

Нереентерабельность DOS связана с тем, что при выполнении большинства функций используется один и тот же системный стек. Если выполнение какой-либо функции DOS будет прервано аппаратным прерыванием, и в обработчике прерывания будет вызвана функция DOS из той же группы, т.е. использующая тот же стек, то диспетчер загрузит в SS:SP в точности те же значения, что и при первом вызове. В результате вложенный вызов будет затирать те данные, которые были сохранены в стеке первым вызовом. Однако при вызове функции из другой группы ничего страшного не произойдет, так как функции ввода-вывода и "дисковые" работают на разных системных стеках. При этом наличие в SDA двух ячеек для хранения не только прошлого, ни и позапрошлого кадров стека; позволяет правильно обрабатывать один вложенный вызов DOS.

Фактический адрес флага занятости DOS можно получить с помощью документированной функции DOS 34h. Она возвращает двухсловный адрес флага InDOS в регистрах ES:BX. Получив и сохранив этот адрес на этапе инициализации, в самом обработчике перед вызовом функций DOS следует выполнить проверку флага занятости (адрес флага сохранен в ячейке in_dos).

Пример 5.2 – Процедура перехватчика прерывания DOS

```
in_dos        dd 0    ; Адрес флага InDOS
task_req      db 0    ; Флаг требования запуска task
```

```
...
dos_busy:
  les  bx, cs:in_dos   ; Получили адрес флага InDOS
  cmp  es:[bx], 0      ; DOS свободна?
  jne  wait_dos        ; Нет, придется ждать
  call task            ; Да, можно вызывать функции DOS
  iret                 ; Завершим обработчик
wait_dos:
  inc cs:task_req      ; Установим флаг требования запуска
  iret                 ; Завершим обработчик
```

В этом фрагменте предполагается, что процедура task как раз и содержит вызовы DOS. Если флаг InDOS сброшен, вызывается эта процедура и после ее завершения завершается и весь обработчик. Если же флаг InDOS установлен, обработчик устанавливает флаг task_req, который говорит о том, что обработчик из-за занятости DOS "недовыполнил" свои функции, и процедура task требует своего запуска. Управление возвращается в прерванную задачу (фактически – в прерванную функцию DOS) и DOS имеет возможность завершить свою работу.

Для иого, чтобы теперь запустить процедуру task, нужен дополнительный "активизатор" нашего обработчика. В качестве такого активизатора естественно воспользоваться прерываниями от таймера. Таким образом, обработчик, помимо своей основной точки входа, должен иметь еще точку входа для обработки прерываний от системного таймера:

Пример 5.3 – Активация отложенного вызова процедуры

```
in_dos         dd 0    ; Адрес флага InDOS
task_req       db 0    ; Флаг требования запуска task
old_08h        dd 0    ; Ячейка для хранения исходного содержимого
вектора 08h
       ...
new_08h:
  ; Передадим управление в системный обработчик прерываний от
таймера
  pushf
  call cs:old_08h
  ; После работы системного обработчика получаем управление
  cmp  cs:task_req, 1  ; Процедура task требует запуска?
  jne  out_08h         ; Нет, можно завершить обработку
  les  bx, cs:in_dos   ; Да, снова выясним,
```

```
  cmp  es:[bx], 0      ; свободна ли D0S?
  jne  out_08h         ; Нет, придется опять ждать
  dec  cs:task_req     ; Да, сбросим флаг требования запуска
  call task            ; и вызовем task
out_08h:
  iret                 ;Завершим обработчик
```

Проверка занятости DOS по состоянию флага InDOS необходима, но не достаточна в тех ситуациях, когда возможно возникновение критической ошибки. Дело в том, что, обнаружив критическую ошибку, DOS выполняет декремент флага InDOS и инкремент флага критической ошибки ErrorMode, находящегося, как и флаг InDOS, в области текущих данных. После этого DOS с помощью прерывания int 24h вызывает обработчик критической ошибки, который выводит на экран запрос, соответствующий сложившейся ситуации и ждет указаний пользователя, например:

Not ready reading drive A

Abort, Retry, Fai1?

И вывод на экран, и ввод с клавиатуры осуществляются с помощью функций DOS из диапазона 0lh...0Ch, причем они вызываются "изнутри" той функции DOS, в процессе выполнения которой возникла критическая ошибка. Диспетчер DOS перед переключением стека проверяет состояние флага критической ошибки и если этот флаг установлен, делает текущим не стек ввода-вывода, как обычно, а вспомогательный стек. В результате такой вложенный вызов DOS не приводит к разрушению системы. Таким образом, в обработчике аппаратного прерывания перед вызовом какой-либо функции DOS следует анализировать состояние не только флага InDOS, но и флага ErrorMode. Если хотя бы один из них не равен 0, вызывать DOS нельзя.

В версиях DOS отсутствуют документированные средства для определения адреса флага критической ошибки. Известно, однако, что в DOS начиная с версии 3.10 он расположен в первом байте области текущих данных перед флагом InDOS, и для получения его адреса можно воспользоваться недокументированной функцией 5D06h. Поскольку сама эта функция не является реентерабельной, определение адреса следует выполнить на этапе инициализации обработчика (вместе с определением адреса флага InDOS).

С учетом сказанного, приведенный выше фрагмент проверки занятости DOS будет выглядеть следующим образом (адрес флага критической ошибки помещен в ячейку crit_err).

Пример 5.4 – Вызов процедуры из обработчика прерывания с учетом флага ErrorMode

```
in_dos        dd     0            ; Адрес флага InDOS
crit_err      dd     0            ; Адрес флага ErrorMode
task_req      db     0            ; Флаг требования запуска task
          ...
dos_busy:
      les    bx: cs:in_dos   ; Получили адрес флага InDOS
      lds    si, cs:crit_err ; Получили адрес флага ErrorMode
      cmp    es:[bx], 0      ; Флаг InDOS сброшен?
      jne    wait_dos        ; Нет, придется ждать
      cmp    ds:[si], 0      ; Флаг ErrorHode сброшен?
      jne    wait_dos        ; Нет, придется ждать
      call   task            ; Да, оба флага сброшены, можно вызывать
функции DOS
      iret                   ; Завершим обработчик
wait_dos:
      inc    cs:task_req     ; Установим флаг требований запуска
      iret                   ; Завершим обработчик
```

Аналогичную коррекцию следует ввести и в процедуру обработки прерываний от таймера.

Описанная методика годится далеко не для всех программ. Если текущая программа ждет ввода с клавиатуры, она не вы ходит из соответствующей функции DOS и флаг занятости DOS непрерывно установлен. То же получается, если текущей программы вообще нет, так как в этом случае активным является командный процессор COMMAND.COM, который ждет ввода с клавиатуры очередной команды пользователя (функцией DOS 0Ah). В такой ситуации наш обработчик никогда не сможет вызвать требуемую функцию DOS. Для преодоления этого затруднения в DOS включено специальное прерывание int 28h, вызываемое функциями ввода с клавиатуры. Выполнение, на пример, функции 0lh, в самых общих чертах выглядит так, как показано на рисунке 5.2.

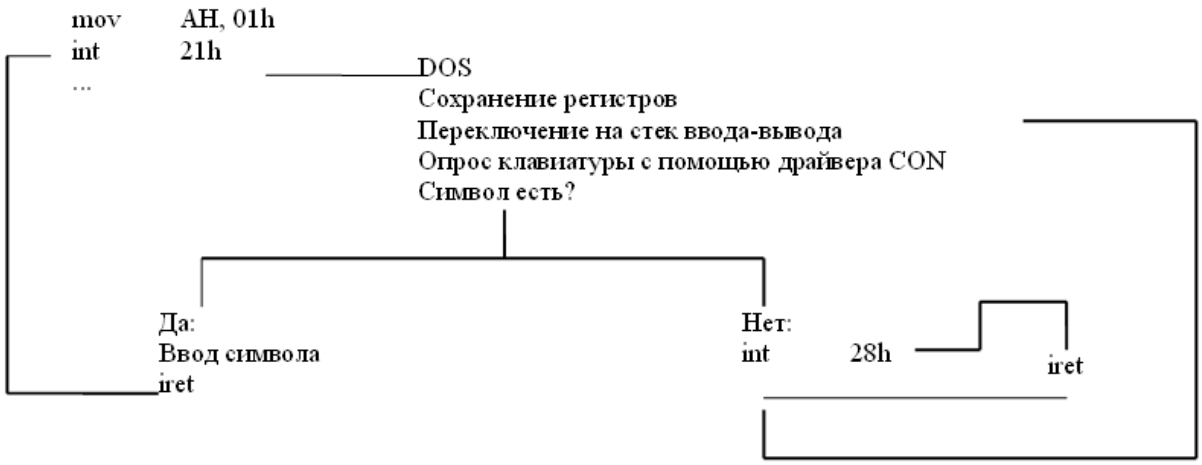

Рисунок 5.2 – Системный алгоритм выполнения функции ввода с клавиатуры.

Системный обработчик прерывания 28h содержит единственную команду iret, поэтому вызов int 28h при выполнении функции ввода-вывода никак не нарушает ход программы. Однако прикладная программа может поместить в вектор 28h адрес своей процедуры обработки этого прерывания. Поскольку прерывание 28h возникает только при выполнении функций, использующих стек ввода-вывода, в обработчике прерывания 28h допустим вызов любых функций DOS из диапазона 0Dh...6Ch. Надо только иметь в виду, что при входе в обработчик 28h все регистры заполнены системными значениями, и для выполнения прикладных функций их надо соответствующим образом инициализировать.

Таким образом, прикладной обработчик аппаратных прерываний, в котором вызываются функции DOS, должен включать, помимо активизатора по прерываниям от таймера (через вектор 08h) еще и активизатор по прерыванию 28h со схожим алгоритмом.

Пример 5.5 – Активация отложенного вызова процедуры в обработчике int 28h

```
in_dos        dd     0      ; Адрес флага InDOS
crit_err      dd     0      ; Адрес Флага ErrorHode
task_req      db     0      ; Флаг требования запуска task
old_28h       dd     0      ; Ячейка для хранения исходного
          ...               ; содержимого вектора 28h
new_28h:cmp   cs:task_req, 1
      jne     out_28h
```

```
        les     bx, cs:in_dos
        lds     si, cs:crit_err
        cmp     ds:[si], 0      ; Флаг ErrorMode сброшен?
        jne     out_28h         ; Нет, придется ждать
        cmp     es:[bx], 1      ; Флаг InDOS не больше 1?
        ja      out_28h         ; Больше, вложенный вызов DOS, вызов DOS
запрещен
        dec     cs:task_req     ; Сбросим флаг требования запуска
        call    task            ; и вызовем task
out_28h:jmp     cs:old_28h      ; Завершим обработчик
```

Прикладной обработчик прерывания 28h (включенный в состав обработчика аппаратного прерывания) прежде всего, проверяет, установлен ли флаг запроса запуска задачи. Если этот флаг сброшен, надо просто вернуться в прерванную функцию DOS для чего, в простейшем случае, можно выполнить команду iret. Если, однако, в системе имеется несколько обработчиков прерывания 28h, такое завершение лишит эти обработчики возможности фиксировать прерывание 28h. Поэтому наш обработчик лучше завершить командой jmp cs:old_28h для передачи управления на тот обработчик, адрес которого мы вытеснили из вектора 28h.

Если флаг запроса запуска задачи установлен, сначала проверяется состояние флага критической ошибки, и, если он сброшен, то и состояние флага занятости DOS. Этот флаг должен быть равен 1 (ведь сейчас выполняется функция ввода с клавиатуры). Если значение флага 1, это свидетельствует о том, что "внутри" прерывания 28h уже вызвана функция DOS, и вызывать функции DOS, увеличивая тем самым уровень вложенности DOS, нельзя. Бывают ситуации (например, при использовании программы Norton Commander), когда в обработчике int 28h флаг InDOS оказывается равен не 1, а 0. Проверка значения флага на условие >1 учитывает и эту возможность.

Наконец, после успешного прохождения всех проверок вызывается процедура task с вызовами DOS, после завершения которой, управление передается предыдущему обработчику int 28h .

Вызов int 28h выполняется функцией DOS на стеке ввода-вывода. Поэтому в обработчике прерывания 28h можно вызывать только функции "дисковой" группы. Также, недопустим вызов файловых функций с указанием стандартных дескрипторов клавиатуры и экрана (0...2). Проблема ввода-вывода решается, открытием терминала, как файла, и использование полученного дескриптора. Другой способ обойти ограничения – воспользоваться функциями BIOS.

Пример 5.6 – Ввод-вывод в обработчике int 28h

```
fname       db 'CON', 0     ; Имя консоли в формате файловых функций
handle      dw 0            ; Ячейка для дескриптора консоли
        ...
    mov   ah, 30h       ; Функция открытия файла
    mov   al, 2         ; Режим ввода и вывод
    mov   dx, offset fname; Адрес имени файла
    int   21h
    mov   handle, ax    ; Сохраним дескриптор консоли
; Введём сообщение с клавиатуры
    mov   ah, 3Fh       ; Файловая функция ввода
    mov   bx, handle    ; Дескриптор консоли
    mov   cx, 80        ; Предельное число вводимых символов
    mov   dx, offset buf  ; Адрес буфера
; Выведем сообщение
    mov   ah, 40h       ; Файловая функция вывода
    mov   bx, handle    ; Дескриптор консоли
    mov   cx, mes_len   ; Длина сообщения
    mov   dx, offset mes  ; Адрес сообщения
    int   21h
```

3.3 Асинхронная активизация резидентных программ командами с клавиатуры

Для управления резидентной программой командами, подаваемыми с клавиатуры, в частности, для активизации программы, в ее состав необходимо включить обработчик прерываний от клавиатуры (прерывание 09h). Выполнять резидентную программу, находясь внутри такого обработчика, можно только в том случае, если в программе не используются вызовы DOS или BIOS. Поэтому обычно в обработчике прерывания от клавиатуры выполняется анализ введенной команды и установка флага запроса запуска задачи, если команда соответствует запросу на активизацию. Сама же активизация задачи осуществляется в обработчике прерывания 28h, 1Ch или 08h.

Пример 5.7 – Структура секции асинхронного запуска задачи командой с клавиатуры

```
old_28h  dd 0   ; Старое содержимое вектора 28h
old_09h  dd 0   ; Старое содержимое вектора 09h
task_req db 0   ; Флаг запроса запуска задачи
```

```
        ...
new_09h proc
        push    ax              ; Сохраним AX
        in      al, 60h         ; Введем скэн-код нажатой клавиши
        cmp     al, 4Eh         ; Нажат серый плюс?
        je      plus            ; Да, на установку флага
        pop     ax              ; Восстановим AX
        jmp     cs:old_09h      ; В обработчик прерывания 09h
plus:
        inc     cs:task_req     ; Установим флаг запроса запуска
        pop     ax              ; Восстановим AX
        jmp     cs:old_09h      ; В обработчик прерывания 09h
new_09h endp

new_28h proc
        cmp     cs:task_req, 1  ; Задача требует актиеизации?
        je      pop_up          ; Да
        jmp     cs:old_28h      ; Нет, в обработчик прерывания 28h
pop_up:
; Проверка флагов критической ошибки
; Сброс флага запроса запуска задачи
; Запуск задачи
iret
new_28h endp
```

3.4 Работа с файлами в резидентном обработчике аппаратных прерываний

Работа с файлами в резидентном обработчике аппаратных прерываний осложняется тем, что при переходе в обработчик, текущей остается прерванная программа, и дескрипторы открываемых файлов создаются системой в таблице файлов задания ее PSP. Для того, чтобы дескрипторы файлов, открываемых в обработчике, полностью принадлежали самому обработчику, необходимо при входе в обработчик объявить его для DOS текущей программой. Для манипуляций с идентификаторами программ (т.е. сегментными адресами, их PSP) в DOS предусмотрены функции 51h (Получить PSP) и 50h. (Установить PSP). Обе функции не используют системные стеки и являются реентерабельными, поэтому их можно безопасно вызывать в обработчиках аппаратных прерываний.

Пример 5.8 – Резидентный обработчик, работающий с файлами через переключение PSP

```
entry:  ; Сохранение регистров прерванной программы
        ...
; Сделаем обработчик текущей программой
        mov     ah, 51h ; Функция получения текущего PSP
        int     21h     ; BX=ID текущей (т.е. прерванной) программы
        push    bx      ; Сохраним ID текущей программы
        mov     ah, 50h ; Функция установки текущего PSP
        mov     bx, cs  ; В .COM-программе CS=ID программы
        int     21h

; Теперь текущей считается программа обработчика. При открытии
файлов
; дескрипторы будут создаваться в JFT обработчика
;       ...
; Работа с файлами обработчика (естественно, если DOS свободна )
;       ...
; Сделаем прерванную программу текущей
        pop     bx      ; Восстановим ID прерванной программы
        mov     ah, 50h ; Функция установки текущей PSP
        int     21h
; Восстановим регистры прерванной программы
        iret
```

В случаях, когда в обработчике вызываются файловые функции поиска файлов 4Eh и 4Fh, использующие область дисковой передачи DTA, переключения ID программы недостаточно. При загрузке программы, DOS записывает в SDA не только адрес PSP этой программы, т. е, ее ID, но и двухсловный адрес текущей DTA. Поэтому в обработчике следует при входе сохранить адрес текущей DTA и установить в качестве текущей DTA обработчика, а при выходе восстановить DTA прерванной программы. Для получения и установки DTA предусмотрены функции 2Fh и 1Ah. Они не реентерабельны, поэтому при их использовании необходимо убедиться, что DOS свободна.

Пример 5.9 – Структура обработчика файловых функций, использующих DTA

```
; Переключение DTA перед использованием файловых функций
        mov  ah, 2Fh    ; Функция получения текущей DTA
```

```
        int  21h        ; ES:BX= адрес DTA
        push cs         ; Настроим DS на PSP программы
        pop  ds         ; (если это не сделано ранее )
        push es         ; Сохраним сегментный адрес DTA
        push bx         ; Сохраним смещение DTA
        mov  ah, 1Ah    ; Функция установки текущей DTA
        mov  dx, 80h    ; Пусть DTA обработчика будет на
        int  21h    ; своем исходном месте – начиная со смещения 80h в PSP
; Теперь можно вызывать файловые функции, использующие DTA
...
; Переключение DTA перед завершением обработчика
        mov    ah, 1Ah        ; Функция установки текущей DTA
        pop    dx             ; Восстановим сегментный адрес DTA
        pop    ds             ; Восстановим смещение DTA
        int    21h
```

Как известно, DTA программы по умолчанию находится в ее PSP, начиная со смещения 80h (то же поле используется и для копирования параметров командной строки). Функция установки DTA 1Ah требует загрузки двухсловного адреса ЭТА в регистры DS:DX. При вызове обработчика прерывания, выполненного в формате COM, в CS находится сегментный адрес PSP. Командами DS также настраивается на сегментный адрес PSP. Теперь достаточно занести в DX смещение 80h и можно вызывать функцию установки DTA.

При использовании в резидентном обработчике функций DOS не исключено возникновение ошибок. Код ошибки возвращается системой в регистре AX, а расширенная информация об ошибке записывается в предназначенные для этого поля SDA. Функция DOS 59h позволяет получить из SDA расширенную информацию об ошибке и программно проанализировать ее. Если прерывание, активизировавшее наш обработчик, возникло после выполнения в прерванной программе функции DOS, завершившейся с ошибкой, и, кроме того, ошибка возникла в самом обработчике, то информация об ошибке прерванной программы в SDA будет затерта новой информацией, поступившей в SDA результате ошибки в обработчике. Если прерванная программа (после возвращения в нее) вызовет функцию DOS 59h для получения расширенной информации об ошибке, она получит информацию, относящуюся к ошибке в обработчике. Поэтому в резидентном обработчике аппаратных прерываний следует перед вызовом функций DOS получить из SDA расширенную информа-

цию об ошибке (с помощью функции DOS 59h), а перед выходом из обработчика – восстановить ее с помощью функции 5D, подфункции 0Ah.

3.5 Выгрузка из памяти резидентных программ

При разработке резидентных программ обычно предусматривают средства выгрузки их из памяти. Следует заметить, что в DOS вообще отсутствуют выгрузки резидентных программ. Единственный предусмотренный для этого механизм – перезагрузка компьютера. Практически, однако, большинство резидентных программных продуктов имеют встроенные средства выгрузки. Обычно выгрузка резидентной программы осуществляется соответствующей командой, подаваемой с клавиатуры и воспринимаемой резидентной программой. Для этого резидентная программа должна перехватывать прерывания, поступающие с клавиатуры, и отлавливать команды выгрузки. Другой, более простой способ, заключается в запуске некоторой программы, которая с помощью, например, мультиплексного прерывания 2Fh передает резидентной программе команду выгрузки. Чаще всего в качестве выгружающей используют саму резидентную программу, точнее, ее вторую копию, которая, будучи запущенной в определенном режиме, не остается в памяти резидентной, а выгружает из памяти свою первую копию.

Выгрузку резидентной программы из памяти можно осуществить разными способами. Наиболее простой – освободить блоки памяти, занимаемые программой (собственно программой и ее окружением) с помощью функции DOS 49h. Другой, более сложный – использовать в выгружающей программе функцию завершении 4Ch, заставив ее завершить не саму выгружающую, а резидентную программу, и, после этого, вернуть управление в выгружающую. В любом случае перед освобождением памяти необходимо восстановить все векторы прерываний, перехваченные резидентной программой. Следует подчеркнуть, что восстановление векторов представляет в общем случае значительную и иногда даже неразрешимую проблему. Во-первых, старое содержимое вектора, которое хранится где-то в полях данных резидентной программы, невозможно извлечь "снаружи", из другой программы, так как нет никаких способов определить, где именно его спрятала резидентная программа в процессе инициализации. Поэтому выгрузку резидентной программы легче осуществить из нее самой, чем из другой программы. Во-вторых, даже если выгрузку осуществляет сама резидентная программа, она может правильно восстановить старое содержимое вектора лишь в том случае, если этот вектор не был позже перехвачен другой резидентной программой. Если же это произошло, в таблице векторов находится уже адрес не вы-

гружаемой, а следующей резидентной программы, и если восстановить старое содержимое вектора, эта следующая программа "повиснет", лишившись средств своего запуска. Поэтому надежно можно выгрузить только последнюю из загруженных резидентных программ.

Для того чтобы удалить из памяти резидентную программу, надо в какой-то программе вызвать прерывание 2Fh с функцией, присвоенной этой программе, и подфункцией 01h. Проще всего создать для этого специальную выгружающую программу. Однако такая методика выгрузки резидентной программы довольно неуклюжа. Для каждой резидентной программы нам придется создавать свою выгружающую программу. Гораздо изящнее использовать в качестве выгружающей саму резидентную программу. Обычно это достигается путем предварительного анализа ключей командной строки в секции инициализации программы, чтобы, например, команда resident.com загружала программу в память, оставляя ее резидентной, а команда resident.com /u дезактивировала программу и выгружала ее из памяти.

Как известно, если программа запускается с клавиатуры с указанием каких-то параметров (имен файлов, ключей, определяющих режим работы программы и проч.), то DOS, загрузив программу в память, помещает все символы, введенные после имени программы (так называемый хвост команды) в префикс программного сегмента программы, начиная с относительного адреса 80h. Хвост команды помещается в PSP в следующем формате: в байт по адресу 80h DOS заносит число символов в хвосте команды (включая пробел, разделяющий на командной строке саму команду и ее хвост). Далее (начиная с байта по адресу 81h) следуют все символы, введенные с клавиатуры до нажатия клавиши <Enter>. Завершается хвост кодом возврата каретки (13).

Таким образом, если программа с именем resident.com была вызвана командой resident.com /u, то в PSP будет записана следующая информация: 4, ' /u', 13. Программа может проанализировать параметры ее запуска в PSP и, обнаружив там необходимый, вызвать предусмотренную заранее функцию выгрузки мультиплексорного прерывания 2Fh.

Рассмотрим подробней один из способов завершения работы резидентной программы – выполнение в управляющей (выгружающей) программе функции DOS 4Ch завершения текущего процесса. Функция DOS 4Ch всегда вызывается в конце программы, чтобы завершить ее выполнение и передать управление системе. Информация о том, какую именно программу (какой процесс) следует завершить, и куда именно, передать управление хранится в PSP выполняемой программы. В слове со смещением 16h от начала PSP записан идентификатор (т.е. сегментный адрес

PSP) родительского процесса, а в двухсловной ячейке со смещением 0Ah конкретный адрес возврата в него (рисунок 5.3). В то же время в области текущих данных SDA в слове со смещением 10h хранится идентификатор текущего процесса. Если компьютер не выполняет никакую программу, текущим является COMMAND.COM, и его ID записан в SDA. При запуске с клавиатуры любой программы (в том числе и той, которой предстоит сделаться резидентной), DOS записывает в SDA ее ID, а в PSP запущенной программы – ID родительского процесса и адрес возврата в него, чем и обеспечивается возможность завершения запущенной программы и возврата в систему. Для всех программ, запускаемых с клавиатуры (т.е. с помощью командного процессора) родительским процессом является COMMAND.COM.

Рисунок 5.3 – Поля PSP и SDA, имеющие отношение к завершению задачи

Функция завершения 4Ch, выполнив все предназначенные ей действия по закрытию файлов, восстановлению векторов 22h, 23h и 24h, освобождению памяти, занимаемой текущим процессом, переносит в SDA содержимое слова PSP со смещением 16h, назначая родительский по отношению к данному процесс, текущим, передавая управление по адресу возврата, записанному в PSP.

Таким образом, для того, чтобы из некоторой программы, выполняющей функции выгружающей, завершить другую (резидентную) программу, нужно выполнить следующие действия (рисунок 5.4):

1) занести в SDA ID завершаемой программы, чтобы объявить ее текущей, и чтобы функция 4Ch завершала именно ее;

2) занести в слово со смещением 16h в PSP завершаемой программы ID выгружающей программы, чтобы после завершения резидентной программы текущей снова стала выгружающая;

3) занести в двухсловную ячейку со смещением 0Ah в PSP завершаемой программы требуемый адрес возврата в выгружающую программу;

4) восстановить кадр стека выгружающей программы и, при необходимости, сегментные регистры, а также и регистры общего назначения, учитывая, что функция 4Ch разрушает все регистры.

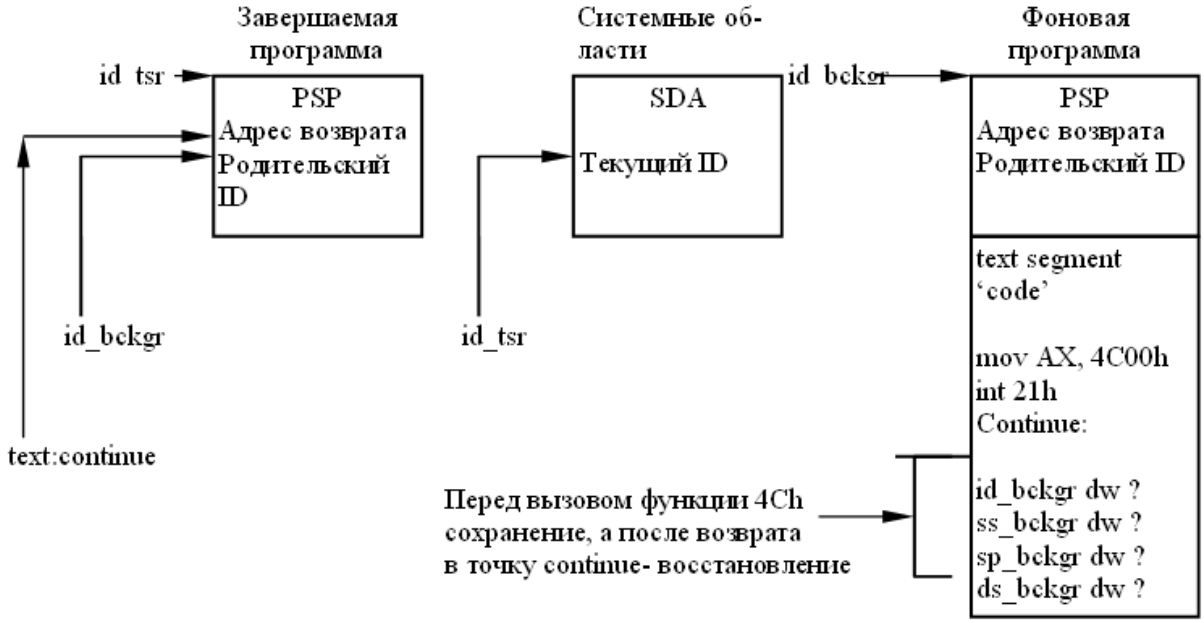

Рисунок 5.4 – Завершение резидентной программы из транзитной. Индексом tsr обозначена завершаемая (резидентная) программа, индексом bckgr – выгружающая (фоновая).

– 3.6 Свопинг области текущих данных DOS

Нереентерабельность DOS связана с использованием ею в процессе выполнения системных функций области текущих данных SDA, где находятся стеки DOS, а также ячейки для хранения целого ряда характерных адресов и других величин, в частности, кадров стеков. Наиболее радикальным методом придания DOS свойств реентерабельности (при вызовах DOS в обработчиках аппаратных прерываний) является сохранение в прикладном обработчике перед вызовом каких-либо системных функций всей области текущих данных, и восстановление исходного содержимого SDA после завершения работы с DOS. Такого рода процедура называется свопингом, что и дало название области текущих данных (Swappale Data Area). Перед выполнением свопинга целесообразно проверить состояние флага InDOS, так как если этот флаг окажется сброшен, в свопинге нет необходимости.

4 Задание для самостоятельного выполнения

–

1. Составить резидентную программу обработки прерываний от клавиатуры анализирующий скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. В обработчике прерываний от клавиатуры предусмотреть проверку состояния флага InDOS и, если DOS свободна – запускать task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществить с помощью обработчика прерывания от таймера, который, обнаружив установленный флаг запроса запуска, в свою очередь проверяет флаг InDOS и либо завершается (если DOS занята), либо запускает task (если DOS свободна). Для завершения задачи, использовать клавишу <Esc> (код 01h). В процессе завершения программы восстановить перехваченные векторы и выгрузить программу из памяти при помощи вызова функции 4Ch.

2. Составить резидентную программу обработки прерываний от таймера, анализирующую скэн-код нажатой клавиши. При изменении текущего времени активизирует процедуру task, которая выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. При поступлении клавиши <Esc> (код 01h) производится выгрузка резидентной программы с помощью функции 4Ch..

3. Составить резидентную программу обработки прерываний от клавиатуры анализирующий скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. В обработчике прерываний от клавиатуры предусмотреть проверку состояния флага InDOS и, если DOS свободна – запускать task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществить с помощью обработчика прерывания от таймера. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. По нажатию «серого плюса» активизируется процедура task. По

нажатию «серого минуса» восстанавливается исходное содержимое экрана. Выгрузка резидентной программы с помощью специальной транзитной программы выгрузки со средствами проверки на повторную загрузку и выгрузки с помощью мультиплексного прерывания 2Fh.

4. Составить резидентную программу обработки прерываний от таймера, анализирующую скэн-код нажатой клавиши. При изменении текущего времени активизирует процедуру task, которая выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – прямым доступом к видео памяти. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. При поступлении клавиши <Esc> (код 01h) производится выгрузка резидентной программы с помощью функции 4Ch. В процессе завершения программы восстановить первоначальный вид экрана.

5. Составить резидентную программу обработки прерываний от клавиатуры анализирующий скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. В обработчике прерываний от клавиатуры предусмотреть проверку состояния флага InDOS и, если DOS свободна – запускать task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществить с помощью обработчика прерывания от таймера. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. По нажатию «серого плюса» активизируется процедура task. Выгрузка резидентной программы с помощью повторного запуска с параметром /u со средствами проверки на повторную загрузку и выгрузки с помощью мультиплексного прерывания 2Fh.

6. Составить резидентную программу обработки прерываний от клавиатуры анализирующий скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстанов-

ления первоначального вида экрана. По нажатию «серого плюса» активизируется процедура task. По нажатию «серого минуса» восстанавливается исходное содержимое экрана. Завершение резидентной программы производится из другой (транзитной) программы. Резидентная программа активизируется из транзитной синхронно, через вектор 60h. Она открывает файл с известным ей именем и дописывает в его конец некоторые данные. Транзитная программа сначала активизирует резидентную программу, после чего завершает ее работу и выгружает из памяти функцией 4Ch.

7. Составить резидентную программу обработки прерываний от клавиатуры анализирующую скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. В обработчике прерываний от клавиатуры предусмотреть проверку состояния флага InDOS и, если DOS свободна – запускать task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществить с помощью обработчика прерывания int 28h. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. По нажатию «серого плюса» активизируется процедура task. По нажатию «серого минуса» восстанавливается исходное содержимое экрана. Для завершения задачи, использовать клавишу <Esc> (код 01h). В процессе завершения программы восстановить перехваченные векторы и выгрузить программу из памяти при помощи вызова функции 4Ch.

8. Составить обработчик прерываний от клавиатуры анализирующий скэн-код нажатой клавиши и при поступлении кода "серого плюса" активизирует процедуру task, которая получает текущую дату (функция 2Ah) и выводит его на экран средствами BIOS. В обработчике прерываний от клавиатуры предусмотреть проверку повторного вызова используемой функции BIOS. Повторные попытки активизации task осуществить с помощью обработчика прерываний от таймера. Для завершения задачи, использовать повторный запуск программы с параметром /u. Выгрузить программу из памяти при помощи мультиплексорного прерывания 2Fh.

9. Составить резидентную программу обработки прерываний от клавиатуры анализирующую скэн-код нажатой клавиши. При поступлении кода "серого плюса" запрещает вывод на экран средствами функции

09h DOS. При поступлении кода "серого минуса" разрешает вывод на экран средствами функции 09h DOS. . Проверка работы и завершение резидентной программы производится из другой (транзитной). Транзитная программа выводит в цикле сообщения при помощи функции 09h, после чего завершает ее работу и выгружает из памяти функцией 4Ch.

10. Составить обработчик прерываний от клавиатуры анализирующий скэн-код нажатой клавиши и при поступлении кода "серого плюса" активизирует процедуру task, которая получает текущую дату (функция 2Ah) и выводит ее на экран средствами DOS. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. При поступлении клавиши <Esc> (код 01h) производится выгрузка резидентной программы с помощью функции 4Ch. В процессе завершения программы восстановить первоначальный вид экрана.

11. Составить резидентную программу обработки прерываний от клавиатуры анализирующий скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. В обработчике прерываний от клавиатуры предусмотреть проверку состояния флага InDOS и, если DOS свободна – запускать task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществить с помощью обработчика прерывания от таймера, который, обнаружив установленный флаг запроса запуска, в свою очередь проверяет флаг InDOS и либо завершается (если DOS занята), либо запускает task (если DOS свободна). Для завершения задачи, использовать клавишу <Esc> (код 01h). В процессе завершения программы восстановить перехваченные векторы и выгрузить программу из памяти при помощи вызова функции 4Ch.

12. Составить резидентную программу обработки прерываний от таймера, анализирующую скэн-код нажатой клавиши. При изменении текущего времени активизирует процедуру task, которая выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. При поступлении клавиши <Esc> (код 01h) производится выгрузка резидентной программы с помощью функции 4Ch..

13. Составить резидентную программу обработки прерываний от клавиатуры анализирующий скэн-код нажатой клавиши. При поступле-

нии кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. В обработчике прерываний от клавиатуры предусмотреть проверку состояния флага InDOS и, если DOS свободна – запускать task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществить с помощью обработчика прерывания от таймера. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. По нажатию «серого плюса» активизируется процедура task. По нажатию «серого минуса» восстанавливается исходное содержимое экрана. Выгрузка резидентной программы с помощью специальной транзитной программы выгрузки со средствами проверки на повторную загрузку и выгрузки с помощью мультиплексного прерывания 2Fh.

14. Составить резидентную программу обработки прерываний от таймера, анализирующую скэн-код нажатой клавиши. При изменении текущего времени активизирует процедуру task, которая выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – прямым доступом к видео памяти. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. При поступлении клавиши <Esc> (код 01h) производится выгрузка резидентной программы с помощью функции 4Ch. В процессе завершения программы восстановить первоначальный вид экрана.

15. Составить резидентную программу обработки прерываний от клавиатуры анализирующий скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. В обработчике прерываний от клавиатуры предусмотреть проверку состояния флага InDOS и, если DOS свободна – запускать task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществить с помощью обработчика прерывания от таймера. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида

экрана. По нажатию «серого плюса» активизируется процедура task. Выгрузка резидентной программы с помощью повторного запуска с параметром /u со средствами проверки на повторную загрузку и выгрузки с помощью мультиплексного прерывания 2Fh.

16. Составить резидентную программу обработки прерываний от клавиатуры анализирующий скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. По нажатию «серого плюса» активизируется процедура task. По нажатию «серого минуса» восстанавливается исходное содержимое экрана. Завершение резидентной программы производится из другой (транзитной) программы. Резидентная программа активизируется из транзитной синхронно, через вектор 60h. Она открывает файл с известным ей именем и дописывает в его конец некоторые данные. Транзитная программа сначала активизирует резидентную программу, после чего завершает ее работу и выгружает из памяти функцией 4Ch.

17. Составить резидентную программу обработки прерываний от клавиатуры анализирующую скэн-код нажатой клавиши. При поступлении кода "серого плюса" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. В обработчике прерываний от клавиатуры предусмотреть проверку состояния флага InDOS и, если DOS свободна – запускать task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществить с помощью обработчика прерывания int 28h. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. По нажатию «серого плюса» активизируется процедура task. По нажатию «серого минуса» восстанавливается исходное содержимое экрана. Для завершения задачи, использовать клавишу <Esc> (код 01h). В процессе завершения программы восстановить перехваченные векторы и выгрузить программу из памяти при помощи вызова функции 4Ch.

18. Составить обработчик прерываний от клавиатуры анализирующий скэн-код нажатой клавиши и при поступлении кода "серого плюса" активизирует процедуру task, которая получает текущую дату (функция 2Ah) и выводит его на экран средствами BIOS. В обработчике прерываний от клавиатуры предусмотреть проверку повторного вызова используемой функции BIOS. Повторные попытки активизации task осуществить с помощью обработчика прерываний от таймера. Для завершения задачи, использовать повторный запуск программы с параметром /u. Выгрузить программу из памяти при помощи мультиплексорного прерывания 2Fh.

19. Составить резидентную программу обработки прерываний от клавиатуры анализирующую скэн-код нажатой клавиши. При поступлении кода "серого плюса" запрещает вывод на экран средствами функции 09h DOS. При поступлении кода "серого минуса" разрешает вывод на экран средствами функции 09h DOS. . Проверка работы и завершение резидентной программы производится из другой (транзитной). Транзитная программа выводит в цикле сообщения при помощи функции 09h, после чего завершает ее работу и выгружает из памяти функцией 4Ch.

20. Составить обработчик прерываний от клавиатуры анализирующий скэн-код нажатой клавиши и при поступлении кода "серого плюса" активизирует процедуру task, которая получает текущую дату (функция 2Ah) и выводит ее на экран средствами DOS. Предусмотреть сохранение в буфере программы исходного содержимого той части экрана, куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана. При поступлении клавиши <Esc> (код 01h) производится выгрузка резидентной программы с помощью функции 4Ch. В процессе завершения программы восстановить первоначальный вид экрана.

5 Контрольные вопросы

1. Какие проблемы возникают при разработке реальных резидентных программ и обработчиков аппаратных прерываний?
2. Почему нельзя обращаться к услугам файловой системы в резидентном обработчике прерываний?
3. Как осуществляется в резидентной программе проверка на повторную установку?
4. Для чего используется недокументированная функция DOS 5B06h?

5. Как используется флаг занятости DOS?
6. Как выполняется асинхронная активизация резидентных программ командами с клавиатуры?
7. Как осуществляется работа с файлами в резидентном обработчике аппаратных прерываний?
8. Как осуществляют выгрузку из памяти резидентных программ?

Лабораторная работа №6. Защищенный режим работы процессора

– 1 Цель и порядок работы

Цель работы – ознакомиться с защищенным режимом работы процессора и научиться разрабатывать программы, используя этот режим.

Порядок выполнения работы:

– ознакомиться с описанием лабораторной работы;
– изучить возможности работы с защищённым режимом;
– получить задание у преподавателя;
– написать программу, ввести программу, отладить и решить ее на ЭВМ;
– оформить отчет.

– 2 Теоретические основы

Перевод микропроцессора в защищенный режим позволяет полностью реализовать все возможности, заложенные в его архитектуру и недоступные в реальном режиме. Сюда можно отнести:

- увеличение адресуемого пространства до 16 Мбайт для МП 286 и до 4 Гбайт для МП 386;
- возможность работать в виртуальном адресном пространстве, превышающем максимально возможный объем физической памяти. Для МП 286 виртуальное пространство составляет 1 Гбайт, а для МП 386 и 486 – 64 Гбайт. Правда, для реализации виртуального режима необходимы, помимо дисков большой емкости, еще и соответствующая операционная система, которая хранит все сегменты выполняемых программ в большом дисковом пространстве, автоматически загружая в оперативную память те или иные сегменты по мере необходимости;
- организация многозадачного режима с параллельным выполнением нескольких программ (процессов). Многозадачный режим организует многозадачная операционная система, однако, микропроцессор предоставляет необходимый для этого режима мощный и надежный механизм защиты задач друг от друга с помощью четырехуровневой системы привилегий;
- страничная организация памяти, повышающая уровень защиты задач друг от друга и эффективность их выполнения.

При включении микропроцессора в нем автоматически устанавливается режим реального адреса. Переход в защищенный режим осу-

ществляется программно путем выполнения соответствующей последовательности команд. Реальный и защищенный режимы не совместимы!

Архитектура современного микропроцессора необычайно сложна. Столь же сложными оказываются и программы, использующие средства защищенного режима. Отдельные архитектурные особенности защищенного режима оказываются в достаточной степени замкнутыми и не зависящими друг от друга. Так, при работе в однозадачном режиме отпадает необходимость в изучении многообразных и замысловатых методов взаимодействия задач; во многих случаях можно отключить (или, точнее, не включать) механизм страничной организации памяти; часто нет необходимости использовать уровни привилегий. Начнем изучение защищенного режима с рассмотрения простейшей программы, которая, будучи запущена обычным образом под управлением MS-DOS, переключает процессор в защищенный режим, выводит на экран для контроля несколько символов, переходит назад в реальный режим и завершается стандартным для DOS образом.

Перед запуском программ защищенного режима следует выгрузить драйверы обслуживания расширенной памяти (типа EMM386.EXE) и оболочку Windows.

Для проверки работы программы удобно использовать специальные программы для эмуляции аппаратного обеспечения различных платформ. Наиболее распрстранены такие виртуальные машины как VMware, Bochs, Parallels Workstation, QEMU, Virtual PC, VirtualBox. Данные виртуальные машины являются гипервизорами (эмуляторами аппаратного обеспечения) и не оеспечивают виртуализацию приложений, как например Java Virtual Machine. В качестве основы для проведения лабораторных работ предлагается свободнораспространяемая виртуальная машина qemu с установленной операционной системой MS DOS 6.22.

Пример 6.1 – Переход в защищенный режим и обратно

```
;В защищенном режиме вывод фиксированных символов на экран
IDEAL                ; (1) Включение режима IDEAL
P386                 ; (2) Разрешение трансляции всех, в том числе
                     ;     привилегированных команд МП 386 и 486
model small          ; (3)
STACK 100h           ; (4) Оперделение сегмента стека и его размера
;Структура для описания дескрипторов сегментов
struc descr          ; (5)
      limit   dw 0   ; (6)
      base_l  dw 0   ; (7)
```

```
        base_m  db 0    ; (8)
        attr_1  db 0    ; (9)
        attr_2  db 0    ; (10)
        base_h  db 0    ; (11)
ends descr              ; (12)
DATASEG                 ; (13) Начало сегмента данных
;Таблица глобальных дескрипторов GDT
   gdt_null     descr <0,0,0,0,0,0>           ;(14) Селектор = 0
   gdt_data     descr <data_size-1,0,0,92h,0,0> ;(15) Селектор = 8
   gdt_code     descr <code_size-1,0,0,98h,0,0> ;(16) Селектор = 16
   gdt_stack    descr <100h-1,0,0,92h,0,0>      ;(17) Селектор = 24
   gdt_screen   descr <4095,8000h,0Bh,92h,0,0>  ;(18) Селектор = 32
  gdt_size = $-gdt_null                        ;(19)
  ;Поля данных программы
   pdescr       dp 0                          ;(20)
   sym          db 1                          ;(21)
   attr         db 1Eh                        ;(22)
   mes          db 10,13,'Real mode',"$"      ;(23)
  data_size = $-gdt_null                       ;(24)
ends                                           ;(25)
CODESEG                        ; (26) Начало сегмента команд
assume cs: @code, ds:@data     ; (27)
sttt equ $                     ; (28)
start:                         ; (29)
   xor  eax, eax               ; (30)
   mov  ax, @data              ; (31) Инициализация реального
   mov  ds, ax                 ; (32) режима
   ;Вычислим 32-битовый линейный адрес сегмента данных и загрузим
его в дескриптор
   ;сегмента данных в таблице GDT. В регистре AX уже находится
сегментный адрес.
   ;Умножим его на 16 сдвигом влево на 4 бита с размещением результата
в регистре EAX
   shl  eax, 4     ; (33) Сдвинем влево EAX на 4 бита (равносильно
умножению на 16)
                   ;      Теперь в EAX 32-битовый линейный адрес сегмента
данных
   mov  ebp, eax   ; (34) Сохраняем в EBP начало сегмента
   mov  bx, offset gdt_data       ; (35) В BX адрес дескриптора
   mov  [(descr ptr bx).base_l], ax; (36) Загрузим младшую часть базы
```

```
  rol  eax, 16    ; (37) Поместим циклическим сдвгом старшие биты
линейного адреса
                ;      (находящиеся в битах [19...16] регистра EAX) в AL
  mov  [(descr ptr bx).base_m], al; (38) Загрузим среднюю часть базы
  ;Вычислим 32-битовый линейный адрес сегмента команд и загрузим
его
  ;в дескриптор сегмента команд в таблице глобальных дескрипторов
  xor  eax, eax    ; (39) Очистка EAX
  mov  ax, cs      ; (40) В AX - адрес сегмента команд
  shl  eax, 4      ; (41) Та же процедура умножения сегментного адреса на
16 сдвигом
  mov  bx, offset gdt_code       ; (42) BX = адрес дескриптора
  mov  [(descr ptr bx).base_l], ax; (43) Загруэка младшей части базы
  rol  eax, 16                   ; (44)
  mov  [(descr ptr bx).base_m], al; (45) Загруэка средней частей базы
  ;Аналогично для адреса сегмента стека
  xor  eax, eax                  ; (46)
  mov  ax, ss                    ; (47)
  shl  eax, 4                    ; (48)
  mov  bx, offset gdt_stack      ; (49) BX = адрес дескриптора стека
  mov  [(descr ptr bx).base_l], ax; (50)
  rol  eax, 16                   ; (51)
  mov  [(descr ptr bx).base_m], al; (52)
  ;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
  mov  [dword ptr pdescr+2], ebp   ; (53) Занесем в pdescr адрес GDT
                                   ; (совпадает с адресом начала сегмента данных)
  mov  [word ptr pdescr], gdt_size-1  ; (54) Граница GDT
  lgdt [pword ptr pdescr]            ; (55) Загрузим регистр GDTR
  ;Подготовимся к переходу в защищенный режим
  cli             ; (56) Запрет аппаратных прерываний
  in   al, 70h    ; (57) Запрет немаскируемых прерываний (NMI) через
  or   al, 80h    ; (58) индексный порт CMOS (70h) путем установки
  out  70h, al    ; (59) бита 7 (10000000b=80h)
  jmp  $+2        ; (60) Задержка (в данном случае необязательно)
  ;Переход в защищенный режим
  ;Слово состояния машины == младшее слово регистра CR0
  mov  eax, CR0       ; (61) Получим слово состояния машины
  or   eax, 1         ; (62) Установим бит PE
  mov  CR0, eax       ; (63) Запишем назад слово состояния
 ;Теперь процессор работает в защищённом режиме
```

```
  ;Загружаем в CS:IP селектор:смещение точки continue и заодно
очищаем очередь команд
        db 0EAh              ; (64) Код команды far jmp
        dw offset continue   ; (65) Смещение
        dw 16                ; (66) Селектор сегмента команд
continue:             ; (67)
   ;Делаем адресуемыми данные
   mov  ax, 8         ; (68) Селектор сегмента данных
   mov  ds, ax        ; (69)
   ;Делаем адресуемым стек
   mov  ax, 24        ; (70) Селектор сегмента стека
   mov  ss, ax        ; (71)
   ;Инициализируем ES селектором видеобуфера и выводим символы
   mov  ax, 32        ; (72) Селектор сегмента видеобуфера
   mov  es, ax        ; (73)
   ;Подготовка к выводу на экран
   mov  bx, 800             ; (74) Начальное смещение на экране
   mov  cx, 640             ; (75) Число выводимым символов
   mov  ax, [word ptr sym]  ; (76) Начальный символ с атрибутом
screen:               ; (77)
   mov  [es:bx], ax    ; (78) Вывод в видеобуфер
   add  bx, 2          ; (79) Сместимся в видеобуфере
   inc  ax             ; (80) Следующий символ
   loop screen         ; (81) Цикл вывода на экран
   ;Вернемся в реальный режим
   mov  eax, CR0       ; (82) Получим слово состояния машины
   and  al, 0FEh       ; (83) Сбросим бит PE
   mov  CR0, eax       ; (84) Запишем назад слово состояния
   ;Теперь процессор работает в реальном режиме
   ;Загружаем в CS:IP селектор:смещение точки return
        db 0EAh              ; (85) Код команды far jmp
        dw offset return     ; (86) Смещение точки возврата
        dw @code             ; (87) Сегментный адрес сегмента кода

return:                   ; (88)
   ;Восстановим операционную среду реального режима
   mov  ax, @data           ; (89) Восстановим адресуемость
   mov  ds, ax              ; (90) данных
   mov  ax, @stack          ; (91) Восстановим
```

```
  mov  ss, ax              ; (92) адресуемость
  mov  sp, 100h            ; (93) стека
  ;Разрешим аппаратные и немаскируемые прерывания
  sti                      ; (94) Разрешение прерываний
  in   al, 70h             ; (95)
  and  al, 07Fh            ; (96) Сброс бита 7 в порте CMOS
  out  70h, al             ; (97) разрешение NMI
  ;Проверим выполнение функций DOS после возврата в реальный
режим
   mov  ah, 09h            ; (98) Функция вывода на экран строки
   mov  dx, offset mes     ; (99) Адрес строки
   int  21h                ; (100) Выэов DOS

   mov  ax, 4C00h          ; (101) Завершим программу обычным
   int  21h                ; (102) образом
ends                       ; (103) Конец сегмента команд
code_size=$-sttt           ; (104) Размер сегмента команд
end start                  ; (105) Конец программы
end                        ; (106)
```

32-разрядные микропроцессоры отличаются от предыдущих расширенным набором команд, часть которых относится к привилегированным. Для того чтобы разрешить транслятору обрабатывать эти команды, в текст программы необходимо включить директиву ассемблера P386.

Программа начинается с описания структуры дескриптора сегмента. В защищенном режиме для каждого сегмента программы (т.е. для сегментов команд, данных и стека) в программе должен быть определен дескриптор – 8-байтовое поле, в котором в определенном формате записываются базовый адрес сегмента, его длина и некоторые другие характеристики (рис. 6.1).

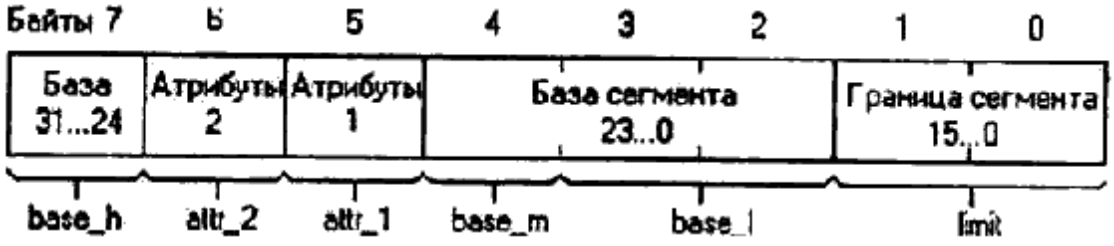

Рисунок 6.1 – Дескриптор сегмента.

Теперь для обращения к требуемому сегменту программист заносит в сегментный регистр не сегментный адрес, а так называемый селектор (рис. 6.2), в состав которого входит номер (индекс) соответствующего

сегменту дескриптора. Процессор по этому номеру находит нужный дескриптор, извлекает из него базовый адрес сегмента и, прибавляя к нему указанное в конкретной команде смещение (относительный адрес), формирует адрес ячейки памяти. Индекс дескриптора (0, 1, 2 и т.д.) записывается в селектор начиная с бита 3, что эквивалентно умножению его на 8. Таким образом, можно считать, что селекторы последовательных дескрипторов представляют собой числа 0, 8, 16, 24 и т.д. (см. предложения 14…18).

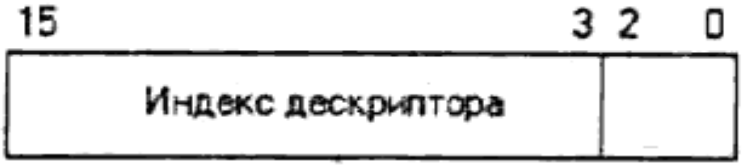

Рисунок 6.2 – Селектор дескриптора.

Структура descr предоставляет шаблон для дескрипторов сегментов, облегчающий их формирование. Сравнивая описание структуры descr в программе с рис. 6.1, нетрудно проследить их соответствие друг другу.

Граница (limit) сегмента представляет собой номер последнего байта сегмента. Так, для сегмента размером 375 байт граница равна 374. Поле границы состоит из 20 бит и разбито на две части. Младшие 16 бит границы занимают байты 0 и 1 дескриптора, а старшие 4 бита входят в байт атрибутов 2, занимая в нем биты 0…3. Получается, что размер сегмента ограничен величиной 1 Мбайт. На самом деле это не так. Граница может указываться либо в байтах (и тогда, действительно, максимальный размер сегмента равен 1 Мбайт), либо в блоках по 4 Кбайт (и тогда размер сегмента может достигать 4 Гбайт). В каких единицах задается граница, определяет самый старший бит байта 7 (атрибуты 2). Этот бит называется битом дробности (гранулярности). Если он равен 0, граница указывается в байтах; если 1 - в блоках по 4 Кбайт.

База сегмента (32 бита) определяет начальный линейный адрес сегмента в адресном пространстве процессора. Линейным называется адрес, выраженный не в виде комбинации *сегмент:смещение*, а просто номером байта в адресном пространстве. Казалось бы, линейный адрес - это просто другое название физического адреса. Для нашего примера это так и есть, в нем линейные адреса совпадают с физическими. Если, однако, в процессоре включен блок страничной организация памяти, то процедура преобразования адресов усложняется. Отдельные блоки размером 4 Кбайт (страницы) линейного адресного пространства могут произвольным образом отображаться на физические адреса, в частности и так, что большие линейные адреса отображаются на начало физической памяти и

наоборот. Страничная адресация осуществляется аппаратно (хотя для ее включения требуются определенные программные усилия) и действует независимо от сегментной организации программы. Поэтому во всех программных структурах защищенного режима фигурируют не физические, а линейные адреса. Если страничная адресация выключена, эти линейные адреса совпадают с физическими, если включена – могут и не совпадать.

Страничная организация повышает эффективность использования памяти программами, однако, практически она имеет смысл лишь при выполнении больших по размеру задач, когда объем адресного пространства задачи (виртуального адресного пространства) превышает наличный объем памяти. В рассматриваемых в лабораторных работах примерах используется чисто сегментная адресация без деления на страницы, и линейные адреса совпадают с физическими.

Поскольку в дескриптор записывается 32-битовый линейный базовый адрес (номер байта), сегмент в защищенном режиме может начинаться на любом байте, а не только на границе параграфа, и располагаться в любом месте адресного пространства 4 Гбайт.

Поле базы, как и поле границы, разбито на 2 части: биты 0…23 занимают байты 2, 3 и 4 дескриптора, а биты 24…31 - байт 7. Для удобства программного обращения в структуре descr база описывается тремя полями: младшим словом (base_l) и двумя байтами: средним (base_m) и старшим (base_h).

В байте атрибутов 1 задастся рад характеристик сегмента. Не вдаваясь пока в подробности этих характеристик, укажем, что в примере 3.1 используются сегменты двух типов: сегмент команд, для которого байт attr_1 должен иметь значение 98h, и сегмент данных (или стека) с кодом 92h.

Некоторые дополнительные характеристики сегмента указываются в старшем полубайте байта attr_2 (в частности, тип дробности). Для всех наших сегментов значение этого полубайта равно 0.

Сегмент данных @data, который для удобства изучения функционирования программы расположен в начале программы, до сегмента команд, начинается с описания важнейшей системной структуры – таблицы глобальных дескрипторов. Как уже отмечалось выше, обращение к сегментам в защищенном режиме возможно исключительно через дескрипторы этих сегментов. Таким образом, в таблице дескрипторов должно быть описано столько дескрипторов, сколько сегментов использует программа. В нашем случае в таблицу включены, помимо обязательного нулевого дескриптора, всегда занимающего первое место в таблице, четыре

дескриптора для сегментов данных, команд, стека и дополнительного сегмента данных, который мы наложим на видеобуфер, чтобы обеспечить возможность вывода в него символов. Порядок дескрипторов в таблице (кроме нулевого) не имеет значения.

Помимо единственной таблицы глобальных дескрипторов (она часто называется GDT от Global Descriptor Table) в памяти может находиться множество таблиц локальных дескрипторов (LDT от Local Descriptor Table). Разница между ними в том, что сегменты, описываемые глобальными дескрипторами, доступны всем задачам, выполняемым процессором, а к сегментам, описываемым локальными дескрипторами, может обращаться только та задача, в которой эта дескрипторы описаны. Поскольку пока мы имеем дело с однозадачным режимом, локальная таблица нам не нужна.

Поля дескрипторов для наглядности заполнены конкретными данными явным образом, хотя объявление структуры descr с нулями во всех полях позволяет описать дескрипторы несколько короче, например:

```
gdt_null  descr <> ;Селектор 0 - обязательный нулевой дескриптор
gdt_data  descr <data_size-1,,,92h,,> ;Селектор 8 - сегмент данных
```

В дескрипторе gdt_data, описывающем сегмент данных программы, заполняется поле границы сегмента (фактическое значение размера сегмента data_size будет вычислено транслятором, см. предложение 24), а также байт атрибутов 1. Код 92h говорит о том, что это сегмент данных с разрешением записи и чтения. База сегмента, т.е. физический адрес его начала, в явной форме в программе отсутствует, поэтому ее придется программно вычислить и занести в дескриптор уже на этапе выполнения.

Дескриптор gdt_code сегмента команд заполняется схожим образом. Код атрибута 98h обозначает, что это исполняемый сегмент, к которому, между прочим, запрещено обращение с целью чтения или записи. Таким образом, сегменты команд в защищенном режиме нельзя модифицировать по ходу выполнения программы.

Дескриптор gdt_stack сегмента стека имеет, как и любой сегмент данных, код атрибута 92h, что разрешает его чтение и запись, и явным образом заданную границу – 255 байт (100h-1), что соответствует размеру стека. Базовый адрес сегмента стека так же придется вычислить на этапе выполнения программы.

Последний дескриптор gdt_screen описывает страницу 0 видеобуфера. Размер видеостраницы, как известно, составляет 4096 байт, поэтому в поле границы указано число 4095. Базовый физический адрес страницы известен, он равен 0B8000h. Младшие 16 бит базы (число 8000h) заполняют слово base_l дескриптора, биты 16…19 (число 0Bh) - байт

base_m. Биты 20…31 базового адреса равны 0, поскольку видеобуфер размещается в первом мегабайте адресного пространства.

Перед переходом в защищенный режим процессору надо будет сообщить физический адрес таблицы глобальных дескрипторов и её размер (точнее, границу). Размер GDT определяется на этапе трансляции в предложении 19.

Сегмент команд @code начинается, объявлением CODESEG. Этот описатель объявляет, что в данном сегменте будут по умолчанию использоваться 16-битовые адреса и операнды. Наша программа будет запускаться под управлением DOS, которая работает в реальном режиме с 16-битовыми адресами и операндами. Указание описателя use16 не запрещает использовать в программе 32-битовые регистры.

Фактически вся программа примера 3.1, кроме ее завершающих строк, а также фрагмента, выполняемого в защищенном режиме, посвящена подготовке перехода в защищенный режим. Прежде всего, надо завершить формирование дескрипторов сегментов программы, в которых остались незаполненными базовые адреса сегментов. Базовые (32-битовые) адреса определяются путем умножения значений сегментных адресов на 16. Перед обычной инициализацией сегментного регистра DS (предложения 31-32), которая позволит нам обращаться к полям данных программы (в реальном режиме) выполняется очистка регистра EAX (предложение 30). В AX содержится сегментный адрес сегмента данных. Командой shl (предложение 33) содержимое регистра EAX также сдвигается на 4 бита, что соответствует умножению на 16. После этого в EAX будет располагаться линейный адрес сегмента данных, совпадающий в нашем случае с физическим.

Следующей командой (предложения 36) содержимое AX отправляется в поле base_l дескриптора gdt_data. После этого нам необходимо получить старшую часть линейного адреса сегмента (находящуюся в старшей части EAX). Для этого используется команда rol, циклически обращающая биты регистра. При сдвиге на 16 бит влево старшая и младшая (AX) части 32-х битного регистра меняются местами. Таким образом, необходимая нам часть адреса окажется в AL, откуда мы помещаем ее в поле base_m (предложение 38).

Поскольку нам еще понадобится линейный адрес сегмента данных, сохраним его предварительно в регистре EBP (предложение 34).

Аналогично вычисляются 32-битовые адреса сегментов команд и стека, помещаемые в дескрипторы gdt_code и gdt_stack.

Следующий этап подготовки к переходу в защищенный режим – загрузка в регистр процессора GDTR (Global Descriptor Table Register, ре-

гистр таблицы глобальных дескрипторов) информации о таблице глобальных дескрипторов. Эта информация включает в себя линейный базовый адрес таблицы и ее границу и размещается в 6 байтах поля данных, называемого иногда псевдодескриптором. Для загрузки GDTR существует специальная привилегированная команда lgdt (load global descriptor table, загрузка таблицы глобальных дескрипторов), которая требует указания в качестве операнда имени псевдодескриптора. Формат псевдодескриптора приведен на рисунке 6.3.

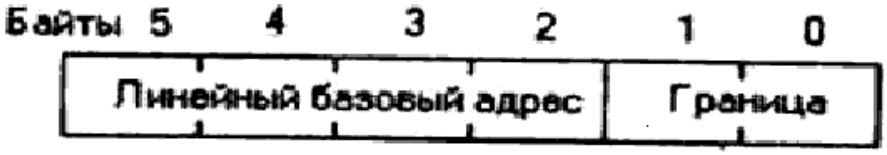

Рисунок 6.3 – Формат псевдодескриптора.

Заполнение псевдодескриптора упрощается вследствие того, что таблица глобальных дескрипторов расположена в начале сегмента данных, и ее базовый адрес совпадает с базовым адресом всего сегмента, который уже был вычислен и помещен в дескриптор gdt_data. В предложении 53 компоненты базового адреса переносятся из дескриптора в требуемые поля pdescr, а в предложении 54 заполняется поле границы. Команда lgdt загружает регистр GDTR и сообщает процессору о местонахождении и размере GDT.

Мы запускаем программу под управлением DOS, и естественно завершить ее также обычным образом, чтобы не нарушить работоспособность системы. Но в защищенном режиме запрещены любые обращения к функциям DOS или BIOS. Причина этого совершенно очевидна – и DOS, и BIOS являются программами реального режима, в которых широко используется сегментная адресация реального режима, т.е. загрузка в сегментные регистры сегментных адресов. В защищенном же режиме в сегментные регистры загружаются не сегментные адреса, а селекторы. Кроме того, обращение к функциям DOS и BIOS осуществляется с помощью команд программного прерывания int с определенными номерами, а в защищенном режиме эти команды приведут к совершенно иным результатам. Таким образом, программу, работающую в защищенном режиме, нельзя завершить средствами DOS. Сначала ее надо вернуть в реальный режим.

Возврат в реальный режим можно осуществить двумя способами.

Первый способ – программный сброс процессора. Действия процессора после сброса определяются одной из ячеек КМОП-микросхемы – байтом состояния отключения, располагаемым по адресу 0Fh. В частно-

сти, если в этом байте записан код 0Ah, после сброса управление немедленно передается по адресу, который извлекается из двухсловной ячейки 40h:67h, расположенной в области данных BIOS. Таким образом, для подготовки возврата в реальный режим мы должны в ячейку 40h:67h записать адрес возврата, а в байт 0Fh КМОП-микросхемы занести код 0Ah. Точка возврата может располагаться в любом месте программы.

Сброс процессора выполняется засылкой команды FEh в порт 64h контроллера клавиатуры. Эта команда возбуждает сигнал на одном из выводов контроллера клавиатуры, который приводит к появлению сигнала сброса на выводе RESET микропроцессора. Перед выполнением сброса текущее содержимое SP сохраняется в ячейке real_sp, чтобы после перехода в реальный режим можно было восстановить состояние стека.

После сброса процессор начинает работать в реальном режиме, причем управление передается программам BIOS. BIOS анализирует содержимое байта состояния отключения (0Fh) КМОП - микросхемы и, поскольку мы записали туда код 0Ah, осуществляет передачу управления по адресу, хранящемуся в ячейке 40h:67h области данных BIOS. В нашем случае переход осуществляется на метку return. Команда hlt (halt, останов) позволяет организовать ожидание сброса процессора, который выполняется не мгновенно. Вместо команды hlt можно было использовать бесконечный цикл. Если команду ожидания опустить, процессор до своего останова успеет выполнить несколько следующих команд, после чего все-таки передаст управление на адрес возврата.

Другой способ, выглядит несколько сложнее. Он заключается в восстановлении состояния процессора, его регистров, определенных областей памяти в исходное состояние. То есть, по сути, необходимо выполнить процесс обратный подготовке переходу в защищенный режим.

Еще одна важная операция, которую необходимо выполнить перед переходом в защищенный режим, заключается в запрете всех аппаратных прерываний. В защищенном режиме процессор выполняет процедуру прерывания не так, как в реальном. При поступлении сигнала прерывания процессор не обращается к таблице векторов прерываний в первом килобайте памяти, как в реальном режиме, а извлекает адрес программы обработки прерывания из таблицы дескрипторов прерываний, построенной схоже с таблицей глобальных дескрипторов и располагаемой в программе пользователя (или в операционной системе). В примере 6.1 такой таблицы нет, и на время работы программы прерывания придется запретить. Запрет всех аппаратных прерываний осуществляется командой cli (предложение 56). Однако желательно запретить еще и немаскируемые

прерывания, которые поступают в процессор по отдельной линии (вход NMI микропроцессора) и не управляются битом IF регистра флагов. Немаскируемые прерывания обычно используются для обработки таких катастрофических событий, как сбой питания, ошибка памяти или ошибка четности на магистрали; в реальном режиме для них зарезервирован вектор 02h. Для запрета немаскируемых прерываний не предусмотрено никаких специальных команд, однако, это можно сделать, установив старший бит в адресном (индексном) порте 70h КМОП-микросхемы.

В предложениях 57-59 из порта 70h читается текущее состояние КМОП-микросхемы, и запрещает немаскируемые прерывания установкой старшего (7-ого) бита и отправкой значения обратно в порт.

Для перехода могут быть использованы команды smsw (Store Machine Status Word, запись слова состояния машины) и lmsw (Load Machine Status Word, загрузка слова состояния машины) работающие с младшими 16 битами регистра CR0, называемыми **словом состояния машины**. Однако они практически не используются и существуют для совместимости с процессором 80286. Вместо них всегда удобнее использовать mov cr0,eax (вместо lmsw) и mov eax,cr0 (вместо smsw).

Переход в защищенный режим осуществляется установкой в 1 бита 0 регистра CR0, называемого PE (Protection Enabled). Поскольку остальные биты этого слова нам могут быть не известны, сначала мы читаем в регистр EAX содержимое CR0, затем устанавливаем в нем бит 0 и, наконец, записываем модифицированное значение назад в регистр CR0 процессора. Все последующие команды выполняются уже в защищенном режиме.

В предложениях 61…63 осуществляется перевод процессора в защищенный режим.

Хотя защищенный режим установлен, однако действия по настройке системы еще не закончены. Действительно, во всех используемых в программе сегментных регистрах хранятся не селекторы дескрипторов сегментов, а базовые сегментные адреса, не имеющие смысла в защищенном режиме. Между прочим, отсюда можно сделать вывод, что после перехода в защищенный режим программа не должна работать, так как в регистре CS пока еще нет селектора сегмента команд, и процессор не может обращаться к этому сегменту. В действительности это не совсем так.

В процессоре для каждого из сегментных регистров имеется так называемый теневой регистр дескриптора, который имеет формат дескриптора. Теневые регистры недоступны программисту; они автоматически загружаются процессором из таблицы дескрипторов каждый раз,

когда процессор загружает соответствующий сегментный регистр. Таким образом, в защищенном режиме программист имеет дело с селекторами, т.е. номерами дескрипторов, а процессор – с самими дескрипторами, хранящимися в теневых регистрах. Именно содержимое теневого регистра (в первую очередь, линейный адрес сегмента) определяет область памяти, к которой обращается процессор при выполнении конкретной команды.

В реальном режиме теневые регистры заполняются не из таблицы дескрипторов, а непосредственно самим процессором. В частности, процессор заполняет поле базы каждого теневого регистра линейным базовым адресом сегмента, полученным путем умножения на 16 содержимого сегментного регистра, как это и положено в реальном режиме. Поэтому после перехода в защищенный режим в теневых регистрах находятся правильные линейные базовые адреса, и программа будет выполняться правильно, хотя с точки зрения правил адресации защищенного режима содержимое сегментных регистров лишено смысла.

Тем не менее, после перехода в защищенный режим, следует загрузить в используемые сегментные регистры (и, в частности, в регистр CS) селекторы соответствующих сегментов. Это позволит процессору правильно заполнить все поля теневых регистров из таблицы дескрипторов. Пока эта операция не выполнена, некоторые поля теневых регистров (в частности, границы сегментов) могут содержать неверную информацию.

Загрузить селекторы в сегментные регистры DS, SS и ES не представляет труда (предложения 68…73). Но как загрузить селектор в программно недоступный регистр CS? Для этого можно воспользоваться искусственно сконструированной командой дальнего перехода, которая, как известно, приводит к смене содержимого и IP, и CS. Предложения 64…66 демонстрируют эту методику. В реальном режиме мы поместили бы во второе слово адреса в сегментный адрес сегмента команд, в защищенном же мы записываем в него селектор этого сегмента (число 16).

Команда дальнего перехода, помимо загрузки в CS селектора, выполняет еще одну функцию – она очищает очередь команд в блоке предвыборки команд процессора. Как известно, в современных процессорах с целью повышения скорости выполнения программы используется конвейерная обработка команд программы, позволяющая совместить во времени фазы их обработки. Одновременно с выполнением текущей (первой) команды осуществляется выборка операндов следующей (второй), дешифрация третьей и выборка из памяти четвертой команды. Таким образом, в момент перехода в защищенный режим уже могут быть расшифрованы несколько следующих команд и выбраны из памяти их

операнды. Однако эти действия выполнялись, очевидно, по правилам реального, а не защищенного режима, что может привести к нарушениям в работе программы. Команда перехода очищает очередь предвыборки, заставляя процессор заполнить ее заново уже в защищенном режиме.

Следующий фрагмент примера (предложения 72…81) является чисто иллюстративным. В нем инициализируется (по правилам защищенного режима) сегментный регистр ES и в видеобуфер экрана выводится некоторое количество цветных символов, чем подтверждается правильное функционирование программы в защищенном режиме.

Чтобы не нарушить работоспособность DOS, процессор следует вернуть в реальный режим, после чего можно будет завершить программу обычным образом.

Для этого необходимо перевести процессор обратно в реальный режим адресации, сбросив бит PE регистра CR0 (предложения 82…84). После этого необходимо загрузить в сегментный регистр CS корректное значение сегментного адреса сегмента кода, а в регистр IP значение смещения точки возврата (предложения 85…87). В нашем случае переход осуществляется на метку return (предложение 88).

Поскольку сегментные регистры содержат селекторы защищенного режима, их следует инициализировать заново, что и выполняется в предложениях 89…93. Заметим, что сохранение и восстановление указателя стека SP не является обязательным, во всяком случае, в нашем примере, где работа программы до перехода в защищенный режим и после возврата из него протекает независимо. Поэтому после возврата в реальный режим можно инициализировать SP заново, выполнив команду mov SP, 256 (в предположении, что стек имеет размер 256 байт).

Для восстановления работоспособности системы следует разрешить маскируемые (предложение 94) и немаскируемые прерывания (предложения 95…97), после чего программа может продолжаться уже в реальной режиме. В рассматриваемом примере для проверки работоспособности системы на экран выводится некоторый текст с помощью функции DOS 09h.

Программа завершается обычным образом функцией DOS 4Ch. Нормальное завершение программы и переход в DOS тоже в какой-то мере свидетельствует о ее правильности.

У рассмотренной программы имеется серьезный недостаток – полное отсутствие средств отладки. Для отладки программ защищенного режима используется механизм прерываний и исключений, в нашей же программе этот механизм не активизирован. Поэтому всякие неполадки при работе в защищенном режиме, которые с помощью указанного ме-

ханизма можно было бы обнаружить и проанализировать, в данном случае будут приводить к сбросу процессора. Однако после сброса программа, скорее всего, будет работать правильно: на экран будет выведена запланированная строка и программа завершится с передачей управления DOS. Таким образом, критерием программных ошибок защищенного режима может служить правильная в целом работа программы при отсутствии на экране цветных символов, которые должны выводиться в защищенном режиме.

– 3 Задание для самостоятельного выполнения

1. Изучить возможности работы в защищённом режиме.
2. Написать программу, переходящую в защищенный режим, выводящую в нем несколько символов на экран, выходящую из него и сигнализирующую об этом. При написании обратить внимание на следующие этапы:
    - 2.1. Определение структуры дескриптора.
    - 2.2. Определение таблицы дескрипторов и дескрипторов используемых сегментов.
    - 2.3. Подготовка таблицы дескрипторов к переходу в защищенный режим.
    - 2.4. Подготовка псевдодескриптора и инициализация регистра GDTR
    - 2.5. Запрет аппаратных и немаскируемых прерываний
    - 2.6. Непосредственно переход в защищенный режим
    - 2.7. Подготовка к работе в защищенном режиме и вывод в видеобуфер
    - 2.8. Подготовка к возврату и возврат в реальный режим
    - 2.9. Восстановление операционной среды реального режима
3. Отладить и протестировать полученную программу.
4. Оформить отчёт.

– 4 Контрольные вопросы

1. Каков формат дескриптора?
2. Каков формат псевдодескриптора?
3. Как переключиться в защищенный режим?
4. Какова структура таблицы дескрипторов?
5. Какова модель памяти при работе в защищенном режиме?

Лабораторная работа №7. Работа с расширенной памятью в защищенном режиме работы процессора

1 Цель и порядок работы

Цель работы – изучение методов с работой с расширенной памятью в защищенном режиме и использование ее при разработке программ.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- изучить возможности работы с защищённым режимом;
- получить задание у преподавателя;
- написать программу, ввести программу, отладить и решить ее на ЭВМ;
- оформить отчет.

2 Теоретические основы

В настоящей работе рассматривается принципы работы с большими сегментами данных, находящимися в расширенной памяти, т.е. за пределами мегабайтного адресного пространства.

Дескрипторы могут быть трех типов: дескрипторы памяти, системные дескрипторы и шлюзы. Форматы дескрипторов памяти и системных практически совпадают, формат же шлюза несколько отличается. Формат дескриптора памяти соответствует структуре descr, и изображен на рисунке 7.1.

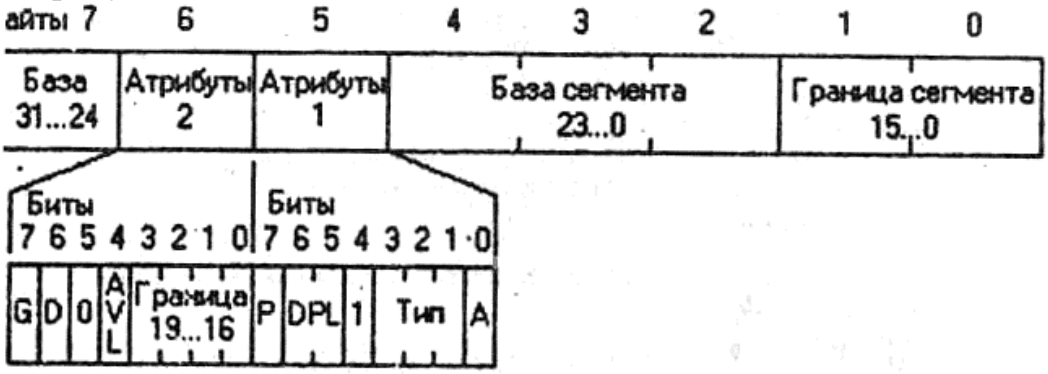

Рисунок 7.1 – Формат дескриптора сегмента памяти

Как видно из рисунка, дескриптор занимает 8 байтов. В байтах 2…4 и 7 записывается линейный сегментный адрес базы сегмента. Поскольку базовый адрес имеет 32 бита, он может быть назначен в любой точке 4-гигабайтного адресного пространства (естественно, только там, где имеется реальная память или другие адресуемые объекты, например, видеобуфер). В байтах 0-1 записываются младшие 16 бит границы сег-

мента, а в младшие четыре бита байта атрибутов 2 - оставшиеся биты 16…19. Таким образом, граница описывается 20-ю битами, и ее численное значение не может превышать 1М. Однако единицы, в которых задается граница, можно изменять, что осуществляется с помощью бита дробности G (бит 7 байта атрибутов 2). Если G=0, граница указывается в байтах; если 1 – в блоках по 4 Кбайт. Таким образом, размер сегмента можно задавать с точностью до байта, но тогда он не может быть больше 1 Мбайт; если же установить G=1, то сегмент может достигать 4 Гбайт, однако его размер будет кратен 4 Кбайт. При этом база сегмента и в том, и в другом случае задается с точностью до байта.

При выделении программе сегментов большого следует иметь в виду особенность вычисления процессором значения границы. Если G=1, истинная граница сегмента определяется следующим образом:

Граница сегмента = граница в дескрипторе * 4К + 4095

Таким образом, сегмент всегда простирается до конца последнего 4К-байтного блока. Пусть, например, базовый адрес сегмента равен 0, граница тоже равна 0, и бит дробности установлен. Тогда истинная граница сегмента будет равна 0*4К+4095=4095, т.е. сегмент будет занимать диапазон адресов 0…4095 и иметь размер ровно 4 Кбайт. Если установить значение границы, равное 1, сегмент будет иметь размер ровно 8 Кбайт и т.д.

Атрибуты сегмента занимают два байта дескриптора.

Бит A (Accessed, было обращение) устанавливается процессором в тот момент, когда в какой-либо сегментный регистр загружается селектор данного сегмента. Анализируя биты обращения различных сегментов, программа (в частности, операционная система защищенного режима) может судить о том, было ли обращение к данному сегменту после того, как программа сбросила бит A.

Тип сегмента занимает 3 бита (иногда бит A включают в поле типа, и тогда тип занимает 4 бит) и может иметь 8 значений, перечисленных в таблице 7.1.

Как видно из таблицы 1, тип определяет правила доступа к сегменту. Указав для некоторого сегмента данных тип 0, можно защитить его от модификации. При потолке записи в такой сегмент возникнет исключение общей защиты. Сегменту команд обычно присваивается тип 4, и тогда его можно только исполнять. Таким образом, в защищенном режиме не предусмотрено составление самомодифицирующихся программ.

Таблица 7.1 – Значения поля типа дескриптора сегмента памяти

Тип	X	Y	Z		Характеристики сегмента

0	0	0	0	000	Разрешено только чтение (сегмент данных)
1	0	0	1	001	Разрешены чтение и запись (сегмент данных)
2	0	1	0	010	Расширение вниз, разрешено только чтение (сегмент стека)
3	0	1	1	011	Расширение вниз, разрешены чтение и запись (сегмент стека)
4	1	0	0	100	Разрешено только исполнение (сегмент команд)
5	1	0	1	101	Разрешены исполнение и чтение (сегмент команд)
6	1	1	0	110	Разрешено только исполнение (подчиненный сегмент)
7	1	1	1	111	Разрешены исполнение и чтение (подчиненный сегмент)

Здесь биты типа, соответствуют столбцам и выполняют следующие функции:

столбец X (2-й бит типа):

0 – сегмент данных,

1 – сегмент кода;

столбец Y (1-й бит типа):

для данных – бит направления расширения сегмента:

0 – данные (расширение вниз, от младших адресов к старшим)

1 – стек (расширение вверх, от старших адресов к младшим)

для кода – бит подчинения

0 – главный сегмент

1 – подчиненный сегмент

столбец Z (0-й бит типа):

для данных – бит разрешения записи

0 – Только чтение (RO)

1 – Чтение и запись (RW)

для кода – бит разрешения чтения

0 – Только исполнение (exec)

1 – Исполнение и чтение (exec+read)

Подчиненные, или согласованные сегменты обычно используются для хранения подпрограмм общего пользования; для них не действуют общие правила защиты программ друг от друга.

Сегменты с расширением вниз, т.е. в сторону меньших адресов, используются для организации стеков, которые по ходу выполнения программы приходится расширять. Для относительно простых программ размеры сегментов обычно фиксированы и для стека можно использовать обычный сегмент данных (естественно, с разрешением как чтения, так и записи).

Бит 4 байта атрибутов 1 является идентификатором сегмента. Если он равен 1, дескриптор описывает сегмент памяти. Значение этого бита 0 характеризует дескриптор системного сегмента.

Поле DPL (Descriptor Privilege Level, уровень привилегий дескриптора) служит для защиты программ друг от друга. Уровень привилегий может принимать значения от 0 (максимальные привилегии) до 3 (минимальные). Программам операционной системы обычно назначается уровень 0, прикладным программам – уровень 3, в результате чего исключается возможность некорректным программам разрушить операционную систему. В наших примерах операционная система защищенного режима отсутствует, в некоторой степени программа сама выполняет функции операционной системы, поэтому всем сегментам наших программ назначается наивысший (нулевой) уровень привилегий, что открывает доступ ко всем средствам защищенного режима.

Бит P говорит о присутствии сегмента в памяти. В основном он используется в тех случаях, когда общий размер программы (или программ в случае многозадачного режима) превышает объем наличной памяти, и часть сегментов программ хранится на диске. Тогда операционная система защищенного режима с помощью этого бита определяет, находится ли требуемый сегмент в памяти, и при необходимости загружает его с диска. Перед выгрузкой ненужного сегмента на диск бит P сбрасывается. Младшая половина байта атрибутов 2 занята старшими битами границы сегмента. Бит AVL (от Available, доступный) не используется и не анализируется процессором и предназначен для использования прикладными программами.

Бит D (Default, умолчание) определяет действующий по умолчанию размер для операндов и адресов. Он изменяет характеристики сегментов двух типов: исполняемых и стека. Если бит D сегмента команд равен 0, в сегменте по умолчанию используются 16-битовые адреса и операнды, если 1 – 32-битовые.

Атрибут сегмента, действующий по умолчанию, можно изменить на противоположный с помощью префиксов замены размера операнда (66h) и замены размера адреса (67h). Таким образом, для сегмента с D=0 префикс 66h перед некоторой командой заставляет ее рассматривать свои операнды, как 32-битовые, а для сегмента с D=1 тот же префикс 66h, наоборот, сделает операнды 16-битовыми. В некоторых случаях транслятор сам включает в объектный модуль необходимые префиксы, в других случаях их приходится вводить в программу "вручную".

Поскольку наши программы работают в реальном режиме под управлением MS-DOS, естественно устанавливать D=0. Это, однако, не препятствует использованию в программе 32-битовых регистров. Если транслятор встречается с командой, в качестве операнда которой используется 32-разрядный регистр, он включает в код команды префикс замены размера операнда 66h. Поэтому команды с явным обращением к 32-битовым операндам транслируются правильно. В то же время при использовании команды iret в защищенном режиме префикс 66h приходится включать в программу в явном виде, чтобы процессор, выполняя эту команду, снял со стека не три слова, как обычно, а 6 слов (три двойных слова). Без префикса команда iret будет выполняться неправильно.

Рисунок 7.2 – Расшифровка значений атрибутов сегментов в программных примерах.

Префикс замены размера адреса 67h необходимо указывать, например, перед командой loop, если в качестве счетчика цикла используется не CX, а ECX. При отсутствии префикса команда loop (в сегменте, где по умолчанию используется 16-битовая адресация) будет выполнять цикл CX, а не ECX раз. Сегменты стека, адресуемые через регистр SS, с помощью бита D определяют, какой регистр использовать в качестве указателя стека в командах push и pop. Если D=0, используется регистр SP, если D=l, то ESP. Сегменты команд, предназначенные целиком для использования в защищенном режиме, транслируются с описателем размера адресов и операндов use32, а описывающие их дескрипторы должны иметь бит D=1.

Расшифровка значений атрибутов сегментов нашей программы представлена на рисунке 7.2.

Рассмотрим теперь формат дескрипторов, из которых строится таблица прерываний IDT, и которые носят название шлюзов. В таблицу дескрипторов прерываний могут входить шлюзы трех типов; ловушек, прерываний и задач. В рассмотренных ранее примерах таблица дескрипторов прерываний состояла из шлюзов прерываний и ловушек, описывающих обработчики аппаратных прерываний и исключений. Формат шлюза изображен на рисунке 7.3.

Если основной частью содержимого сегмента памяти был его линейный адрес, то в шлюзе вместо линейного адреса указывается полный трехсловный адрес обработчика, действующий в защищенном режиме и состоящий из селектора и смещения. Смещение имеет 32 бита и занимает в дескрипторе два поля – байты 0-1 и 6-7. Селектор имеет 16 бит и занимает байты 2-3. Байт 4 в шлюзах ловушек, прерываний и задач не используется.

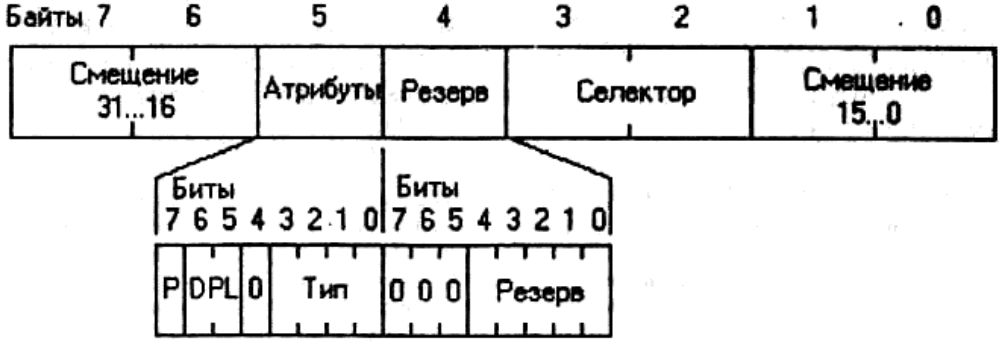

Рисунок 7.3 – Формат шлюзов, входящих в IDT

Байт атрибутов имеет такую же структуру, как и в дескрипторах памяти и включает тип, идентификатор дескриптора (бит 4), уровень привилегий дескриптора DPL и бит присутствия P. Тип дескриптора мо-

жет принимать 16 различных значений (табл. 7.2), хотя, как уже отмечалось, в IDT допустимо описывать только шлюзы задач, прерываний и ловушек.

Таблица 7.2. Значения поля типа для системных дескрипторов и шлюзов

Тип		Назначение дескриптора
0	0000	Не определено (Зарезервированный тип)
1	0001	Свободный сегмент состояния задачи (TSS) 80286 (16-битный)
2	0010	Дескриптор таблицы LDT
3	0011	Занятый сегмент состояния задачи (TSS) 80286 (16-битный)
4	0100	Шлюз вызова 80286 (16-битный)
5	0101	Шлюз задачи
6	0110	Шлюз прерываний 80286 (16-битный)
7	0111	Шлюз ловушки 80286 (16-битный)
8	1000	Не определено (Зарезервированный тип)
9	1001	Свободный сегмент состояния задачи (TSS) 386/486 (32-битный)
Ah	1010	Не определено (Зарезервированный тип)
Bh	1011	Занятый сегмент состояния задачи (TSS) 80386/486 (32-битный)
Ch	1100	Шлюз вызова 80386 и 486 (32-битный)
Dh	1101	Не определено (Зарезервированный тип)
Eh	1110	Шлюз прерываний 80386 и 486 (32-битный)
Fh	1111	Шлюз ловушки 80386 и 486 (32-битный)

В примере 7.1 в расширенной памяти создается сегмент размером 2 Мбайт, который заполняется натуральным рядом чисел (512К четырехбайтовых чисел) с визуальным контролем заполнения.

Пример 7.1 – Работа с расширенной памятью

```
;Работа с расширенной памятью
IDEAL
P386
model small
stack 100h
;Макрос для отладки
macro debug
```

```
  push ax
  push bx
  push cx
  push dx
  push ax
  and  ax, 0F000h
  shr  ax, 12
  mov  bx, offset tbl_hex
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0F00h
  shr  ax, 8
  inc  si
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0F0h
  shr  ax, 4
  inc  si
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0Fh
  inc  si
  xlat
  mov  [si], al
  pop  ax
  pop  dx
  pop  cx
  pop  bx
  pop  ax
endm
struc descr
      limit   dw 0
      base_l  dw 0
      base_m  db 0
      attr_1  db 0
      attr_2  db 0
      base_h  db 0
ends descr
;Структура для шлюзов ловушки
struc trap            ; (1)
```

```
        offs_l  dw 0    ; (2)
        sel     dw 16   ; (3) Селектор сегмента кода
        rsrv    db 0    ; (4)
        attr    db 8Fh  ; (5) 8F == 32-битный шлюз ловушки, P=1, DPL=0
        offs_h  dw 0    ; (6)
ends trap

DATASEG
   gdt_null     descr <0,0,0,0,0,0>           ; Селектор = 0
   gdt_data     descr <data_size-1,0,0,92h,0,0> ; Селектор = 8
   gdt_code     descr <code_size-1,0,0,98h,0,0> ; Селектор = 16
   gdt_stack    descr <100h-1,0,0,92h,0,0>      ; Селектор = 24
   gdt_screen   descr <4095,8000h,0Bh,92h,0,0>  ; Селектор = 32
   gdt_himem    descr <511,0,10h,92h,80h,0>     ; (7) Селектор = 40
  gdt_size = $-gdt_null
   ;Таблица прерываний
   idt  trap 10 dup (<dummy_exc>)             ; (8)
        trap <exc_0a>                         ; (9)
        trap <exc_0b>                         ; (10)
        trap <exc_0c>                         ; (11)
        trap <exc_0d>                         ; (12)
        trap <exc_0e>                         ; (13)
        trap 17 dup (<dummy_exc>)             ; (14)
  idt_size = $-idt                            ; (15)
   ;Содержимое регистра IDTR в реальном режиме
   idtr_real    dw 3FFh, 0, 0                 ; (16)
 ;Поля данных программы
   pdescr       dp 0
   mes          db 10,13,'Real mode','$'
   tbl_hex      db '0123456789ABCDEF'                ; (17)
   number       db '???? ????'                       ; (18)
   string       db '**** **** **** **** **** **** ****' ; (19)
  len = $-string                                    ; (20)
   home_sel     dw home                 ; (21) Точка выхода
                dw 10h                  ; (22) Селектор сегмента кода
  data_size = $-gdt_null
ends
CODESEG
assume cs: @code, ds:@data
sttt equ $
;Обработчик исключений с номерами 0-9 и 0F-1F
proc dummy_exc
   pop  eax
   pop  eax
   mov  si, offset string+5
   debug
```

```
  mov  ax, 1111h
  jmp  [dword ptr home_sel]
endp
proc exc_0a ;Обработчик исключения 0A
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Ah
  jmp  [dword ptr home_sel]
endp
proc exc_0b ;Обработчик исключения 0B
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Bh
  jmp  [dword ptr home_sel]
endp
proc exc_0c ;Обработчик исключения 0C
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Ch
  jmp  [dword ptr home_sel]
endp
proc exc_0d ;Обработчик исключения 0D
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Dh
  jmp  [dword ptr home_sel]
endp
proc exc_0e ;Обработчик исключения 0E
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Eh
  jmp  [dword ptr home_sel]
endp
start:
  xor  eax, eax
  mov  ax, @data
```

```
mov  ds, ax
shl  eax, 4
mov  ebp, eax
mov  bx, offset gdt_data
mov  [(descr ptr bx).base_l], ax
rol  eax, 16
mov  [(descr ptr bx).base_m], al
xor  eax, eax
mov  ax, cs
shl  eax, 4
mov  bx, offset gdt_code
mov  [(descr ptr bx).base_l], ax
rol  eax, 16
mov  [(descr ptr bx).base_m], al
xor  eax, eax
mov  ax, ss
shl  eax, 4
mov  bx, offset gdt_stack
mov  [(descr ptr bx).base_l], ax
rol  eax, 16
mov  [(descr ptr bx).base_m], al
;Подготовка к загрузке GDTR
mov  [dword ptr pdescr+2], ebp
mov  [word ptr pdescr], gdt_size-1
lgdt [pword ptr pdescr]
; Запрет аппаратных прерываний и NMI
cli
in   al, 70h
or   al, 80h
out  70h, al
;Подготовка к загрузке IDTR
mov  [word ptr pdescr], idt_size-1   ; (23) Граница IDT
xor  eax, eax                        ; (24)
mov  ax, offset idt                  ; (25)
add  eax, ebp                        ; (26) Вычисляем адрес начала IDT
mov  [dword ptr pdescr+2], eax       ; (27)
lidt [pword ptr pdescr]              ; (28)
;Открыть линию A20
mov  al, 0D1h                        ; (29)
out  64h, al                         ; (30)
mov  al, 0DFh                        ; (31)
out  60h, al                         ; (32)
;Переход в защищенный режим
mov  eax, CR0
or   eax, 1
mov  CR0, eax
```

```
        db 0EAh
        dw offset continue
        dw 16
continue:
  mov  ax, 8
  mov  ds, ax
  mov  ax, 24
  mov  ss, ax
  mov  ax, 32
  mov  es, ax
  ;Инициализируем GS селектором сегмента в расширенной памяти
  mov  ax, 40                    ; (33)
  mov  gs, ax                    ; (34)
  ;Заполнение 2Мб сегмента в расширенной памяти
  mov  eax, 0                  ; (35)
  mov  ebx, 0                  ; (36)
  mov  ecx, 80000h               ; (37) Количество зписываемых чисел
fill:                          ; (38)
  mov  [dword ptr gs:ebx], eax       ; (39) Запись числа
  push eax                       ; (40)
  push cx                        ; (41)
  ;Подготовка текущее число для вывода на экран
  mov  si, offset number+5           ; (41)
  debug                          ; (42)
  shr  eax, 16                   ; (43)
  mov  si, offset number             ; (44)
  debug                          ; (45)
  mov  si, offset number             ; (46)
  mov  cx, 9                     ; (47) Количество символов
  mov  ah, 43h                   ; (48) Атрибут цвета
  mov  di, 1040                  ; (49) Позиция для вывода
  ;Вывод на экран
scrh:                          ; (50)
  lodsb                          ; (51)
  stosw                          ; (52)
  loop scrh                      ; (53)
  pop  cx                        ; (54)
  pop  eax                       ; (55)
  add  ebx, 4                    ; (56)
  inc  eax                       ; (57)
  db 67h                         ; (58) Использовать ECX
  loop fill_1                    ; (59)
  jmp  go                        ; (60)
fill_1:                        ; (61)
  jmp  fill                      ; (62)
```

```
  ;Контрольное чтение расширенной памяти и вывод на экран
go:                             ; (63)
  mov  ebx, 0                   ; (64) Чтение первого слова
  mov  eax, [dword ptr gs:ebx]  ; (65)
  mov  si, offset string+15     ; (66)
  debug                         ; (67)
  shr  eax, 16                  ; (68)
  mov  si, offset string+10     ; (69)
  debug                         ; (70)
  mov  ebx, 2FFFFCh             ; (71) Чтение последнего слова
  mov  eax, [dword ptr gs:ebx]  ; (72)
  mov  si, offset string+25     ; (73)
  debug                         ; (74)
  shr  eax, 16                  ; (75)
  mov  si, offset string+20     ; (76)
  debug                         ; (77)
  mov  ax, 0FFFFh               ; (78)
home:                           ; (79)
  mov  si, offset string        ; (80)
  debug                         ; (81)
  mov  si, offset string        ; (82)
  mov  cx, len                  ; (83)
  mov  ah, 74h                  ; (84)
  mov  di, 1600                 ; (85)
scr:                            ; (86)
  lodsb                         ; (87)
  stosw                         ; (88)
  loop scr                      ; (89)
  ;Закрыть линию A20
  mov  al, 0D1h                 ; (90)
  out  64h ,al                  ; (91)
  mov  al, 0DDh                 ; (92)
  out  60h, al                  ; (93)
  ; Возврат в реальный ражим
  mov  eax, CR0
  and  al, 0FEh
  mov  CR0, eax
     db 0EAh
     dw offset return
     dw @code
return:
  ;Восстановим операционную среду реального режима
  mov  ax, @data
  mov  ds, ax
```

```
  mov  ax, @stack
  mov  ss, ax
  mov  sp, 100h
  ;Восстановим значение IDTR для работы в реальном режиме
  lidt [fword ptr idtr_real]          ; (94)
  ;Разрешим аппаратные и немаскируемые прерывания
  sti
  in   al, 70h
  and  al, 07Fh
  out  70h, al
  mov  ah, 09h
  mov  dx, offset mes
  int  21h
  mov  ax, 4C00h
  int  21h
ends
code_size=$-sttt
end start
end
```

Желая образовать, дополнительный сегмент объемом 2 Мбайт в расширенной памяти, необходимо предусмотреть описывающий его дескриптор в таблице глобальных дескрипторов:

gdt_himem descr <511, 0,10h, 92h, 80h, >; (9) Селектор 40

Будучи расположен в таблице GDT на шестом месте, этот дескриптор получает индекс 5, что соответствует селектору 40.

Линейный адрес первого байта расширенной памяти равен 100000h. Из этого адреса младшие 16 бит (0000h) записываются в поле base_l, следующие 8 бит, содержащие 10h, - в поле base_m, и старшие 8 бит (00h) - в поле base_h.

Поскольку размер сегмента превышает 1 Мбайт, его границу следует указывать в блоках по 4 Кбайт. Поэтому мы устанавливаем в 1 бит дробности G в байте атрибутов 2, который, таким образом, принимает значение 80h. Размер сегмента (2 Мбайт) составляет 512 блоков по 4 Кбайт, поэтому значение границы составляет 511. Байт атрибутов 1, как и для других сегментов памяти, принимает значение 92h (для чтения и записи, присутствует).

В поля данных программы включена символьная строка **number**, в которую по мере заполнения расширенной памяти числами будет выводиться текущее число. Это даст возможность контролировать ход заполнения памяти и заодно проверить правильность содержимого последней заполненной ячейки.

Перед переходом в защищенный режим (или после перехода в него) следует открыть линию A20, т.е. адресную линию, на которой устанавливается единичный уровень сигнала, если происходит обращение к мегабайтам адресного пространства с номерами 1, 3, 5 и т.д. (первый мегабайт имеет номер 0). В реальном режиме линия A20 заблокирована, и если значение адреса выходит за пределы FFFFFh, выполняется его циклическое оборачивание (линейный адрес 100000h превращается в 00000h, адрес 100001h - .в 00001h и т.д.). Открытие (разблокирование) линии A20 выключает механизм циклического оборачивания адреса, что позволяет адресоваться к расширенной памяти. Управление блокированием линии A20 осуществляется через порт 64h, куда сначала следует послать команду D1h управления линией A20, а затем - код открытия (DFh). Эти действия выполняются в предложениях 29…32.

Переход в защищенный режим и действия по инициализации сегментных регистров выполняются обычным образом.

Заполнение расширенной памяти начинается с предложения 33. В свободный сегментный регистр GS заносится селектор сегмента в расширенной памяти и инициализируются регистры EAX, EBX и ECX (число шагов в цикле превышает 64 К, поэтому требуется 32-битовый регистр ECX). В предложении 39 число-заполнитель отправляется в расширенную память. Для динамического контроля хода заполнения памяти в каждом шаге цикла очередное число выводится на экран (предложения 50…53). Далее выполняется смещение индекса в указателе, инкремент числа-заполнителя и возврат в начало цикла.

Для того чтобы команда loop использовала в своей работе регистр ECX, а не CX (вспомним, что мы установили для сегмента команд D=0 и назначили тем самым по умолчанию 16-битовые адреса и операнды), перед ней в программу включен код префикса замены размера адреса (предложение 58). Другая сложность возникла из-за того, что в цикле оказалось больше 128 байтов, а команда loop может осуществлять только короткие переходы в диапазоне +127…-128 байтов. Поэтому командой loop (предложение 59) выполняется переход не на начало цикла, а на вспомогательную строку fill_l, откуда командой безусловного перехода управление может быть передано уже в любую точку программы.

Команда безусловного перехода jmp в предложении 60 позволяет обойти предложение 62 после завершения цикла.

В программе предусмотрено контрольное чтение расширенной памяти и вывод в диагностическую строку string прочитанных чисел. В предложениях 64..70 читается и выводится первое 4-байтовое слово из сегмента расширенной памяти (0000 0000h), в предложениях 71…73 –

последнее 4-байтовое слово двухмегабайтного сегмента (0007 FFFFh). Эта проверка может оказаться весьма полезной. Действительно, если программа из-за неправильно написанных строк цикла обратится к памяти за пределами объявленного в таблице, глобальных дескрипторов сегмента, возникнет исключение общей защиты. Однако процессор не реагирует на отсутствие физической памяти по указанному в команде адресу. Поэтому если попытаться заполнить больше памяти, чем есть в компьютере (предварительно установив соответственно границу сегмента в дескрипторе), то программа будет работать так, как если бы эта память имелась, хотя, конечно, реально числа записываться в отсутствующую память не будут. Чтение из памяти последнего записанного в нее числа позволит убедиться в том, что это число действительно записано в память.

Перед переходом в реальный режим следует закрыть линию A20, для чего в порт 64h посылается команда D1h управления линией A20, а затем – код закрытия линии (DDh). Эти действия выполняются в предложениях 90…93. Завершение программы выполняется так же, как и в предыдущей работе, за одним небольшим исключением.

После возврата в реальный режим необходимо восстановить содержимое регистра IDTR для работы в реальном режиме (предложение 94) – адрес 0 и размер 4 * 256, соответствующие таблице векторов прерываний реального режима. Для этого объявлена переменная idtr_real (предложение 16), заполненная соответственными данными и представляющая собой псевдодескриптор таблицы прерываний реального режима.

Можно модифицировать программу, увеличив или уменьшив размер сегмента в расширенной памяти и, соответственно, число шагов в цикле заполнения, настроив ее под конкретную конфигурацию своего компьютера.

3 Задание для самостоятельного выполнения

1. Изучить теоретические основы работы с расширенной памятью.
2. Набрать и отладить предложенную программу. При этом обратить внимание на следующие этапы:
   - 2.1. Определение структуры для шлюзов ловушки.
   - 2.2. Структура и применение макроса для отладки
   - 2.3. Определение таблицы дескрипторов и дескрипторов используемых сегментов.
   - 2.4. Определение таблицы прерываний и используемых шлюзов ловушки

2.5. Содержимое регистра IDTR в реальном режим
2.6. Определение точки выхода из обработчиков исключений
2.7. Структура и содержимое обработчиков исключений
2.8. Подготовка таблицы дескрипторов к переходу в защищенный режим.
2.9. Подготовка псевдодескриптора и инициализация регистра IDTR
2.10. Открытие линии A20
2.11. Настройка работы с расширенной памяти
2.12. Заполнение сегмента в расширенной памяти
2.13. Проверка записанных значений в расширенной памяти и вывод их на экран
2.14. Закрытие линии A20
2.15. Подготовка к возврату и возврат в реальный режим
2.16. Восстановление значения IDTR для работы в реальном режиме

3. Модифицировать программу, увеличив или уменьшив размер сегмента.
4. Отладить и протестировать полученную программу.
5. Оформить отчёт.

4 Контрольные вопросы

1. Каков формат дескриптора сегмента памяти?
2. Какие атрибуты входят в дескриптор сегмента памяти?
3. Каково назначение атрибутов дескриптора сегмента памяти.
4. Какие различают сегменты памяти? Чем отличаются их дескрипторы?
5. Чем отличается формат дескриптора шлюза задачи?

Лабораторная работа №8. Переключение задач в защищённом режиме

1 Цель и порядок работы

Цель работы – изучить процесс переключения задач и научиться использовать его при разработке программ.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя по вариантам;
- написать программу, ввести программу, отладить и решить ее на ЭВМ;
- оформить отчет.

2 Краткая теория

Важнейшей особенностью защищенного режима является возможность параллельного выполнения нескольких вычислительных задач с надежной защитой их друг от друга. Использование таблиц локальных дескрипторов позволяет разнести физические адресные пространства отдельных задач без права их обращения к "чужой" памяти, а система привилегий, встроенная в архитектуру процессора, предотвращает несанкционированный вызов задачами защищенных процедур (например, программ операционной системы) или обращение к устройствам ввода-вывода. Однако организация многозадачного режима не обязательно предполагает использование механизмов защиты по привилегиям. В простых и достаточно распространенных случаях параллельное выполнение нескольких задач осуществляется на одном уровне привилегий, защита же адресных пространств или процедур осуществляется путем аккуратного написания программ комплекса.

Под задачей понимают выполняемую на компьютере программу или ее фрагмент, имеющий относительно самостоятельное значение. В частности, задачей можно назвать всю сколь угодно сложную программу, и тогда мультизадачность обозначает параллельное выполнение нескольких, возможно не связанных друг с другом программ. Однако задачей может быть и достаточно самостоятельная часть программы. Например, обработчики аппаратных прерываний уместно назвать задачами: для них как раз характерно выполнение параллельно и независимо от основной части программы. Вообще ничто не мешает включить в состав одной программы несколько самостоятельных участков, переключение между которыми может осуществляться периодически или по каким-то событи-

ям: нажатиям определенных клавиш, прерываниям от диска и т.д. Эти фрагменты могут образовывать отдельные сегменты команд, но могут входить в состав единого программного сегмента в виде процедур.

Организация многозадачного режима, опирается на следующие аппаратные и программные средства:

- сегмент состояния задачи (Task State Segment, TSS);
- дескриптор сегмента состояния задачи;
- регистр задачи (Task Register, TR);
- дескриптор шлюза задачи (Task gate).

Сегмент состояния задачи (TSS) представляет собой поле данных, либо включаемое в состав сегмента данных программы, либо образующее отдельный сегмент небольшого размера. Каждая задача, участвующая в процедуре переключения, должна иметь свой TSS; именно наличие TSS делает данный программный объект задачей. Сегменты состояния задач служат для хранения, при переключениях задач, их текущих контекстов, т.е. содержимого аппаратных регистров процессора и другой важной информации. Поскольку TSS представляется процессору отдельным сегментом, ему должен соответствовать дескриптор.

В отличие от обычных сегментов данных, TSS описывается не дескриптором сегмента памяти, а системным дескриптором, который к тому же может находиться только в таблице глобальных дескрипторов. Системные дескрипторы имеют тот же формат, что и дескрипторы памяти, отличаясь только кодом типа. Для процессоров 386 и 486 дескрипторы TSS имеют код 9. На рисунке 6.1 приведен формат дескриптора TSS для процессора 486.

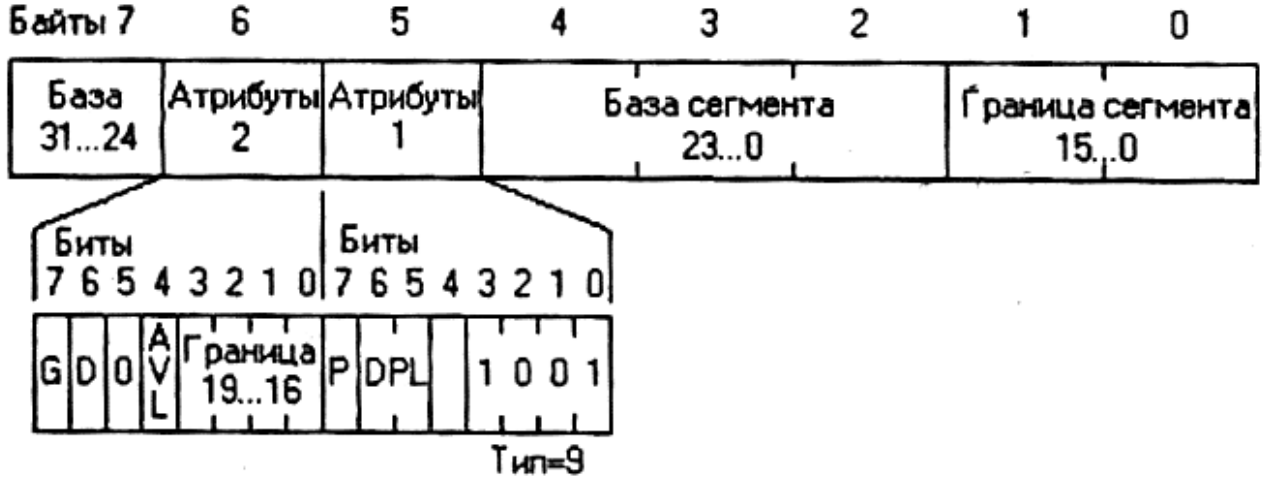

Рисунок 6.1 – Формат системного дескриптора, описывающего TSS для МП 386 и 486.

В зависимости от геометрического места расположения дескриптора TSS в таблице дескрипторов, ему соответствует тот или иной селектор. Селектор TSS текущей (активной) задачи должен быть загружен в регистр задачи TR. Для исходной, главной задачи эта загрузка осуществляется программно с помощью предназначенной для этого команды ltr (load task register, загрузка регистра задач); при переключении на новую задачу программа передает процессору селектор нового TSS и перезагрузку регистра TR осуществляет процессор в ходе переключения задач.

Переключение на новую задачу осуществляется командой дальнего вызова call dword ptr или, в некоторых случаях, дальнего перехода jmp dword ptr. В качестве аргумента этих команд указывается двухсловное поле, в первом слове которого записывается селектор требуемого TSS. Существует и другой способ переключения – не через селектор TSS, а через шлюз задачи (дескриптор шлюза с кодом типа, равным 5). В этом случае селектор требуемого TSS указывается в поле для селектора в шлюзе задачи. Переключение через шлюз задачи имеет то преимущество, что его можно выполнить по аппаратному прерыванию, так как, в отличие от дескриптора сегмента состояния задачи TSS, шлюз задачи можно расположить в таблице дескрипторов прерываний.

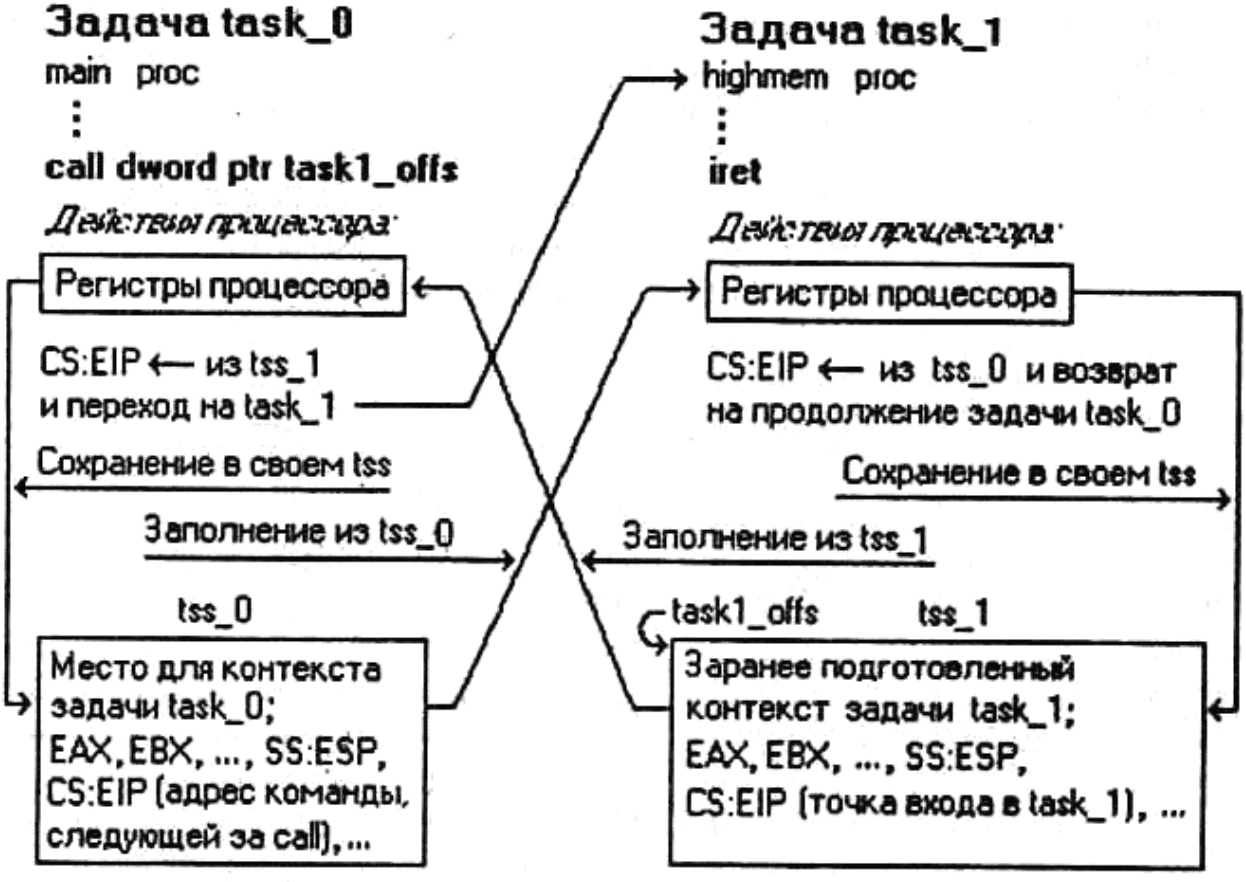

Рисунок 6.2 – Основные действия при переключении задач

В процессе переключения на новую задачу (рис. 6.2) процессор сохраняет контекст текущей задачи в ее TSS и загружает новый контекст (включая селектор и относительный адрес точки входа в задачу) из TSS новой задачи.

Задача, на которую осуществляется переключение, должна в этом случае завершаться командой iret, которая обрабатывается процессором совсем не так, как iret обычного обработчика прерывания. В случае переключения задач процессор по команде iret выполняет обратное перемещение контекстов – в контекст завершающейся задачи (на момент ее завершения) сохраняется все TSS, а в регистры процессора из TSS исходной задачи загружается сохраненный там контекст, соответствующий моменту переключения на вторую задачу. Таким образом, TSS можно рассматривать, как функциональный аналог стека, который тоже используется для сохранения содержимого регистров и другой информации в обработчиках прерываний и подпрограммах. Однако сохранение в стеке и восстановление из него выполняются с помощью последовательностей команд процессора, а сохранение и восстановление контекстов с помощью TSS осуществляется процессором аппаратно в процессе переключения задач.

Поля TSS исходной задачи заполняются процессором при первом переключении на новую задачу (чтобы можно было вернуться в исходную); программист может не заботиться о его содержимом. Другое дело TSS задачи, на которую осуществляется переключение. В TSS содержится такая важная для выполнения задачи информация, как адрес точки входа, а также исходное содержимое сегментных регистров и регистров общего назначения. По крайней мере, некоторые из этих полей должны быть заполнены в исходной задаче еще перед переключением на новую. Рассмотрим кратко содержимое TSS (рис. 6.3).

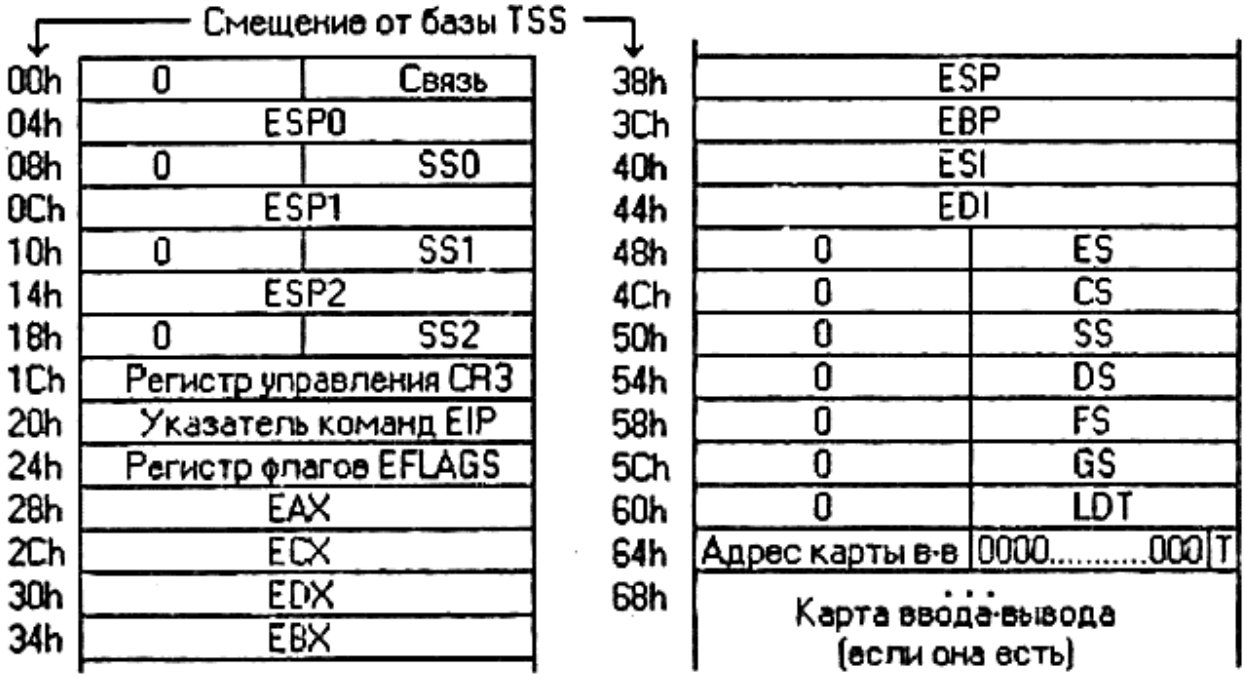

Рисунок 6.3 – Формат сегмента состояния задачи

Сегмент состояния задачи имеет размер минимум 104 байт. В самом начале TSS предусмотрено 16-битовое поле связи, используемое при переключении задач. Для исходной, главной задачи (task_0) его содержимое не имеет значения. При переключении на новую задачу (task_l) процессор заносит в поле связи TSS новой задачи селектор TSS исходной задачи, чем создается связь между новой и старой задачами. Если новая задача, в свою очередь, переключается на следующую задачу (task_2), в поле связи TSS этой следующей задачи процессор заносит селектор TSS предыдущей задачи и т.д. В результате создается связный список вложенных задач (рис. 6.4).

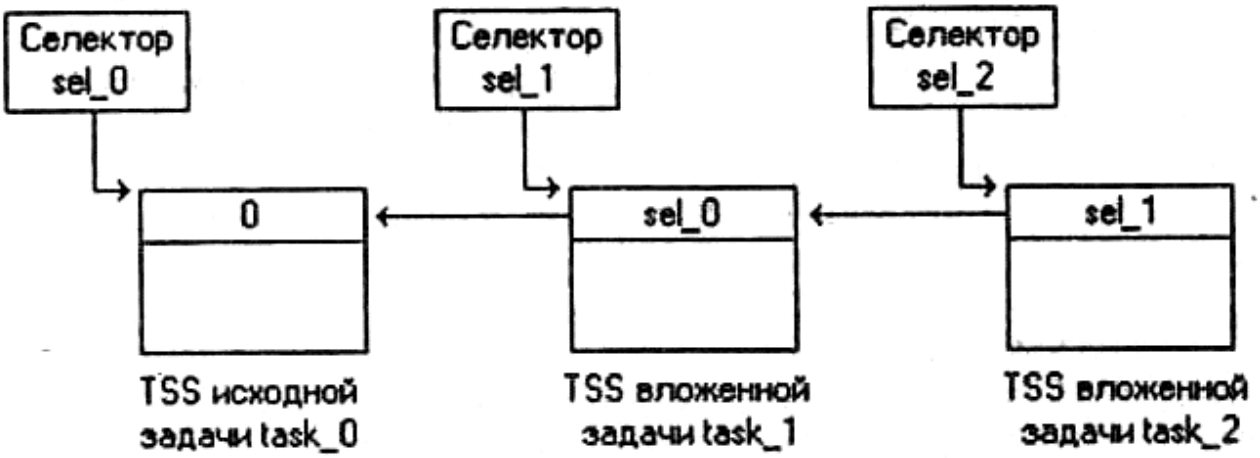

Рисунок 6.4 – Цепочка связанных TSS

Команда iret, которой должна завершаться каждая вложенная задача, выполняет обратное переключение задач. В процессе этой операции процессор извлекает из поля связи TSS последней вложенной задачи селектор TSS предыдущей по порядку вложенности задачи и передаст ей управление, восстановив перед этим из TSS этой задачи ее контекст.

По адресам 04h, 0Ch и 14h относительно базы TSS располагаются кадры стеков уровней привилегий 0, 1 и 2. Содержимое этих полей загружается в регистры SS и ESP, если при переключении задачи происходит смена уровня привилегий. Задачи примеров будут работать на одном (нулевом) уровне привилегий; в этом случае поля кадров стеков не используются, а регистры SS и ESP инициализируются содержимым ячеек TSS с адресами 50h и 38h.

Регистр управления CR3 (поле TSS с адресом 1Ch) содержит базовый физический адрес каталога страниц и используется, если включено страничное преобразование. Наличие в TSS поля для регистра CR3 позволяет иметь для каждой задачи свой каталог страниц и, соответственно, свои таблицы отображения виртуальных страниц программы на физические адреса памяти. В примерах страничное преобразование выключено и регистр CR3 не используется.

Двухсловное поле TSS с адресом 20h предназначено для хранения значения указателя команд EIP. В TSS исходной задачи это поле при переключении на новую задачу заполняется процессором адресом команды, следующей за командой переключения, т.е. адресом возврата. В TSS задачи, на которую планируется осуществить переключение, поле для EIP должно быть заполнено программно смещением точки входа в задачу. При этом следует иметь в виду, что при возврате в старую задачу командой iret процессор записывает в поле для EIP завершившейся задачи

в качестве "адреса возврата" адрес команды, следующей iret, т.е. адрес уже за пределами задачи. Если планируется повторное переключение на эту задачу, перед каждым переключением необходимо восстанавливать в TSS этой задачи адрес се точки входа.

Сохранение в TSS исходной задачи текущего содержимого регистра флагов EFLAGS (по адресу 24h) позволяет осуществлять переключение в любой точке задачи без потери ее работоспособности.

Участок TSS с адресами 28h…47h отведен для хранения содержимого регистров общего назначения. При переключении с исходной задачи на новую задачу процессор сохраняет в этих полях TSS исходной задачи текущее содержимое регистров, а при обратном переключении командой iret восстанавливает регистры из этих полей TSS исходной задачи, обеспечивая правильное продолжение ее выполнения. Что же касается TSS новой задачи, то, заполнив заранее поля регистров в TSS новой задачи, можно передать ей исходные параметры.

Младшие половины шести двухсловных полей, начиная с адреса 48h, отведены под содержимое сегментных регистров. Эти поля используются точно так же, как и поля регистров общего назначения.

Если новая задача использует таблицу локальных дескрипторов, ее селектор следует занести в TSS по адресу 60h.

Бит 0 слова по адресу 64h используется для отладки переключаемых задач. Если в TSS новой задачи установлен этот бит, то сразу же после переключения генерируется исключение отладки с номером 1. Остальные биты слова по адресу 64h должны быть равны 0.

Слово с адресом 66h содержит смещение битовой карты ввода-вывода, которая, при ее наличии, располагается в TSS по последующим адресам и используется для защиты портов компьютера от несанкционированного доступа. Каждый бит этой карты соответствует одному порту (вся карта, таким образом, может достигать 64 Кбит, или 8 Кбайт). Если бит, закрепленный за некоторым портом, равен 0, задача любого уровня привилегий может обращаться к этому порту. Если бит равен 1, то при обращении к порту задачей с недостаточно высоким уровнем привилегий генерируется исключение общей защиты.

При переключении на новую задачу команда iret должна инициировать довольно сложную процедуру обратного переключения через селектор TSS, хранящийся в поле связи TSS текущей задачи. Однако в обработчиках прерываний и исключений та же команда iret выполняется иначе: она просто снимает со стека три 32-битовых слова (EFLAGS, CS и EIP) и загружает их в соответствующие регистры, обеспечивая возврат из обработчика в прерванную задачу.

Режим выполнения команды iret определяется состоянием специального флага NT (Nested Task, вложенная задача), расположенного в бите 14 регистра флагов EFLAGS. Команда iret анализирует состояние флага NT и, если он сброшен, осуществляет обычный возврат из программы обработки прерывания (через стек); если же флаг NT установлен, команда iret инициирует обратное переключение задач через селектор в TSS.

После загрузки компьютера флаг NT находится в установленном состоянии. Однако любое аппаратное прерывание или исключение сбрасывает этот флаг, в результате чего команда iret, завершающая обработчик, выполняется в "облегченном" варианте. То же происходит при выполнении процессором команды программного прерывания int. Поскольку команда iret восстанавливает исходное состояние регистра флагов, после завершения обработчика флаг NT снова оказывается установленным (если, конечно, он не был явным образом сброшен выполняемой программой).

При выполнении процедуры переключения на новую задачу через шлюз задачи или непосредственно через TSS, процессор сохраняет в TSS текущей задачи слово флагов и устанавливает в регистре флагов бит NT (независимо от того, был ли он перед этим сброшен или установлен). Команда iret, завершающая задачу, обнаруживает NT=1 и, вместо осуществления возврата через стек, инициирует механизм обратного переключения задач.

Если вложенная задача, в свою очередь, выполняет переключение на следующую задачу, текущее слово флагов с установленным битом NT сохраняется в TSS текущей задачи. После завершения новой задачи это слово будет возвращено в регистр флагов и, таким образом, задача будет продолжаться с NT=1, что обеспечит се правильное завершение.

Воспользуемся примером предыдущей лабораторной для изучения техники переключения задач. Для этого выделим строки заполнения расширенной памяти в отдельную процедуру и оформим ее в качестве задачи. Выполним в главной программе переключение на эту задачу и возврат из нее.

Пример 6.1 – Техника переключения задач

```
; Техника переключения задач
IDEAL
P386
model small
stack 100h
;Макрос для отладки
macro debug
```

```
  push ax
  push bx
  push cx
  push dx
  push ax
  and  ax, 0F000h
  shr  ax, 12
  mov  bx, offset tbl_hex
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0F00h
  shr  ax, 8
  inc  si
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0F0h
  shr  ax, 4
  inc  si
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0Fh
  inc  si
  xlat
  mov  [si], al
  pop  ax
  pop  dx
  pop  cx
  pop  bx
  pop  ax
endm
struc descr
      limit   dw 0
      base_l  dw 0
      base_m  db 0
      attr_1  db 0
      attr_2  db 0
      base_h  db 0
ends descr
;Структура для шлюзов ловушки
struc trap
```

```
        offs_l  dw 0
        sel     dw 16
        rsrv    db 0
        attr    db 8Fh
        offs_h  dw 0
ends trap
DATASEG
   gdt_null     descr <0,0,0,0,0,0>           ; Селектор = 0
   gdt_data     descr <data_size-1,0,0,92h,0,0> ; Селектор = 8
   gdt_code     descr <code_size-1,0,0,98h,0,0> ; Селектор = 16
   gdt_stack    descr <100h-1,0,0,92h,0,0>      ; Селектор = 24
   gdt_screen   descr <4095,8000h,0Bh,92h,0,0>  ; Селектор = 32
   gdt_himem    descr <511,0,10h,92h,80h,0>     ; Селектор = 40
   gdt_tss_0    descr <103,0,0,89h,0,0>         ; (1) Селектор = 48
   gdt_tss_1    descr <103,0,0,89h,0,0>         ; (2) Селектор = 56
  gdt_size = $-gdt_null
   idt  trap 10 dup (<dummy_exc>)
        trap <exc_0a>
        trap <exc_0b>
        trap <exc_0c>
        trap <exc_0d>
        trap <exc_0e>
        trap 17 dup (<dummy_exc>)
  idt_size = $-idt
   idtr_real    dw 3FFh, 0, 0
   pdescr       dp 0
   mes          db 10,13,'Real mode','$'
   tbl_hex      db '0123456789ABCDEF'
   number       db '???? ????'
   string       db '**** **** **** **** **** **** ****'
  len = $-string
   home_sel     dw home
                dw 10h
   color        db 43h                ; (3) Атрибут символов для заполнения
   tss_0        db 104 dup(0)         ; (4) TSS 0-й задачи
   tss_1        db 104 dup(0)         ; (5) TSS 1-й задачи
   ;Переменная для переключения на задачу 1
   task1_offs   dw 0                  ; (6) Смещение 1-й задачи
                dw 56                 ; (7) Селектор TSS 1 = 56
  data_size = $-gdt_null
ends
CODESEG
assume cs: @code, ds:@data
sttt equ $
proc dummy_exc ;Обработчик исключений с номерами 0-9 и 0F-1F
   pop  eax
```

```
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 1111h
  jmp  [dword ptr home_sel]
endp
proc exc_0a ;Обработчик исключения 0A
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Ah
  jmp  [dword ptr home_sel]
endp
proc exc_0b ;Обработчик исключения 0B
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Bh
  jmp  [dword ptr home_sel]
endp
proc exc_0c ;Обработчик исключения 0C
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Ch
  jmp  [dword ptr home_sel]
endp
proc exc_0d ;Обработчик исключения 0D
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Dh
  jmp  [dword ptr home_sel]
endp
proc exc_0e ;Обработчик исключения 0E
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Eh
  jmp  [dword ptr home_sel]
endp
```

```
start:
  xor  eax, eax
  mov  ax, @data
  mov  ds, ax
  shl  eax, 4
  mov  ebp, eax
  mov  bx, offset gdt_data
  mov  [(descr ptr bx).base_l], ax
  rol  eax, 16
  mov  [(descr ptr bx).base_m], al
  xor  eax, eax
  mov  ax, cs
  shl  eax, 4
  mov  bx, offset gdt_code
  mov  [(descr ptr bx).base_l], ax
  rol  eax, 16
  mov  [(descr ptr bx).base_m], al
  xor  eax, eax
  mov  ax, ss
  shl  eax, 4
  mov  bx, offset gdt_stack
  mov  [(descr ptr bx).base_l], ax
  rol  eax, 16
  mov  [(descr ptr bx).base_m], al
  ;Вычислим 32-битовый линейный адрес сегмента TSS 0 и загрузим его
  ;в дескриптор сегмента TSS 0 в таблице глобальных дескрипторов
  mov  eax, ebp                    ; (8) Адрес начала сегмента данных
  add  ax, offset tss_0            ; (9) Добавим смещение TSS0
  mov  bx, offset gdt_tss_0        ; (10)
  mov  [(descr ptr bx).base_l], ax    ; (11)
  rol  eax, 16                     ; (12)
  mov  [(descr ptr bx).base_m], al    ; (13)
  ;Вычислим 32-битовый линейный адрес сегмента TSS 1 и загрузим его
  ;в дескриптор сегмента TSS 1 в таблице глобальных дескрипторов
  mov  eax, ebp                ; (14) Адрес начала сегмента данных
  add  ax, offset tss_1            ; (15) Добавим смещение TSS1
  mov  bx, offset gdt_tss_1        ; (16)
  mov  [(descr ptr bx).base_l], ax    ; (17)
  rol  eax, 16                     ; (18)
  mov  [(descr ptr bx).base_m], al    ; (19)
  ;Подготовка к загрузке GDTR
  mov  [dword ptr pdescr+2], ebp
  mov  [word ptr pdescr], gdt_size-1
  lgdt [pword ptr pdescr]
  ; Заполним поля TSS 1, TSS 0 заполнится автоматически
  mov  [word ptr tss_1+4Ch], 16            ; (20) CS
```

```
  mov  [word ptr tss_1+20h], offset highmem    ; (21) IP
  mov  [word ptr tss_1+50h], 24                ; (22) SS
  mov  [word ptr tss_1+38h], 128               ; (23) SP
  mov  [word ptr tss_1+54h], 8                 ; (24) DS
  mov  [word ptr tss_1+48h], 32                ; (25) ES
  ; Запрет аппаратных прерываний и NMI
  cli
  in   al, 70h
  or   al, 80h
  out  70h, al
  ;Подготовка к загрузке IDTR
  mov  [word ptr pdescr], idt_size-1
  xor  eax, eax
  mov  ax, offset idt
  add  eax, ebp
  mov  [dword ptr pdescr+2], eax
  lidt [pword ptr pdescr]
  ;Открыть линию A20
  mov  al, 0D1h
  out  64h, al
  mov  al, 0DFh
  out  60h, al
  mov  eax, CR0
  or   eax, 1
  mov  CR0, eax
       db 0EAh
       dw offset continue
       dw 16
continue:
  mov  ax, 8
  mov  ds, ax
  mov  ax, 24
  mov  ss, ax
  mov  ax, 32
  mov  es, ax
  ;Инициализируем GS селектором сегмента в расширенной памяти
  mov  ax, 40
  mov  gs, ax
  ;Загрузим селектор TSS 0 в регистр TR
  mov  ax, 48          ; (26)
  ltr  ax              ; (27)
  ; Переключимся на задачу 1
  call [dword ptr task1_offs]  ; (28)
  ; Сбросим задачу 1 на начало
  mov  [word ptr tss_1+20h], offset highmem    ; (29) Меняем IP
```

```
  ; Изменим атрибут выводимых символов
  mov  ah, 74h              ; (30) Красный на сером фоне
  mov  [color], ah          ; (31)
  ; Снова вызовим на задачу 1
  call [dword ptr task1_offs]  ; (32)
  ; Вывод отладочной информации
  mov  ax, 0FFFFh
home:
  mov  si, offset string
  debug
  mov  si, offset string
  mov  cx, len
  mov  ah, 74h
  mov  di, 1600
scr:
  lodsb
  stosw
  loop scr
  ;Закрыть линию A20
  mov  al, 0D1h
  out  64h ,al
  mov  al, 0DDh
  out  60h, al
  ;Возврат в реальный ражим
  mov  eax, CR0
  and  al, 0FEh
  mov  CR0, eax
      db 0EAh
      dw offset return
      dw @code
return:
  ;Восстановим операционную среду реального режима
  mov  ax, @data
  mov  ds, ax
  mov  ax, @stack
  mov  ss, ax
  mov  sp, 100h
  ;Восстановим значение IDTR для работы в реальном режиме
  lidt [fword ptr idtr_real]
  ;Разрешим аппаратные и немаскируемые прерывания
  sti
  in   al, 70h
  and  al, 07Fh
  out  70h, al
```

```
  mov  ah, 09h
  mov  dx, offset mes
  int  21h
  mov  ax, 4C00h
  int  21h
; Процедура заполняющая 2Мб сегмент в расширенной памяти
; Выполняет действия как и в задании ЛР №7.
proc highmem
  mov  ax, 40
  mov  gs, ax
  xor  eax, eax
  xor  ebx, ebx
  mov  ecx, 80000h
fill:
  mov  [dword ptr gs:ebx], eax
  push eax
  push cx
  mov  si, offset number+5
  debug
  shr  eax, 16
  mov  si, offset number
  debug
  mov  si, offset number
  mov  cx, 9
  mov  ah, [color]     ; (33) Атрибуты выводимых символов
  mov  di, 1040
scrh:
  lodsb
  stosw
  loop scrh
  pop  cx
  pop  eax
  add  ebx, 4
  inc  eax
  db 67h
  loop fill_1
  jmp  go
fill_1:
  jmp  fill
go:
  iret
endp
ends
code_size=$-sttt
end start
end
```

Таблица глобальных дескрипторов дополнена двумя новыми дескрипторами gdt_tss_0 и gdt_tss_l. Это дескрипторы сегментов состояния задач TSS. В нашем примере используются TSS минимального размера – по 104 байта, поэтому в поле границы дескрипторов TSS указано число 103. Атрибут 1 дескрипторов имеет значение 89h: присутствующий, уровень привилегий DPL=0, свободный TSS МП 486 (рис. 6.5).

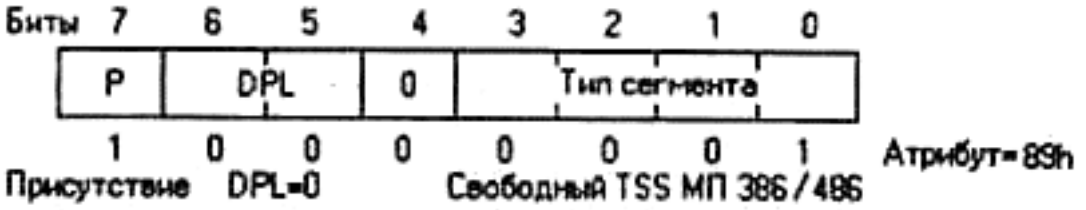

Рисунок 6.5 – Расшифровка значения атрибута для дескриптора TSS

Вспомним, что дескриптор TSS может быть четырех типов: свободный TSS МП 286, занятый TSS МП 286, свободный TSS МП 386/486 и занятый TSS МП 386/486. Описывая сегменты состояния задач в таблице дескрипторов, с помощью кода типа указывают, что они свободны. Как только селектор дескриптора TSS будет загружен в регистр задачи TR, процессор изменяет код атрибута дескриптора этого TSS, объявляя его занятым. При этом TSS исходной задачи остается занятым до ее завершения, даже если выполняются переключения на другие задачи. TSS вложенной задачи помечается процессором, как занятый, как только произойдет переключение на эту задачу, а после ее завершения и возврата в исходную задачу TSS вложенной задачи снова освобождается. Попытка переключения на задачу, TSS который занят, приводит к исключению нарушения общей защиты. Тем самым предотвращается повторный запуск активной задачи.

В сегменте данных главной задачи зарезервировано место для двух сегментов состояния задач. Там же предусмотрено двухсловное поле адреса для команды call dword ptr переключения на вложенную задачу. Поскольку переключение осуществляется через TSS задачи, в качестве сегментного адреса задачи указывается селектор ее TSS (в данном случае 56). Слово со смещением при переключении задач игнорируется, хотя должно присутствовать в программе согласно формату команды call.

Перед переключением на задачу 1 следует инициализировать некоторые поля TSS_1 (TSS_0 будет заполнен процессором при первом возврате из вложенной задачи в исходную). В данном примере инициализация TSS_1 выполняется еще в реальном режиме, хотя эту операцию

можно было бы перенести в защищенный режим. Поля для CS и IP заполняются селектором сегмента команд задачи 1 (в данном случае это общий для обеих задач сегмент с селектором 16) и смещением highmem точки входа в задачу 1. Для стека задачи 1 произвольно выделена область, начиная с середины нашего сегмента стека. Поскольку задача 1 будет использовать общий сегмент данных и обращаться к видеобуферу, ей передаются селекторы этих сегментов.

Далее следуют уже рассмотренные ранее операции запрета прерываний, подготовки адреса возврата в реальный режим, загрузки регистра IDTR, открытия линии A20 и перехода в защищенный режим. После перехода в защищенный режим и выполнения обычных процедур инициализации сегментных регистров в регистр задачи TR загружается селектор TSS исходной задачи. Переключение на вложенную задачу можно будет выполнить и без этого, однако загрузка TR обеспечит возврат из вложенной задачи в исходную.

Наконец, командой дальнего косвенного вызова осуществляется переключение на задачу 1, в качестве которой выступает процедура highnicm, размещенная в самом конце сегмента команд.

Задача 1 заполняет расширенную память последовательными числами, выводя их одновременно на экран, что позволяет наглядно контролировать ее работу. Завершающая команда iret этой задачи осуществляет обратное переключение, передавая управление на очередную команду задачи 0, следующую за командой дальнего вызова. Здесь продемонстрирована техника повторного переключения на задачу 1: в поле TSS для указателя команд восстанавливается начальный адрес процедуры highmem и снова выполняется команда дальнего вызова. Продолжение исходной задачи не отличается от предыдущего примера.

Отладочная макрокоманда debug, используемая в программе, дает возможность детально изучить процесс переключения задач и участие в этом процессе различных системных структур. Можно выполнить анализ следующих объектов.

1. Содержимое области связи в TSS_1 до переключения на задачу 1 и после этого переключения. До переключения эта область пуста, а после переключения в ней должен быть записан селектор TSS_0 (48=30h)

2. Значение кода атрибута в дескрипторах сегментов состояния задач. До активизации задачи код должен быть равен 89h, после активизации - 8Bh, поскольку сегмент становится занятым.

3. Состояние регистра флагов и, в частности, флага NT, до переключения на вложенную задачу, "внутри нее" и после возврата в исходную

задачу. Вывести на экран содержимое регистра флагов (нас будет интересовать только его младшее слово) можно следующим образом:

```
pushf                   ;Отправим флаги в стек
pop  ax                 ;Извлечем флаги из стека в AX
mov  si, offset string+10   ;Преобразуем в символьную
debug                   ;форму и выведем на экран
```

Этот эксперимент может оказаться не очень наглядным, поскольку, как уже отмечалось, в исходном состоянии компьютера флаг NT установлен. Естественно, он останется установленным и после переключения. Для того, чтобы детально изучить роль флага NT, следует сбросить его перед переключением на вложенную задачу. Это можно осуществить командами:

```
mov  ax, 0   ;Обнулим AX
push ax      ;Отправим O в стек
popf         ;Перенесем его в регистр флагов
```

(то, что сбрасываются и остальные флаги, роли не играет). Теперь анализ регистра флагов окажется более поучительным: флаг NT будет сброшен при выполнении задачи 0, но установлен при выполнении задачи 1. После возврата в задачу 0 флаг снова сбросится, так как именно такое состояние регистра флагов будет храниться в TSS_0.

Чтобы проанализировать процесс сохранения и восстановления содержимого регистра флагов, можно после очистки регистра флагов принудительно установить в нем какой-либо флаг, например CF (что можно выполнить командой stc) и вывести на экран, кроме слова флагов, еще и содержимое ячеек TSS_0 и TSS_1 со смещением 24h.

4. Процесс передачи параметров в вызываемую задачу через ее TSS. Для этого можно заполнить какими-либо числами поля TSS_1, предназначенные для хранения содержимого регистров общего назначения, и проанализировать содержимое этих регистров при входе в задачу 1.

3 Контрольные вопросы

1. Что понимают под задачей?
2. На какие аппаратные и программные средства опирается организация многозадачного режима?
3. Что представляет собой сегмент состояния задачи (TSS), каков его формат?
4. Что представляет собой шлюз задачи?
5. Какие действия происходят при переключении задачи?
6. Для чего предназначен флаг NT?
7. Как осуществляется связь между задачами?

4 Задание для самостоятельной работы

1. Изучить теоретические основы использования многозадачного режима.
2. Набрать и отладить и протестировать предложенную программу с использованием переключения задач. При этом обратить внимание на следующие этапы:
   - 2.1. Определение таблицы дескрипторов и дескрипторов используемых сегментов.
   - 2.2. Определение TSS задач.
   - 2.3. Определение переменной для переключения на задачу
   - 2.4. Подготовка таблицы дескрипторов к переходу в защищенный режим.
   - 2.5. Подготовка дескрипторов сегментов TSS.
   - 2.6. Подготовка TSS дочерней задачи.
   - 2.7. Структура процедуры дочерней задачи.
   - 2.8. Переключение на дочернюю задачу
   - 2.9. Повторное переключение на дочернюю задачу
   - 2.10. Подготовка к возврату и возврат в реальный режим
3. Модифицировать программу, добавив дополнительное переключение между задачами.
4. Отладить и протестировать полученную программу.
5. Оформить отчёт.

5 Содержание отчета

1. Титульный лист.
2. Краткое теоретическое описание.
3. Задание на лабораторную работу.
4. Листинг программы.
5. Результаты выполнения работы.

Лабораторная работа №9. Обработка аппаратных прерываний в защищенном режиме

1 Цель и порядок работы

Цель работы – изучить работу микропроцессора при обработке прерываний и научиться разрабатывать программы обработки прерываний в защищенном режиме.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- изучить работу микропроцессора при обработке прерываний;
- написать программу, ввести программу, отладить и выполнить ее на ЭВМ;
- оформить отчет.

2 Краткая теория

Обработка прерывания выполняется, за исключением своей начальной аппаратной стадии, так же, как и обработка исключения. При поступлении на вход INT микропроцессора сигнала от внешнего устройства процессор считывает с шины данных выставленный контроллером прерываний номер вектора, находит в таблице дескрипторов прерываний дескриптор с соответствующим номером и, сохранив предварительно в стеке адрес возврата, осуществляет переход на обработчик прерывания. Команда iret, которой заканчивается обработчик прерывания, возвращает управление в программу. Таким образом, для обработки аппаратных прерываний необходимо дополнить таблицу IDT дескрипторами обработчиков аппаратных прерываний и включить сами обработчики в текст программы.

Поскольку первые 32 вектора зарезервированы для обработки исключений, аппаратным прерываниям придется назначить какие-то другие векторы, например, начиная с номера 32=20h. Однако в машинах типа IBM PC контроллеры прерываний всегда программируются в процессе начальной загрузки так, что базовый вектор ведущего контроллера равен 8, а ведомого – 70h. Таким образом, перед переходом в защищенный режим необходимо перепрограммировать контроллеры прерываний, назначив ведущему контроллеру базовый вектор 20h, а ведомому – 28h. (или оставив у ведомого контроллера базовый вектор 70h). Различие базовых векторов в реальном и защищенном режимах является еще одной причиной программной несовместимости этих режимов.

Очевидно, что перед возвратом в реальный режим контроллеры надо снова перепрограммировать, иначе не смогут работать обработчики аппаратных прерываний BIOS.

Вторая особенность обработки прерываний связана с форматом дескриптора прерывания (шлюза).

В таблицу IDT могут входить шлюзы нескольких типов. В дескрипторе тип шлюза указывается в специально предназначенном для этого байте атрибутов. Формат этого байта изображен на рисунке 7.1.

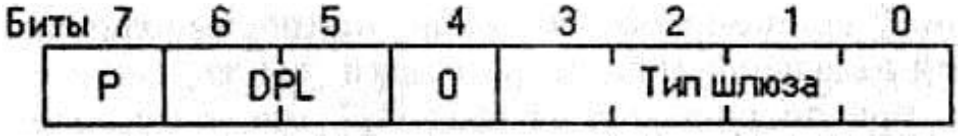

Рисунок 7.1 – Байт атрибутов дескриптора прерываний

Поле типа, занимающее 4 бита, может принимать 16 различных значений, однако в таблице IDT допустимо описывать только шлюзы следующих типов:

- шлюз задачи (тип 5);
- шлюз прерывания МП 286 (тип 6);
- шлюз ловушки МП 286 (тип 7);
- шлюз прерывания МП 386 или 486 (тип 0Eh);
- шлюз ловушки МП 386 или 486 (тип 0Fh).

Шлюз задачи используется в тех случаях, когда обработчик прерывания расположен в другой задаче; шлюзы прерываний и ловушек предполагают наличие обработчиков в текущей (прерываемой) задаче. Как видно из приведенного перечня, дескрипторы аппаратных прерываний должны иметь тип 0Fh.

Поле DPL определяет уровень привилегий шлюза. Уровень привилегий может принимать значения от 0 (максимальные привилегии) до 3 (минимальные привилегии) и используется для защиты программ друг от друга. DPL шлюза проверяется процессором только при выполнении команд программных прерываний-ловушек int и into, чтобы предотвратить обращение программ пользователя к системным программным прерываниям. Для остальных исключений и прерываний поле DPL процессором игнорируется.

Бит P в дескрипторе шлюза должен быть установлен в 1, в противном случае процессор будет рассматривать шлюз, как неправильный, и обращение к такому шлюзу вызовет исключение.

Таким образом, атрибут дескриптора шлюза прерывания МП 486 должен быт равен 8Eh, в отличие от дескрипторов ловушек, где он равен 8Fh.

Так же, как это имеет место и в реальном режиме, если переход на обработчик осуществляется через шлюз прерывания, процессор сбрасывает при входе в обработчик флаг IF в регистре флагов EFLAGS. Команда iret, загружая из стека сохраненное там исходное содержимое EFLAGS, снова разрешает прерывания. При переходе на обработчик через шлюз ловушки состояние флага IF не изменяется.

Для простоты программы ограничимся содержательной обработкой прерываний только от одного источника – таймера. Поскольку в компьютере таймер является единственным постоянно активным источником прерываний, а другие прерывания (от клавиатуры, дисков, КМОП-микросхемы и т.д.) сами по себе не возникают, мы можем выполнить инициализацию только этого прерывания: ограничиться перепрограммированием только ведущего контроллера и включить в таблицу IDT лишь один шлюз прерывания. Чтобы полностью обезопасить себя от незапланированных прерываний, их можно замаскировать в контроллерах прерываний.

В программе каждому исключению соответствует свой обработчик. Такое построение программы вместе со средствами отладки (макрокоманда Debug) весьма удобно, так как позволяет, в случае каких-либо неправильностей в программе, сразу же получить на экране номер возникшего исключения. Однако программа оказывается довольно громоздкой. В то же время практически при отладке наших программ будут возникать лишь исключения с номерами 10...14. Поэтому мы оставим только обработчики этих исключений, а для всех остальных (в том числе зарезервированных) исключений используем общий обработчик, который выводит на экран условный код 1111.

Пример 7.1 – Обработка аппаратных прерываний от таймера

```
;Обработка аппаратных прерываний от таймера
IDEAL
P386
model small
stack 100h
;Макрос для отладки
macro debug
  push ax
  push bx
  push cx
```

```
  push dx
  push ax
  and  ax, 0F000h
  shr  ax, 12
  mov  bx, offset tbl_hex
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0F00h
  shr  ax, 8
  inc  si
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0F0h
  shr  ax, 4
  inc  si
  xlat
  mov  [si], al
  pop  ax
  push ax
  and  ax, 0Fh
  inc  si
  xlat
  mov  [si], al
  pop  ax
  pop  dx
  pop  cx
  pop  bx
  pop  ax
endm
struc descr
      limit   dw 0
      base_l  dw 0
      base_m  db 0
      attr_1  db 0
      attr_2  db 0
      base_h  db 0
ends descr
;Структура для шлюзов ловушки
struc trap
      offs_l  dw 0
      sel     dw 16
      rsrv    db 0
```

```
        attr    db 8Fh
        offs_h  dw 0
ends trap
;Структура для шлюзов прерываний
struc intr
        offs_l  dw 0
        sel     dw 16
        rsrv    db 0
        attr    db 8Eh
        offs_h  dw 0
ends intr
DATASEG
   gdt_null     descr <0,0,0,0,0,0>            ; Селектор = 0
   gdt_data     descr <data_size-1,0,0,92h,0,0> ; Селектор = 8
   gdt_code     descr <code_size-1,0,0,98h,0,0> ; Селектор = 16
   gdt_stack    descr <100h-1,0,0,92h,0,0>      ; Селектор = 24
   gdt_screen   descr <4095,8000h,0Bh,92h,0,0>  ; Селектор = 32
  gdt_size = $-gdt_null
   ; Исключения 0 - 7 имеют общий обработчик
   idt          trap 10 dup (<dummy_exc>)
                trap <exc_0a>           ; Исключение 0Ah
                trap <exc_0b>           ; Исключение 0Bh
                trap <exc_0c>           ; Исключение 0Ch
   ; Исключение 0Dh - #GP (General Protection Fault)
                trap <exc_0d>
                trap <exc_0e>           ; Исключение 0Eh
   ; Исключения 0Fh-1Fh имеют общий обработчик
                trap 17 dup (<dummy_exc>)
   idt_08       intr <new_08h>          ; Обработчик системного таймера
  idt_size = $-idt
   idtr_real    dw 3FFh, 0, 0
   pdescr       dp 0
   mes          db 10,13,'Real mode','$'
   tbl_hex      db '0123456789ABCDEF'
   string       db '**** **** **** **** **** **** ****'
  len = $-string
   home_sel     dw home
                dw 10h
   mark_08h     dw 480
   color_08h    db 71h
   time_08h     db 0
   ; Переменные для маскирования прерываний
   master_mask  db 0
   slave_mask   db 0

  data_size = $-gdt_null
```

```
ends
CODESEG
assume cs: @code, ds:@data
sttt equ $
proc dummy_exc ;Обработчик исключений с номерами 0-9 и 0F-1F
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 1111h
  jmp  [dword ptr home_sel]
endp
proc exc_0a ;Обработчик исключения 0A
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Ah
  jmp  [dword ptr home_sel]
endp
proc exc_0b ;Обработчик исключения 0B
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Bh
  jmp  [dword ptr home_sel]
endp
proc exc_0c ;Обработчик исключения 0C
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Ch
  jmp  [dword ptr home_sel]
endp
proc exc_0d ;Обработчик исключения 0D
  pop  eax
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Dh
  jmp  [dword ptr home_sel]
endp
proc exc_0e ;Обработчик исключения 0E
  pop  eax
```

```
  pop  eax
  mov  si, offset string+5
  debug
  mov  ax, 0Eh
  jmp  [dword ptr home_sel]
endp

proc new_08h ;Обработчик прерывания системного таймера (IRQ0)
  ;Аппаратное прерывание - сохранить регистры
  push ax
  push bx
  ;Проверить значение счетчика
  test [time_08h], 03h
  jnz  @@skip
  mov  al, 21h
  mov  ah, [color_08h]
  mov  bx, [mark_08h]
  ; Вывести символ
  mov  [word ptr es:bx], ax
  ; Сместиться
  add  [mark_08h], 2
@@skip:
  inc  [time_08h]     ; Увеличить счетчик
  ;Послать EOI контроллеру прерываний
  mov  al,20h
  out  20h,al
  ;Восстановим регистры
  pop  bx
  pop  ax
  db 66h      ; Префикс замены размера операнда
  iret
endp
start:
  xor  eax, eax
  mov  ax, @data
  mov  ds, ax
  shl  eax, 4
  mov  ebp, eax
  mov  bx, offset gdt_data
  mov  [(descr ptr bx).base_l], ax
  rol  eax, 16
  mov  [(descr ptr bx).base_m], al
  xor  eax, eax
  mov  ax, cs
  shl  eax, 4
  mov  bx, offset gdt_code
```

```
mov  [(descr ptr bx).base_l], ax
rol  eax, 16
mov  [(descr ptr bx).base_m], al
xor  eax, eax
mov  ax, ss
shl  eax, 4
mov  bx, offset gdt_stack
mov  [(descr ptr bx).base_l], ax
rol  eax, 16
mov  [(descr ptr bx).base_m], al
;Подготовка к загрузке GDTR
mov  [dword ptr pdescr+2], ebp
mov  [word ptr pdescr], gdt_size-1
lgdt [pword ptr pdescr]
; Запрет аппаратных прерываний и NMI
cli
in   al, 70h
or   al, 80h
out  70h, al
;Перепрограммируем ведущий контроллер IRQ0-IRQ7
; (по умолчанию отображается на int 8h - int 15h)
mov  dx, 20h ; Порт ведущего контроллера
mov  al, 11h ; СКИ1 - инициализировать два контроллера
out  dx, al
jmp  $+2     ; Задержка
inc  dx      ; Второй порт контроллера (21h)
mov  al, 20h ; СКИ2 - базовый вектор
out  dx, al
jmp  $+2
mov  al, 4   ; СКИ3 - ведомый подключен к IRQ2 (4 = 000000100)
out  dx, al
jmp  $+2
mov  al, 1   ; СКИ4 - 80x86, программная генерация EOI
out  dx, al
; Маскируем прерывания ведущего контроллера
; 0FEh = 11111110b -> IRQ0 разрешено, IRQ1-IRQ7 запрещены
mov  dx, 021h
in   al, dx  ; Читаем текущее состояние маски
mov  [master_mask], al      ; Сохраним маску
mov  al, 0FEh; Разрешим IRQ0 - Системный таймер
out  dx, al
; Маскируем прерывания ведомого контроллера
mov  dx, 0A1h
in   al, dx  ; Читаем текущее состояние маски
mov  [slave_mask], al       ; Сохраним маску
```

```
  mov  al, 0FFh; Зпретим все прерывания
  out  dx, al
  ;Подготовка к загрузке IDTR
  mov  [word ptr pdescr], idt_size-1
  xor  eax, eax
  mov  ax, offset idt
  add  eax, ebp
  mov  [dword ptr pdescr+2], eax
  lidt [pword ptr pdescr]
  mov  eax, CR0
  or   eax, 1
  mov  CR0, eax
       db 0EAh
       dw offset continue
       dw 16
continue:
  mov  ax, 8
  mov  ds, ax
  mov  ax, 24
  mov  ss, ax
  mov  ax, 32
  mov  es, ax
  ;Разрешим аппаратные и немаскируемые прерывания
  sti
  in   al, 70h
  and  al, 07Fh
  out  70h, al
  ; Вывод символов на экарн
  mov  bx, 1600
  mov  cx, 800
  mov  dx, 3001h
xxxx:
  push cx
  mov  cx, 0
zzzz:
  loop zzzz
  mov  [word ptr es:bx], dx
  inc  dl
  add  bx, 2
  pop  cx
  loop xxxx
  mov ax, 0FFFFh
home:
  mov si, offset string
  debug
  mov si, offset string
```

```
  mov cx, len
  mov ah, 74h
  mov di, 1280
scr:
  lodsb
  stosw
  loop scr
  ; Запрет аппаратных прерываний и NMI
  cli
  in   al, 70h
  or   al, 80h
  out  70h, al
  ;Возврат в реальный ражим
  mov  eax, CR0
  and  al, 0FEh
  mov  CR0, eax
      db 0EAh
      dw offset return
      dw @code
return:
  ;Восстановим операционную среду реального режима
  mov  ax, @data
  mov  ds, ax
  mov  ax, @stack
  mov  ss, ax
  mov  sp, 100h
  ;Восстановим значение IDTR для работы в реальном режиме
  lidt [fword ptr idtr_real]
  ;Восстановим обратоно контроллер прерываний
  ;Перепрограммируем ведущий контроллер IRQ0-IRQ7
  ; (по умолчанию отображается на int 8h -int 15h)
  mov  dx, 20h ; Порт ведущего контроллера
  mov  al, 11h ; СКИ1 - инициализировать два контроллера
  out  dx, al
  jmp  $+2     ; Задержка
  inc  dx      ; Второй порт контроллера (21h)
  mov  al, 08h ; СКИ2 - базовый вектор
  out  dx, al
  jmp  $+2
  mov  al, 4   ; СКИ3 - ведомый подключен к IRQ2 (4 = 000000100)
  out  dx, al
  jmp  $+2
  mov  al, 1   ; СКИ4 - 80x86, программная генерация EOI
  out  dx, al
  ; Восстановим маски прерываний
  ; Маскируем прерывания ведущего контроллера
```

```
  mov  dx, 021h
  mov  al, [master_mask]
  out  dx, al
  ; Маскируем прерывания ведомого контроллера
  mov  dx, 0A1h
  mov  al, [slave_mask]
  out  dx, al
  ;Разрешим аппаратные и немаскируемые прерывания
  sti
  in   al, 70h
  and  al, 07Fh
  out  70h, al
  mov  ah, 09h
  mov  dx, offset mes
  int  21h
  mov  ax, 4C00h
  int  21h
ends
code_size=$-sttt
end start
end
```

Помимо структуры шлюзов ловушек в данную программу включена структура шлюзов прерываний, отличающаяся только значением байта типа.

Таблица прерываний должна содержать дескрипторы, расположенные по порядку их векторов. Поэтому начинается таблица с 10 одинаковых дескрипторов исключений, имеющих в программе общий обработчик dummy_exc. Затем идут дескрипторы исключений 10...14, а за ними еще 17 одинаковых дескрипторов исключений (реальных и зарезервированных). Наконец, на 33-м месте (вектор 32) описан дескриптор обработчика аппаратного прерывания от таймера (new_08h).

В сегмент данных включено несколько переменных для обслуживания обработчика от таймера (mark_08h, color_08h и timc_08h).

В сегменте команд обработчики прерываний и исключений, так же, как и все остальные процедуры, могут следовать в любом порядке. Наш сегмент команд начинается с обработчиков исключений с номерами 10...14. Предложения процедуры exc_0ah позволяют отправить в диагностическую строку string (на второе место в ней) содержимое IP, извлеченное из стека, т.е. адрес той команды, при выполнении которой возникло данное исключение. Произвольное смещение указателя стека (в стеке остались сегментный адрес и регистр EFLAGS) в данном случае не имеет значения, так как в случае исключения выполнение программы в

защищенном режиме завершается, и процессор переводится в реальный режим.

Текст общего обработчика исключений dummy_exc практически повторяет процедуры обработки исключений 10...14, затем исключением, что в строку string выводится не номер исключения, а условный код 1111h.

Далее следует обработчик прерываний от таймера. После сохранения используемых в нем регистров, выполняется проверка содержимого ячейки time_08h, причем программа обработчика продолжается, только если в этой ячейке сброшены оба младших бита. Тем самым осуществляется пересчет прерываний в отношении 4:1. На экран выводится цветной символ (конкретно восклицательный знак) и выполняются инкременты позиции на экране (в ячейке mark_08h) и счетчика прерываний time_08h. Обработчик завершается генерацией сигнала конца прерывания EOI для ведущего контроллера (порт 20h), восстановлением сохраненных ранее регистров и командой iret.

В главной процедуре обычным образом заполняются глобальные дескрипторы, загружается регистр GDTR, запрещаются немаскируемые и маскируемые прерывания.

Далее начинаются строки перепрограммирования контроллера прерываний. Некоторые настройки контроллера, например, изменение маски прерываний или генерацию сигнала EOI, можно выполнить засылкой соответствующего кода в порт управления контроллером. Однако сменить базовый вектор таким образом нельзя. Для смены базового вектора требуется выполнить полностью процедуру инициализации контроллера, которая состоит из ряда так называемых команд инициализации (СКИ/ICW), посылаемых в строгой последовательности друг за другом. Формат первого слова инициализации СКИ1, посылаемого (для ведущего контроллера) в порт 20h, представлен на рисунке 7.2.

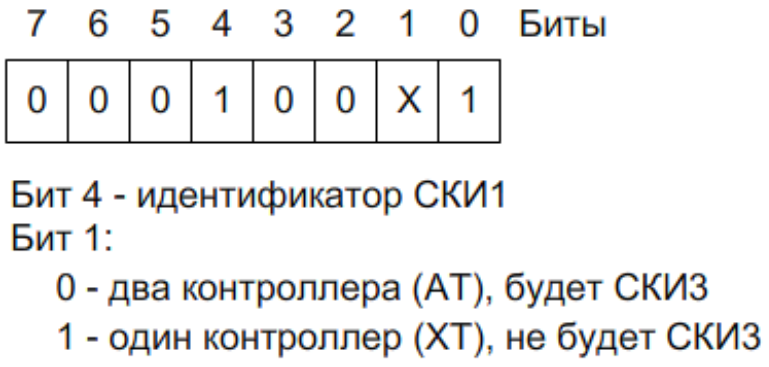

Рисунок 7.2 – Слово команд инициализации СКИ1

Структура СКИ1:

биты 7 – 4: 0001
бит 3: 1/0 – срабатывание по уровню/фронту сигнала IRQ (принято 0)
бит 2: 1/0 – размер вектора прерывания 4 байта/8 байт (1 для 80x86)
бит 1: каскадирования нет, СКИ3 не будет послано
бит 0: СКИ4 будет послано

Компьютер имеет два контроллера прерываний, поэтому СКИ1 равно 11h.

Второе слово инициализации СКИ2, посылаемое во второй порт контроллера (для ведущего контроллера – 21h) задает базовый вектор (номер обработчика прерывания) для IRQ0/IRQ8 (кратный восьми) (08h – для первого контроллера, 70h – для второго). Мы решили расположить векторы прерываний сразу вслед за векторами исключений, поэтому базовый вектор первого контроллера равен 32 = 20h.

Третье слово инициализации СКИ3 выглядит по-разному для ведущего и ведомого контроллеров. Для ведущего оно определяет номер входа, к которому подсоединен ведомый контроллер. Этот номер записывается в виде двоичной 1, установленной в том бите слова, который соответствует входу для ведомого. Для всех компьютеров типа IBM PC ведомый контроллер подсоединяется к входу 2 ведущего , поэтому в СКИ3 должен быть установлен бит 2, что соответствует числу 4 (0100b). СКИ3 посылается (для ведущего контроллера) в порт 21h. Биты 3 – 0 СКИ3 для подчиненного контроллера определяют номер выхода ведущего контроллера, к которому подсоединен ведомый.

Формат четвертого слова инициализации СКИ4 представлен на рисунке 7.3.

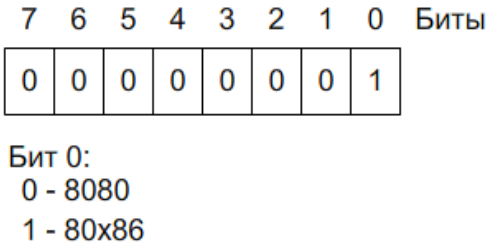

Рисунок 7.3 – Слово команд инициализации СКИ4

Более подробна структура СКИ4 представлена ниже:

биты 7 – 5: 0
бит 4: контроллер в режиме фиксированных приоритетов
биты 3 – 2: режим:

00, 01 – небуферированный
10 – буферированный/подчиненный
11 – буферированный/ведущий
бит 1: режим с автоматическим EOI
бит 0:
0 – режим совместимости с 8080
1 – обычный (80x86)

В режиме с автоматическим EOI (бит установлен в 1) осуществляется автоматический сброс бита регистра обслуживаемых запросов и снятие блокировки нижележащих уровней прерываний, то есть обработчикам не надо посылать EOI в контроллер. В противном случае требуется программная генерация сигнала EOI.

Для нашего случая (МП 80x86, требуется сигнал EOI) слово СКИ4 равно 1.

На этом последовательность инициализации заканчивается, однако еще надо установить маску прерываний. Таймер подключен к уровню 0, поэтому слово маски, посылаемое в порт 20h, равно 0FEh. Предварительно необходимо сохранить текущее состояние маски прерывания.

Инициализация ведомого контроллера, от которого прерывания поступать заведомо не будут, в программе не выполняется, но для надежности все прерывания в нем маскируются.

Теперь, когда вся система прерываний настроена, можно загрузить регистр IDTR, "подложив" процессору новую таблицу дескрипторов прерываний и исключений. Наконец, установкой бита 0 управляющего регистра CR0 процессор переводится в защищенный режим.

Сразу после перехода в реальный режим следует установить в контроллерах правильные значения масок. Конкретные значения зависят от конфигурации компьютера, поэтому их необходимо восстановить в исходное состояние.

Если программа подготовлена без ошибок, она будет функционировать следующим образом. В 10-ю и последующие строки экрана выводятся с небольшой скоростью черные символы по бирюзовому фону. Их количество (800) задано в программе. Одновременно в 3-ю строку с частотой 18,2/4 раз в секунду поступают изображения восклицательного знака (синие по белому фону). Перед завершением программы в 8-ю строку экрана выводится диагностическая строка

FFFF ****-**** ****__**** ****

и, наконец, после перехода процессора в реальный режим в позицию курсора выводится строка – Real mode.

Усложним предыдущий пример, сделав рассмотренную программу несколько более универсальной. Для этого введем в нее следующие добавления:

- обработчик прерываний от клавиатуры;
- инициализацию второго, ведомого контроллера прерываний;
- обработчики-заглушки остальных уровней прерываний.

Все добавления носят локальный характер, поэтому полный текст программы не приводится.

Поскольку необходимо обрабатывать все 15 аппаратных прерываний, в таблице дескрипторов прерываний должно быть соответствующее количество шлюзов. Добавим в нее после дескриптора int_08h следующие строки:

```
  idt_09 intr <new_09h>   ; Вектор 21h - прерывание от клавиатуры
        intr 6 dup (<master>); Векторы 22h...27h – аппаратные, ведущий
контроллер
        intr 8 dup (<slave>) ; Векторы 2Eh...2Fh - аппаратные, ведомый
контроллер
```

Этим самым однозначно определили, что векторы ведомого контроллера располагаются сразу вслед за векторами ведущего и, следовательно, номер базового вектора ведомого контроллера равен 20h+8=28h

Как видно из описания дескрипторов, прерывания от клавиатуры будут вызывать обработчик new_09h, остальные прерывания, проходящие через ведущий контроллер, – общий для них обработчик master, а все прерывания ведомого контроллера – общий для них обработчик slave.

В поля данных программы следует добавить две переменные, используемые обработчиком прерываний от клавиатуры:

```
  mark_09h     dw 800  ; Позиция на экране для вывода обработчиком
new_09h
  color_09h    db 1Eh  ; Атрибут символов
```

Вслед за обработчиком прерывания от таймера расположим процедуру new_09h обработчика прерываний от клавиатуры. В настоящем примере функции этого обработчика – прием скан-кодов нажимаемых клавиш и вывод их без всякого преобразования на экран. Естественно, на экране появляются символы, коды ASCII которых совпадают с поступающими скан-кодами. Эта программа поучительна в том отношении, что обрабатывает без фильтрации все прерывания от клавиатуры, отображая, таким образом, и скан-коды нажатия (make-codes), и скан-коды отпускания (break-codes). Более того, с ее помощью можно определить последовательности скен-кодов тех клавиш расширенной клавиатуры, которые

при их нажатии генерируют не один, а несколько сигналов прерываний (<Pause>, <PrtScr>, клавиши со стрелками и др.).

Пример 7.2 – Текст процедуры new_09h

```
; Обработчик прерывания клавиатуры (IRQ1)
; Обрабатываются оба скан-кода и нажатия, и отпускания
proc new_09h far
  push ax      ; Сохраним используемые
  push bx      ; регистры
  in   al, 60h ; Вводим скен-код из порта 60h
  mov  bx, [mark_09h]  ; Текущая позиция на экране
  mov  ah, [color_09h] ; Атрибут символов
  ;Вывод символа в видеобуфер
  mov  [word ptr es:bx], ax
  cmp  al, 80h ; Скан-код нажатия (<80h) ?
  jb   make
  ; Да, после него сдвинемся на 1 место
  add  [mark_09h], 2
make:
  ;Нет, сдвинемся еще на одно место
  add  [mark_09h], 2
  in   al, 61h ; Получим содержимое порта
  or   al, 80h ; Установкой старшего бита
  out  61h, al ; и последующим  сбросом его
  and  al, 7Eh ; сообщим контроллеру клавиатуры о
  out  61h, al ; приеме скан-кода символа
  mov  al, 20h ; Сигнал конца прерывания EOI
  out  20h, al ; в ведущий контроллер
  pop   bx     ; Восстановим используемое
  pop   ax     ; регистры
  db 66h       ; Возврат
  iret         ; в программу
endp
```

Помимо процедуры обработчика прерываний от клавиатуры, в сегмент команд следует включить обработчики "лишних" прерываний master (для ведущего контроллера) и slave (для ведомого). Их функции заключаются в создании сигналов EOI для обоих контроллеров в ответ на прерывания, содержательная обработка которых не предусмотрена в программе. Сигнал EOI разблокирует нижележащие уровни прерываний, которые аппаратно блокируются контроллером, когда процессор начинает обслуживать данное прерывание. Строго говоря, в нашем случае необходимости в этих обработчиках нет, во-первых, потому, что эти прерывания реально не возникают, а во-вторых, потому, что обрабатываемые прерывания от таймера и клавиатуры имеют наивысшие приорите-

ты и любое другое прерывание их заблокировать не может. Однако рассматриваемые фрагменты носят общий характер и могут оказаться полезными в других условиях.

Пример 7.3 – Тексты процедур master и slave

```
;Процедура для ведущего контроллера
proc master
  push ax     ;Сохраним используемый регистр
  mov  al, 20h ; Сигнал EOI
  out  20h, al
  pop  ax     ; Восстановим регистр
  db 66h      ; Возврат
  iret        ; в программу
endp master
; Процедура для ведомого контроллера
proc slave
  push ax     ; Сохраним используемый регистр
  mov  al,20h  ; Сигнал EOI для
  out  0A0h, al; ведомого контроллера
  mov  al, 20h ; Сигнал EOI для
  out  20h, al ; ведущего контроллера
  pop  ax     ; Восстановим регистр
  db 66h      ; Возврат
  iret        ; в программу
endp slave
```

Предполагая обрабатывать все аппаратные прерывания, необходимо инициализировать не только ведущий, но и ведомый контроллер. Так же нужно разрешить все прерывания убрав маски.

Пример 7.4 – Инициализация ведомого контроллера

```
;Перепрограммируем ведомый контроллер IRQ8-IRQ16
; (по умолчанию отображается на int 70h -int 77h)
mov  dx, 0A0h; Порт ведомого контроллера
mov  al, 11h ; СКИ1 - инициализировать два контроллера, будет СКИ3
out  dx, al
jmp  $+2     ; Задержка
inc  dx      ; Второй порт контроллера 0A1h
mov  al, 28h ; СКИ2: базовый вектор
out  dx, al
jmp  $+2
mov  al, 2   ;СКИ3: ведомый подключен к IRQ2
out  dx, al
jmp  $+2
```

```
mov  al, 1   ; СКИ4 - 80x86, программная генерация EOI
out  dx, al
jmp  $+2
```

Пример 7.5 – Реинициализация ведомого контроллера после возврата в реальный режим:

```
;Перепрограммируем ведомый контроллер IRQ8-IRQ16
;(по умолчанию отображается на int 70h - int 77h)
mov  dx, 0A0h; Порт ведомого контроллера
mov  al, 11h ; СКИ1 - инициализировать два контроллера, будет СКИ3
out  dx, al
jmp  $+2     ; Задержка
inc  dx      ; Второй порт контроллера 0A1h
mov  al, 70h ; СКИ2: базовый вектор
out  dx, al
jmp  $+2
mov  al, 2   ;СКИ3: ведомый подключен к IRQ2
out  dx, al
jmp  $+2
mov  al, 1   ; СКИ4 - 80x86, программная генерация EOI
out  dx, al
```

Также необходимо установить стандартные для компьютера значения масок прерываний.

3 Контрольные вопросы

1. Как происходит инициализация ведущего контроллера?
2. Чем отличается инициализация ведомого контроллера?
3. Как осуществляется процесс обработки аппаратных прерываний в защищённом режиме?
4. Опишите формат байта атрибутов шлюза.

4 Задание

1. Изучить теоретические основы обработки аппаратных прерываний.
2. Набрать и отладить и протестировать предложенную программу обработки прерываний от таймера. При этом обратить внимание на следующие этапы:
   - 2.1. Определение таблицы дескрипторов и дескрипторов используемых сегментов.
   - 2.2. Определение таблицы прерываний.
   - 2.3. Определение структуры шлюзов прерываний.

2.4. Определение переменных для маскирования прерываний.
2.5. Определение и структура обработчика системного таймера (IRQ0).
2.6. Перепрограммирование ведущего контроллера прерываний.
2.7. Маскирование прерывания ведущего и ведомого контроллера.
2.8. Работа программы обработки прерываний.
2.9. Восстановление настроек контроллера прерываний.
2.10. Восстановление масок прерываний.
2.11. Подготовка к возврату и возврат в реальный режим

3. Отладить и протестировать полученную программу.
4. Усложнить программу обработки прерываний от таймера, добавив в неё:
   4.1. Обработчик прерываний от клавиатуры
   4.2. Инициализацию ведомого контроллера прерываний
   4.3. Обработчики-заглушки остальных уровней прерываний.
5. Отладить и протестировать полученную программу.
6. Оформить отчёт.

5 Содержание отчета

1. Титульный лист.
2. Краткое теоретическое описание.
3. Задание на лабораторную работу.
4. Листинг программы.
5. Результаты выполнения работы.

Лабораторная работа №10 Многозадачный режим с управлением от клавиатуры

1 Цель и порядок работы

Цель работы – изучить работу многозадачного режима с управлением от клавиатуры и научиться писать программы с его использованием.

2 Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя;
- написать программу, ввести программу, отладить и решить ее на ЭВМ;
- оформить отчет.

3 Краткая теория

В многозадачной системе должна быть предусмотрена возможность переключения с задачи на задачу по каким-либо событиям, например, истечению заданного кванта времени или командам с клавиатуры. Рассмотрим детально процесс переключения задач в условиях действия аппаратных прерываний.

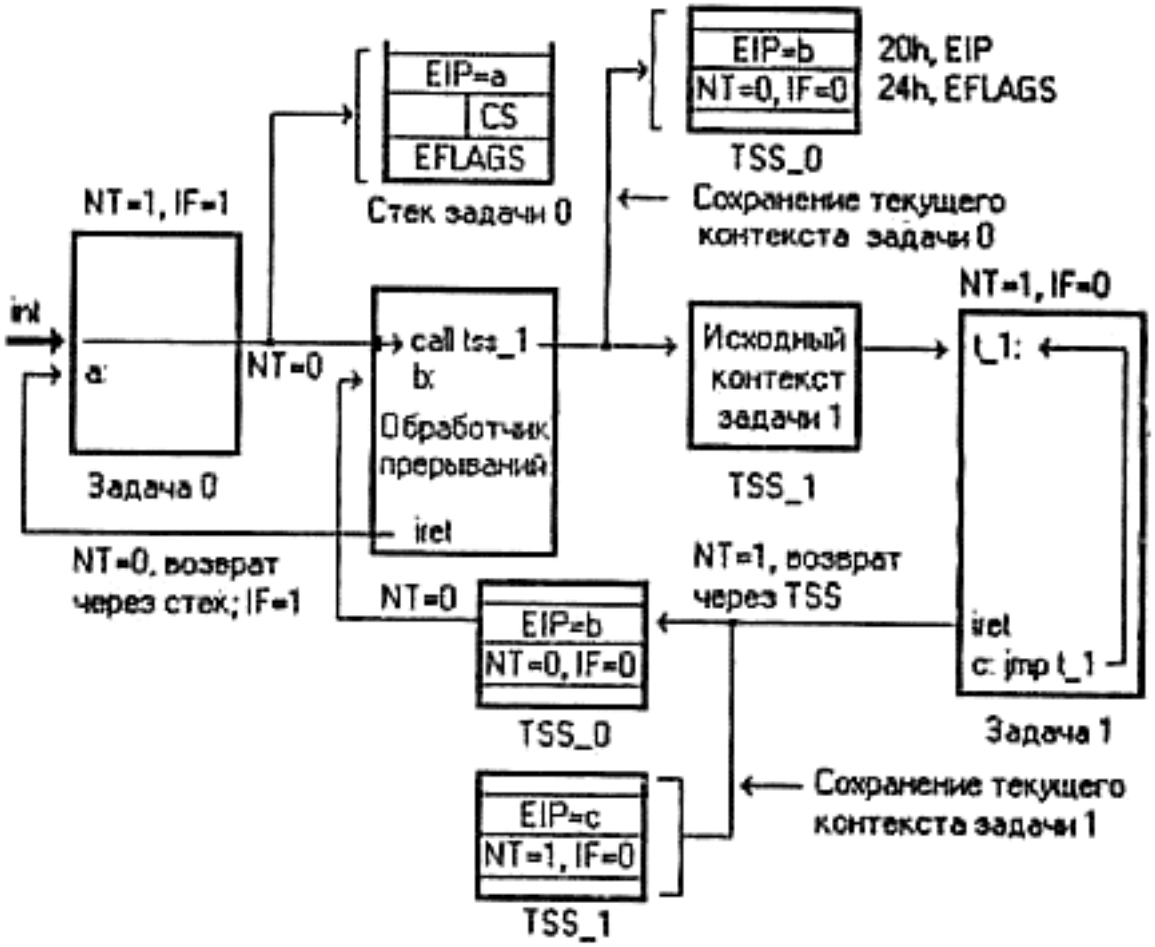

Рисунок 8.1 – Взаимодействие вычислительных объектов в процессе переключения задач

На рисунке 8.1 схематически изображено взаимодействие программно-аппаратных элементов при выполнении программы, которая носит "полусинхронный" характер – вторую задачу можно запустить в произвольный момент нажатием клавиши, однако дальше активизированная задача становится неуправляемой, выполняясь до завершающей команды iret.

При загрузке компьютера ведущий контроллер прерываний (IRQ0 – IRQ7) отображается на прерывания начиная с базового вектора 08h (int 08h – int 0Fh), ведомый (IRQ8 – IRQ15) – 70h (int 70h – int 77h). Номера прерываний, на которые отображаются аппаратные прерывания (IRQ), вызываемые первым контроллером по умолчанию, совпадают с номерами некоторых исключений. Конечно, можно из обработчика опрашивать контроллер прерываний, чтобы определить, выполняется ли обработка аппаратного прерывания или это исключение, но Intel рекомендует пере-

настраивать контроллер прерываний так, чтобы никакие аппаратные прерывания не попадали на область от 0 до 1Fh (зарезервировано для исключений). Поэтому необходимо выполнить перепрограммирование контроллера прерываний, как это рассматривалось в одной из предыдущих работ.

Исходная задача 0 выполняется в условиях установленных флагов NT и IF. Начальное состояние флага NT особого значения не имеет, а прерывания, естественно, должны быть разрешены. Приход сигнала аппаратного прерывания приводит к сбросу флагов NT и IF, занесению вектора прерванного процесса в стек задачи 0 и передаче управления через шлюз прерывания на обработчик прерываний от клавиатуры. В примере он находится в том же сегменте, что и задача 0. В качестве адреса возврата в стеке сохраняется адрес очередной команды (на рисунке он обозначен меткой a:).

В обработчике после анализа кода нажатой клавиши выполняется команда дальнего косвенного вызова call dword ptr task1_offs для которой в качестве сегментного адреса перехода указан селектор сегмента состояния задачи 1 TSS_1. На рисунке эта команда условно обозначена call tss_l, чтобы подчеркнуть, что вызов осуществляется через сегмент состояния задачи. Команда call инициирует действия по переключению задач: сегмент состояния первой задачи TSS_0 заполняется текущим контекстом, а содержимое TSS_1 используется в качестве исходного контекста для запуска задачи 1.

В этой процедуре есть два тонких момента. Во-первых, в TSS_0 загружается фактически не контекст задачи 0, а контекст обработчика прерываний к моменту выполнения команды call. В частности, в качестве точки возврата указывается адрес возврата в обработчик (метка b:), а в слове флагов сброшены флаги NT и IF (оба флага сбрасываются процессором при получении сигнала аппаратного прерывания). Если обработчик изменил содержимое каких-то регистров, то в TSS попадут именно эти измененные значения.

Другая особенность первого переключения на задачу 1 заключается в том, что контекст задачи целиком берется из сегмента состояния задачи TSS_1, который должен быть предварительно программно инициализирован. В примере поле флагов (по относительному адресу 24h) содержит 0. Следовательно, при выполнении задачи 1 будут запрещены прерывания. Что же касается флага NT, то, хотя в слове флагов TSS он был сброшен, задача 1 будет выполняться при NT=1, так как этот флаг устанавливается процессором при переключении задачи.

Задача 1 выполняется до завершающей команды iret, которая при NT=1 инициирует обратное переключение задач, получив из области связи текущего TSS_1 адрес "предыдущего" TSS_0. TSS_1 заполняется текущим контекстом задачи 1, при этом в поле для EIP записывается адрес команды, следующей за командой iret (метка c: на рисунке). Из TSS_0 берется контекст возврата (фактически - контекст обработчика прерываний) и управление передается на метку b: обработчика. Программа обработчика продолжается и, поскольку флаг NT сброшен, завершающая команда iret передает управление не через сегмент состояния задачи, а через стек, обеспечивая правильный возврат в исходную задачу на метку a:.

Последующие переключения на задачу 1 (по командам с клавиатуры) будут передавать управление на метку c:. Поскольку по этому адресу у нас находится команда безусловного перехода на начало программы (метка t_l на рисунке), задача будет выполняться правильно.

В рассматриваемом примере разрушение контекста задачи 0 обработчиком и сохранение в TSS_0 "неправильного" контекста не приводит к нарушению работы всего комплекса, если только в начале программы обработчика сохраняются (например, в стеке) все используемые в нем регистры, а перед завершающей командой iret выполняется их восстановление. С другой стороны, задача 1 выполняется при запрещенных прерываниях, и воздействовать на ход ее выполнения нельзя – она всегда будет выполняться до команды iret.

Для того чтобы получить возможность переключаться не только на задачу 1, но и из нее, надо выполнять эту задачу при разрешенных прерываниях. Прерывания можно разрешить как в самой задаче командой sti, так и в TSS задачи, записав в него исходное слово флагов с установленным флагом IF. Теперь нажатие клавиши в процессе выполнения задачи 1 снова активизирует обработчик прерывания, в котором, проанализировав код нажатой клавиши, можно выполнить команду переключения на другую задачу. Дескриптор сегмента состояния задачи может описывать TSS двух типов – свободные (код 9) и занятые (код Bh). Создавая в программе для каждой задачи сегменты состояния, объявляем их свободными. Команда call, осуществляя переключение на задачу, изменяет атрибут в дескрипторе соответствующего TSS, объявляя его занятым. TSS останется занятым до тех пор, пока в этой задаче не будет выполнена команда iret возврата назад по цепочке связанных задач. При этом команда call может активизировать только задачу со свободным TSS; при попытке вызывать командой call задачу с занятым TSS возбуждается исключение общей защиты. Из вложенной задачи с помощью ко-

манды call можно активизировать следующую задачу, но нельзя вернуться назад – для этого нужна команда iret.

Для того чтобы обойти указанное ограничение, можно воспользоваться командой дальнего перехода через сегмент состояния задачи. Команда jmp, выполняя переключение задачи, не включает ее в связный список вложенных задач и, соответственно, но изменяет тип дескриптора TSS. Поэтому с помощью команды jmp можно осуществлять переключение задач как "вперед" – на новую задачу, так и "назад" – на ту задачу, из которой была активизирована данная.

Рассмотрим пример многозадачной системы, в которой переключение задач осуществляется по командам с клавиатуры с помощью команд дальнего косвенного перехода jmp. Предусмотрим в программе главную задачу 0, в которой будет осуществляться инициализация всей системы, а также две демонстрационные задачи 1 и 2. Поначалу ради простоты расположим все задачи в одном сегменте команд, и будем считать, что всем задачам доступен общий сегмент данных.

Пример 8.1 – Многозадачный режим с переключением от клавиатуры

```
;Многозадачный режим с переключением от клавиатуры
IDEAL
P386
model small
stack 300h
;Макрос для отладки
macro debug
   push ax
   push bx
   push cx
   push dx
   push ax
   and  ax, 0F000h
   shr  ax, 12
   mov  bx, offset tbl_hex
   xlat
   mov  [si], al
   pop  ax
   push ax
   and  ax, 0F00h
   shr  ax, 8
   inc  si
   xlat
   mov  [si], al
   pop  ax
   push ax
```

```
   and  ax, 0F0h
   shr  ax, 4
   inc  si
   xlat
   mov  [si], al
   pop  ax
   push ax
   and  ax, 0Fh
   inc  si
   xlat
   mov  [si], al
   pop  ax
   pop  dx
   pop  cx
   pop  bx
   pop  ax
endm
struc descr
        limit   dw 0
        base_l  dw 0
        base_m  db 0
        attr_1  db 0
        attr_2  db 0
        base_h  db 0
ends descr
;Структура для шлюзов ловушки
struc trap
        offs_l  dw 0
        sel     dw 16
        rsrv    db 0
        attr    db 8Fh
        offs_h  dw 0
ends trap
;Структура для шлюзов прерываний
struc intr
        offs_l  dw 0
        sel     dw 16
        rsrv    db 0
        attr    db 8Eh
        offs_h  dw 0
ends intr
DATASEG
   gdt_null     descr <0,0,0,0,0,0>             ; Селектор = 0
   gdt_data     descr <data_size-1,0,0,92h,0,0> ; Селектор = 8
   gdt_code     descr <code_size-1,0,0,98h,0,0> ; Селектор = 16
   gdt_stack    descr <300h-1,0,0,92h,0,0>      ; Селектор = 24
   gdt_screen   descr <4095,8000h,0Bh,92h,0,0>  ; Селектор = 32
   gdt_tss_0    descr <103,0,0,89h,0,0>         ; Селектор = 40
   gdt_tss_1    descr <103,0,0,89h,0,0>         ; Селектор = 48
   gdt_tss_2    descr <103,0,0,89h,0,0>         ; Селектор = 56
  gdt_size = $-gdt_null
```

```
   idt             trap 10 dup (<dummy_exc>); Исключения 0 - 7 имеют
общий обработчик
                   trap <exc_0a>             ; Исключение 0Ah
                   trap <exc_0b>             ; Исключение 0Bh
                   trap <exc_0c>             ; Исключение 0Ch
                   trap <exc_0d>    ; Исключение 0Dh - #GP (General
Protection Fault)
                   trap <exc_0e>             ; Исключение 0Eh
                   trap 17 dup (<dummy_exc>); Исключения 0Fh-1Fh
имеют общий обработчик
                   intr <master>             ; IRQ0 - Вектор 20h -
общий обработчик
                   intr <new_09h>   ; IRQ1 - Вектор 21h - прерывание
от клавиатуры
                   intr 6 dup (<master>)     ; Векторы 22h...27h -
аппаратные,
                                             ; ведущий контроллер
прерываний
                   intr 8 dup (<slave>)      ; Векторы 2Eh...2Fh -
аппаратные,
                                             ; ведомый контроллер
прерываний
  idt_size = $-idt
   idtr_real     dw 3FFh, 0, 0
   pdescr        dp 0
   mes           db 10,13,'Real mode','$'
   tbl_hex       db '0123456789ABCDEF'
   string        db '**** **** **** **** **** **** ****'
  len = $-string
   home_sel      dw home
                 dw 10h
   ;Переменные для переключения на задачи
   t0_addr       dw 0, 40                  ; Смещение + селектор TSS
0 = 40
   t1_addr       dw 0, 48                  ; Смещение + селектор TSS
1 = 48
   t2_addr       dw 0, 56                  ; Смещение + селектор TSS
2 = 56
   tss_0         db 104 dup(0)             ; TSS 0-й задачи
   tss_1         db 104 dup(0)             ; TSS 1-й задачи
   tss_2         db 104 dup(0)             ; TSS 2-й задачи
   mes1 db 22,22,22," task_1 ",22,22,22 ; Строка выводимая задачей
1
  mes1_len = $-mes1                        ; Длинна строки
   mes2 db 22,22,22," task_2 ",22,22,22 ; Строка выводимая задачей
2
  mes2_len = $-mes2                        ; Длинна строки
   ; Переменные для маскирования прерываний
   master_mask  db 0
   slave_mask   db 0
```

```
  data_size = $-gdt_null
ends
CODESEG
assume cs: @code, ds:@data
sttt equ $
proc dummy_exc ;Обработчик исключений с номерами 0-9 и 0F-1F
   pop  eax
   pop  eax
   mov  si, offset string+5
   debug
   mov  ax, 1111h
   jmp  [dword ptr home_sel]
endp
proc exc_0a ;Обработчик исключения 0A
   pop  eax
   pop  eax
   mov  si, offset string+5
   debug
   mov  ax, 0Ah
   jmp  [dword ptr home_sel]
endp
proc exc_0b ;Обработчик исключения 0B
   pop  eax
   pop  eax
   mov  si, offset string+5
   debug
   mov  ax, 0Bh
   jmp  [dword ptr home_sel]
endp
proc exc_0c ;Обработчик исключения 0C
   pop  eax
   pop  eax
   mov  si, offset string+5
   debug
   mov  ax, 0Ch
   jmp  [dword ptr home_sel]
endp
proc exc_0d ;Обработчик исключения 0D
   pop  eax
   pop  eax
   mov  si, offset string+5
   debug
   mov  ax, 0Dh
   jmp  [dword ptr home_sel]
endp
proc exc_0e ;Обработчик исключения 0E
   pop  eax
   pop  eax
   mov  si, offset string+5
   debug
   mov  ax, 0Eh
```

```
   jmp  [dword ptr home_sel]
endp
;Обработчик прерывания клавиатуры (IRQ1)
;Нельзя завершить обработчик просто командой IRET, потому что:
; во-первых, обработчик аппаратного прерывания клавиатуры должен
; установить бит 7 порта 61h, а затем вернуть его в исходное
состояние
; во-вторых, он должен сообщить контроллеру прерываний, что
обработка
; аппаратного прерывания закончилась командами
proc new_09h
   ;Аппаратное прерывание - сохранить регистры
   push ax
   push dx
   ;Чтение скан-кода
   in   al, 60h ; Прочитать скан-код нажатой клавиши,
   mov  dl, al
   ;Обработка прерывания клавиатуры
   in   al, 61h
   or   al, 80h ; Установить бит 7 порта 61h
   out  61h, al
   and  al, 7Fh ; Восстанавливаем бит 7
   out  61h, al
   ;Послать EOI контроллеру прерываний
   mov  al, 20h
   out  20h, al
   ;Проверяем введенный символ
   cmp  dl, 0Bh ; 0Bh = Клавиша <0>
   je   tsk0
   cmp  dl, 2   ; 02h = Клавиша <1>
   je   tsk1
   cmp  dl, 3   ; 03h = Клавиша <2>
   je   tsk2
   jmp  out_09h
tsk0:
   ;Переключение на задачу 0
   jmp  [dword ptr t0_addr]
   jmp  out_09h
tsk1:
   ;Переключение на задачу 1
   jmp  [dword ptr t1_addr]
   jmp  out_09h
tsk2:
   ;Переключение на задачу 2
   jmp  [dword ptr t2_addr]
out_09h:
   ;Выход из обработчика
   ;Восстанавливаем измененные регистры
   pop  dx
   pop  ax
   db 66h       ; Префикс замены размера операнда
```

```
   iret
endp
;Процедура для ведущего контроллера
proc master
   push ax      ;Сохраним используемый регистр
   mov  al, 20h ; Сигнал EOI
   out  20h, al
   pop  ax      ; Восстановим регистр
   db 66h       ; Возврат
   iret         ; в программу
endp master
; Процедура для ведомого контроллера
proc slave
   push ax      ; Сохраним используемый регистр
   mov  al,20h  ; Сигнал EOI для
   out  0A0h, al; ведомого контроллера
   mov  al, 20h ; Сигнал EOI для
   out  20h, al ; ведущего контроллера
   pop  ax      ; Восстановим регистр
   db 66h       ; Возврат
   iret         ; в программу
endp slave

start:
   xor  eax, eax
   mov  ax, @data
   mov  ds, ax
   shl  eax, 4
   mov  ebp, eax
   mov  bx, offset gdt_data
   mov  [(descr ptr bx).base_l], ax
   rol  eax, 16
   mov  [(descr ptr bx).base_m], al
   xor  eax, eax
   mov  ax, cs
   shl  eax, 4
   mov  bx, offset gdt_code
   mov  [(descr ptr bx).base_l], ax
   rol  eax, 16
   mov  [(descr ptr bx).base_m], al
  xor  eax, eax
   mov  ax, ss
   shl  eax, 4
   mov  bx, offset gdt_stack
   mov  [(descr ptr bx).base_l], ax
   rol  eax, 16
   mov  [(descr ptr bx).base_m], al
   ;Вычислим 32-битовый линейный адрес сегмента TSS 0 и загрузим
его
   ;в дескриптор сегмента TSS 0 в таблице глобальных дескрипторов
   mov  eax, ebp                          ; Адрес начала сегмента
```

```
данных
   add  ax, offset tss_0                ; Добавим смещение TSS0
   mov  bx, offset gdt_tss_0
   mov  [(descr ptr bx).base_l], ax
   rol  eax, 16
   mov  [(descr ptr bx).base_m], al
   ;Вычислим 32-битовый линейный адрес сегмента TSS 1 и загрузим
его
   ;в дескриптор сегмента TSS 1 в таблице глобальных дескрипторов
   mov  eax, ebp
   add  ax, offset tss_1
   mov  bx, offset gdt_tss_1
   mov  [(descr ptr bx).base_l], ax
   rol  eax, 16
   mov  [(descr ptr bx).base_m], al
   ;Вычислим 32-битовый линейный адрес сегмента TSS 2 и загрузим
его
   ;в дескриптор сегмента TSS 2 в таблице глобальных дескрипторов
   mov  eax, ebp
   add  ax, offset tss_2
   mov  bx, offset gdt_tss_2
   mov  [(descr ptr bx).base_l], ax
   rol  eax, 16
   mov  [(descr ptr bx).base_m], al
   ;Подготовка к загрузке GDTR
   mov  [dword ptr pdescr+2], ebp
   mov  [word ptr pdescr], gdt_size-1
   lgdt [pword ptr pdescr]
   ; TSS 0 заполнится автоматически
   ; Заполним поля TSS 1
   mov  [word ptr tss_1+4Ch], 16                ; CS
   mov  [word ptr tss_1+20h], offset task1      ; IP
   mov  [word ptr tss_1+50h], 24                ; SS
   mov  [word ptr tss_1+38h], 256               ; SP
   mov  [word ptr tss_1+54h], 8                 ; DS
   mov  [word ptr tss_1+48h], 32                ; ES
   ; Установим флаг IF и сохраним регистр флагов EFLAGS в EAX
   sti
   pushfd
   pop  eax
   mov  [dword ptr tss_1+24h], eax              ; EFLAGS


   ; Заполним поля TSS 2
   mov  [word ptr tss_2+4Ch], 16                ; CS
   mov  [word ptr tss_2+20h], offset task2      ; IP
   mov  [word ptr tss_2+50h], 24                ; SS
   mov  [word ptr tss_2+38h], 512               ; SP
   mov  [word ptr tss_2+54h], 8                 ; DS
   mov  [word ptr tss_2+48h], 32                ; ES
   mov  [dword ptr tss_2+24h], eax              ; EFLAGS
```

```
    ; Запрет аппаратных прерываний и NMI
    cli
    in    al, 70h
    or    al, 80h
    out   70h, al
;Перепрограммируем ведущий контроллер IRQ0-IRQ7
    mov   dx, 20h ; Порт ведущего контроллера
    mov   al, 11h ; СКИ1 - инициализировать два контроллера
    out   dx, al
    jmp   $+2     ; Задержка
    inc   dx      ; Второй порт контроллера (21h)
    mov   al, 20h ; СКИ2 - базовый вектор
    out   dx, al
    jmp   $+2
    mov   al, 4   ; СКИ3 - ведомый подключен к IRQ2 (4 = 000000100)
    out   dx, al
    jmp   $+2
    mov   al, 1   ; СКИ4 - 80x86, программная генерация EOI
    out   dx, al
    jmp   $+2
    ;Перепрограммируем ведомый контроллер IRQ8-IRQ16
    mov   dx, 0A0h; Порт ведомого контроллера
    mov   al, 11h ; СКИ1 - инициализировать два контроллера, будет
СКИ3
    out   dx, al
    jmp   $+2     ; Задержка
    inc   dx      ; Второй порт контроллера 0A1h
    mov   al, 28h ; СКИ2: базовый вектор
    out   dx, al
    jmp   $+2
    mov   al, 2   ;СКИ3: ведомый подключен к IRQ2
    out   dx, al
    jmp   $+2
    mov   al, 1   ; СКИ4 - 80x86, программная генерация EOI
    out   dx, al
    jmp   $+2
    ; Маскируем прерывания ведущего контроллера
    ; 0FEh = 11111101b -> IRQ1 разрешено, IRQ0, IRQ2-IRQ7 запрещены
    mov   dx, 021h
    in    al, dx  ; Читаем текущее состояние маски
    mov   [master_mask], al       ; Сохраним маску
    ; Разрешим IRQ1 - Клавиатура
    and   al, 0000010b
    out   dx, al
    ; Маскируем прерывания ведомого контроллера
    mov   dx, 0A1h
    in    al, dx  ; Читаем текущее состояние маски
    mov   [slave_mask], al        ; Сохраним маску
    and   al, 00000000b   ; Зпретим все прерывания
    out   dx, al
```

```
   ;Подготовка к загрузке IDTR
   mov  [word ptr pdescr], idt_size-1
   xor  eax, eax
   mov  ax, offset idt
   add  eax, ebp
   mov  [dword ptr pdescr+2], eax
   lidt [pword ptr pdescr]

   ;Открыть линию A20
   mov  al, 0D1h
   out  64h, al
   mov  al, 0DFh
   out  60h, al

   mov  eax, CR0
   or   eax, 1
   mov  CR0, eax
        db 0EAh
        dw offset continue
        dw 16
continue:
   mov  ax, 8
   mov  ds, ax
   mov  ax, 24
   mov  ss, ax
   mov  ax, 32
   mov  es, ax
   ;Загрузим селектор TSS 0 в регистр TR
   mov  ax, 40
   ltr  ax
   ;Разрешим аппаратные и немаскируемые прерывания
   sti
   in   al, 70h
   and  al, 07Fh
   out  70h, al
   ; Задача 0
   mov  bx, 1600
   mov  cx, 800
   mov  dx, 3001h
xxxx:
   push cx
   mov  cx, 0
zzzz:
   loop zzzz
   mov  [word ptr es:bx], dx
   inc  dl
   pop  cx
   add  bx, 2
   loop xxxx
```

```
   mov ax,0ffffh
home:
   mov si, offset string
   debug
   mov si, offset string
   mov cx, len
   mov ah, 74h
   mov di, 1600
scr:
   lodsb
   stosw
   loop scr
   ;Закрыть линию A20
   mov  al, 0D1h
   out  64h ,al
   mov  al, 0DDh
   out  60h, al
   ; Запрет аппаратных прерываний и NMI
   cli
   in   al, 70h
   or   al, 80h
   out  70h, al
   ;Возврат в реальный ражим
   mov  eax, CR0
   and  al, 0FEh
   mov  CR0, eax
        db 0EAh
        dw offset return
        dw @code
return:
   ;Восстановим операционную среду реального режима
   mov  ax, @data
   mov  ds, ax
   mov  ax, @stack
   mov  ss, ax
   mov  sp, 100h
   ;Восстановим значение IDTR для работы в реальном режиме
   lidt [fword ptr idtr_real]
   ;Восстановим обратоно контроллер прерываний
   ;Перепрограммируем ведущий контроллер IRQ0-IRQ7 (по умолчанию
int 8h - int 15h)
   mov  dx, 20h ; Порт ведущего контроллера
   mov  al, 11h ; СКИ1 - инициализировать два контроллера
   out  dx, al
   jmp  $+2     ; Задержка
   inc  dx      ; Второй порт контроллера (21h)
   mov  al, 08h ; СКИ2 - базовый вектор
   out  dx, al
   jmp  $+2
```

```
   mov  al, 4    ; СКИ3 - ведомый подключен к IRQ2 (4 = 000000100)
   out  dx, al
   jmp  $+2
   mov  al, 1    ; СКИ4 - 80x86, программная генерация EOI
   out  dx, al
   ;Перепрограммируем ведомый контроллер IRQ8-IRQ16 (по умолчанию
int 70h - int 77h)
   mov  dx, 0A0h; Порт ведомого контроллера
   mov  al, 11h ; СКИ1 - инициализировать два контроллера, будет
СКИ3
   out  dx, al
   jmp  $+2      ; Задержка
   inc  dx       ; Второй порт контроллера 0A1h
   mov  al, 70h ; СКИ2: базовый вектор
   out  dx, al
   jmp  $+2
   mov  al, 2    ;СКИ3: ведомый подключен к IRQ2
   out  dx, al
   jmp  $+2
   mov  al, 1    ; СКИ4 - 80x86, программная генерация EOI
   out  dx, al
   jmp  $+2
   ; Восстановим маски прерываний
   ; Маскируем прерывания ведущего контроллера
   mov  dx, 021h
   mov  al, [master_mask]
   out  dx, al
   ; Маскируем прерывания ведомого контроллера
   mov  dx, 0A1h
   mov  al, [slave_mask]
   out  dx, al
   ;Разрешим аппаратные и немаскируемые прерывания
   sti
   in   al, 70h
   and  al, 07Fh
   out  70h, al

   mov  ah, 09h
   mov  dx, offset mes
   int  21h

   mov  ax, 4C00h
   int  21h

; Задача 1: выводит на экран сообщение mes1
proc task1
a1:
   mov  ah, 71h ; Атрибут символов
   mov  cx, 2
a2:
   push cx
```

```
    mov  di, 800
    mov  cx, mes1_len
    mov  si, offset mes1
    cld
    ;Вывод на экран
a3:
    lodsb
    stosw
    loop a3
    ;Задержка
    mov  cx, 0FFFFh
a4:
    loop a4
    pop  cx
    mov  ah, 17h
    loop a2
    jmp  a1
endp
; Задача 2: выводит на экран сообщение mes2
proc task2
b1:
    mov  ah, 34h ; Атрибут символов
    mov  cx, 2
b2:
    push cx
    mov  di, 1120
    mov  cx, mes2_len
    mov  si, offset mes2
    cld
    ;Вывод на экран
b3:
    lodsb
    stosw
    loop b3
    ;Задержка
    mov  cx, 0FFFFh
b4:
    loop  b4
    pop  cx
    mov  ah, 43h
    loop b2
    jmp  b1
endp
ends
code_size=$-sttt
end start
end
```

В таблице глобальных дескрипторов предусмотрены три дескриптора gdt_tss_0, gdt_tss_l и gdt_tss_2 дня сегментов состояния задач. Раз-

мер TSS - 104 байта (граница равна 103), тип дескриптора – сободные TSS МП 486.

В таблице IDT на месте, соответствующем вектору клавиатуры, расположен дескриптор - шлюз прерывания для вызова обработчика new_09h.

Три двухсловных ячейки t0_addr, t1_addr и t2_addr предназначены для команд дальних косвенных переходов, реализующих переключение задач. Первые слова этих ячеек игнорируются, во-вторых, записаны селекторы сегментов состояния задач TSS_0, TSS_1 и TSS_2.

Сообщения с именами mes1 и mcs2 предназначены для вывода задачами 1 и 2. Сообщения украшены с обеих сторон символами полосочек (код ASCII 22).

Обработчик прерываний от клавиатуры анализирует коды нажимаемых клавиш и осуществляет переключение на задачи 0, 1 или 2 при нажатии клавиш <0>, <1> и <2>, соответственно. Поскольку при запуске программы начинает работать задача 0, первое переключение должно быть на задачу 1 или 2. После этого задачи можно переключать в любом порядке, хотя недопустимо переключение на активную задачу.

На этапе инициализации следует обратить внимание на инициализацию сегментов состояния переключаемых задач. TSS задачи 0, как и раньше, не инициализируем: он будет заполнен процессором при первом переключении на задачи 1 или 2. TSS задач 1 и 2 инициализируются почти одинаково. В TSS заполняются поля CS (селектором общего сегмента команд), IP (относительными адресами задач), SS (селектором общего сегмента стека), DS (селектором общего сегмента данных), ES (селектором видеобуфера). В поля для SP заносятся разные смещения в пределах одного сегмента стека, размер которого увеличен в три раз». Таким образом, фактически задачи будут работать на разных стеках, что необходимо, поскольку переключения по аппаратным прерываниям могут происходить в произвольные моменты времени. Для того чтобы вход в задачи 1 и 2 происходил при разрешенных прерываниях, в поля для EFLAGS заносится текущее содержимое EFLAGS после того, как командой sti явно разрешены прерывания.

Задачи 1 и 2 практически одинаковы. В каждой из них выводится на экран мигающая строка текста, для чего организован цикл из двух шагов. В одном шаге цикла строка выводится с одним атрибутом, в другом шаге – с другим. Завершающая команда jmp обеспечивает бесконечное выполнение каждой задачи. Таким образом, в программе не предусматривается "естественное” завершение задач, что потребовало бы их заметного усложнения.

Рассмотрим ход выполнения программного комплекса, а также его возможности и недостатки (рис. 8.2).

Исходная задача 0, как и в предыдущем примере, выполняется в условиях установленных флагов NT и IF. Приход сигнала аппаратного прерывания приводит к сбросу флагов NT и IF, занесению вектора прерванного процесса в стек задачи 0 и передаче управления через шлюз прерывания на обработчик прерываний от клавиатуры. В качестве адреса возврата в стеке сохраняется адрес очередной команды задачи 0 (на рисунке он обозначен меткой a:).

В обработчике после анализа кода нажатой клавиши с помощью команды дальнего косвенного перехода Jmp dword ptr task1_offs выполняется переключение на задачу 1. Сегмент состояния главной задачи TSS_0 заполняется текущим контекстом, а содержимое TSS_1 используется в качестве исходного контекста для запуска задачи 1. Поскольку в момент переключения выполняется программа обработчика, в TSS_0 в поле для EIP записывается адрес команды обработчика, следующей за командой jmp и помеченной на рисунке меткой b. Задача 1 представляет собой бесконечный цикл, поэтому она будет выполняться до тех пор, пока пользователь не подаст с клавиатуры команду переключения задач.

Приход прерывания от клавиатуры, как и раньше, приводит к сбросу флагов NT и IF, занесению вектора прерванного процесса в стек задачи 1 и передаче управления на обработчик прерываний от клавиатуры.

Пусть пользователь подал команду возврата в задачу 0 (нажав на клавишу <0>). Обработчик, проанализировав код нажатой клавиши, передает управление на команду переключения на задачу 0 (на рисунке она обозначена, как jmp tss_0). Эта команда инициирует в процессоре процедуру переключения, в процессе которой TSS задачи 1 заполняется текущим контекстом, а содержимое TSS_0 используется в качестве исходного контекста для запуска задачи 0. Но в TSS_0 в качестве адреса возврата указано значение метки b в обработчике.

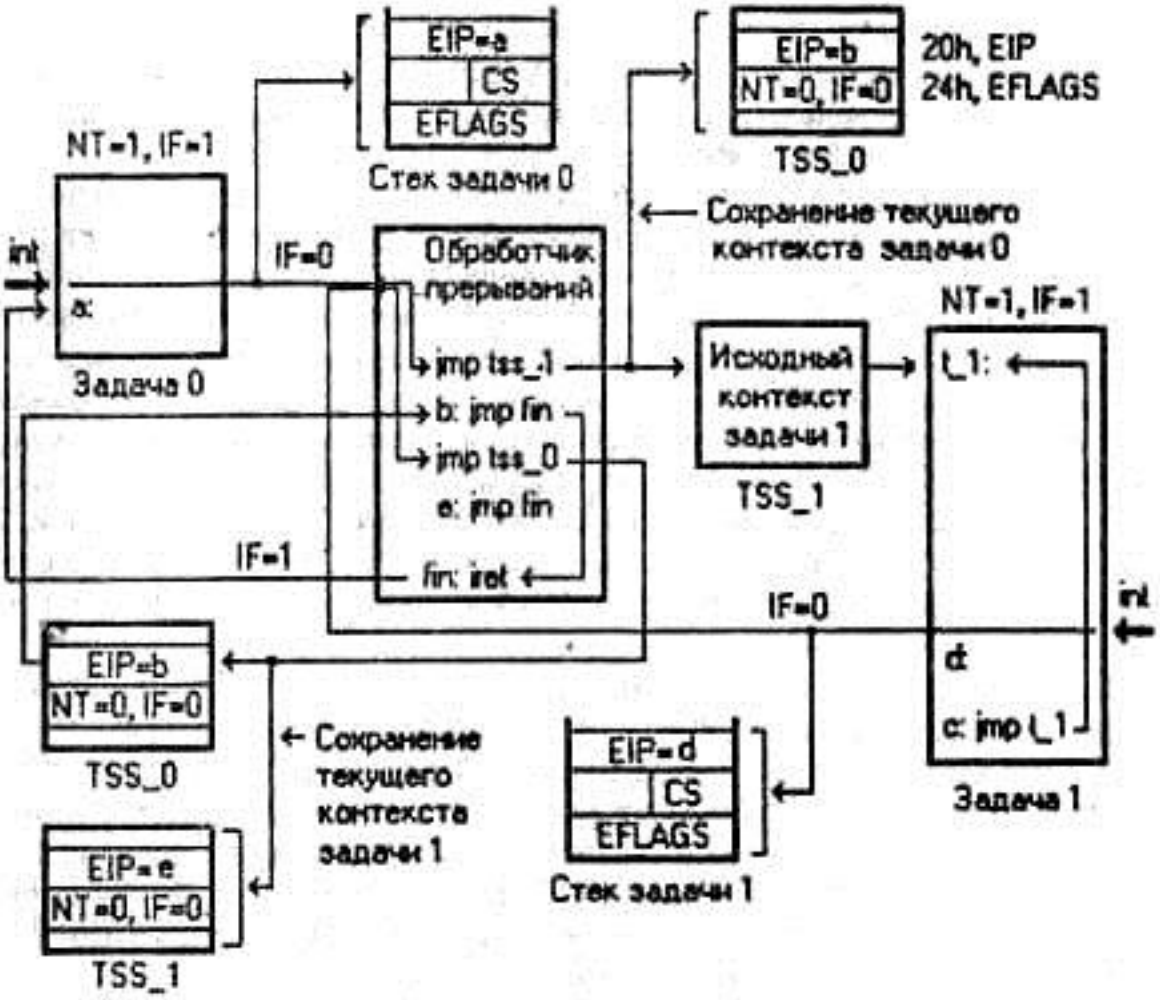

Рисунок 8.2 – Взаимодействие вычислительных объектов в многозадачном комплексе

Таким образом, происходит переключение не на задачу 0, а на обработчик. После каждой команды переключения jmp в обработчике стоит команда перехода на завершающую команду iret. Поскольку флаг IF сброшен, команда iret выполняет обычный возврат через стек текущей задачи, т.е. задачи 0. В результате происходит возврат в задачу 0 в ту точку, где она была прервана при переключении на задачу 1.

Все три задачи нашего программного комплекса вполне равноправны, поэтому, подавая соответствующие команды с клавиатуры, можно переключать их в произвольном порядке. Правда, в программе не предусмотрена защита от повторного переключения на активную задачу. Если попытаться выполнить такое переключение, будет возбуждено исключение общей защиты. Не предусмотрено также завершение задач 1 или 2 – они прекращают работу при завершении главной задачи.

Отмеченные выше недостатки являются следствием примитивности нашей системы, которая предназначена не для изучения принципов построения многозадачных систем, а лишь для демонстрации техники переключения задач и знакомства с соответствующими средствами микропроцессора. Однако и с этой точки зрения приведенный пример далек от совершенства. Одним из слабых его мест является использование всеми задачами комплекса общего сегмента данных. Так, обработчик прерываний выполняет переход на задачи через ячейки с селекторами TSS, расположенные в сегменте данных главной задачи; задачи 1 и 2 выводят сообщения, также хранящиеся в общем сегменте данных. Казалось бы, задачи 1 и 2 вполне могут использовать собственные сегменты данных, так как при переключении задач происходит сохранение старого контекста (включая сегментные регистры) и загрузка нового. Однако вызов обработчика аппаратных прерываний происходит без переключения контекста (заменяется только содержимое CS:IP); обработчик фактически работает на контексте прерванной задачи. Поэтому манипуляции с сегментными регистрами в задаче могут оказаться фатальными для се работоспособности. Впрочем, здесь нет ничего нового: и в реальном режиме при входе в обработчик прерываний сегментные регистры адресуют сегменты прерванного процесса.

Рассмотрим пример, в котором несколько снижена взаимозависимость операционных сред задач и обработчика прерываний.

Ниже описаны лишь изменения, внесенные в программу примера 8.1.

Из общего сегмента данных удалены ячейки t0_addr, tl_addr и t2_addr с адресами TSS задач, а также сообщения задач mesl и mes2. Адреса TSS задач перенесены в процедуру обработчика прерываний, а сообщения задач – в процедуры задач.

Пример 8.2 – Модифицированная программа обработчика прерываний клавиатуры

```
proc new_09h
   push ax
   push dx
   ;Чтение скан-кода
   in   al, 60h ; Прочитать скан-код нажатой клавиши,
   mov  dl, al
   ;Обработка прерывания клавиатуры
   in   al, 61h
   or   al, 80h ; Установить бит 7 порта 61h
   out  61h, al
```

```
   and  al, 7Fh ; Восстанавливаем бит 7
   out  61h, al
   ;Послать EOI контроллеру прерываний
   mov  al, 20h
   out  20h, al
   ;Проверяем введенный символ
   cmp  dl, 0Bh ; 0Bh = Клавиша <0>
   je   tsk0
   cmp  dl, 2   ; 02h = Клавиша <1>
   je   tsk1
   cmp  dl, 3   ; 03h = Клавиша <2>
   je   tsk2
   jmp  out1
tsk0:
   ;Переключение на задачу 0
   pop  dx
   pop  ax
   jmp  [dword ptr cs:t0]
   jmp  out_09h
tsk1:
   ;Переключение на задачу 1
   pop  dx
   pop  ax
   jmp  [dword ptr cs:t1]
   jmp  out_09h
tsk2:
   ;Переключение на задачу 2
   pop  dx
   pop  ax
   jmp  [dword ptr cs:t2]
out_09h:
   ;Выход из обработчика
   db 66h       ; Префикс замены размера операнда
   iret
out1:
   pop  dx
   pop  ax
   jmp  out_09h

   t0   dw 0,40
   t1   dw 0,48
   t2   dw 0,56
endp
```

Команды вида jmp dword ptr cs:t0 с заменой сегмента и адресацией через сегмент команд возможны лишь при условии, что сегмент команд объявлен с разрешением исполнения и чтения (тип дескриптора 5). В первых программах, где сегмент команд объявлялся только исполняемым (тип 4), такая адресация привела бы к нарушению общей защиты. Байт атрибутов 1 должен иметь значение 9Ah.

При инициализации сегментов состояния задач изменено содержимое DS: поскольку поля данных задач размещены в их же процедурах, в DS загружается селектор сегмента команд:

Пример 8.3 – Модифицированная инициализация полей TSS

```
; Заполним поля TSS 1
mov  [word ptr tss_1+4Ch], 16                ; CS
mov  [word ptr tss_1+20h], offset task1      ; IP
mov  [word ptr tss_1+50h], 24                ; SS
mov  [word ptr tss_1+38h], 256               ; SP
mov  [word ptr tss_1+54h], 16                ; DS
mov  [word ptr tss_1+48h], 32                ; ES

sti
pushfd
pop  eax

mov  [dword ptr tss_1+24h], eax              ; EFLAGS
; Заполним поля TSS 2
mov  [word ptr tss_2+4Ch], 16                ; CS
mov  [word ptr tss_2+20h], offset task2      ; IP
mov  [word ptr tss_2+50h], 24                ; SS
mov  [word ptr tss_2+38h], 512               ; SP
mov  [word ptr tss_2+54h], 16                ; DS
mov  [word ptr tss_2+48h], 32                ; ES
mov  [dword ptr tss_2+24h], eax              ; EFLAGS
```

В задачах 1 и 2 вывод сообщений на экран выполняется непосредственно из сегмента команд. Поскольку регистры DS и ES настраиваются автоматически в процессе переключения, требуется только настроить SI и DI (для выполнения команд обработки цепочек):

Пример 8.4 – Модифицированная процедура задачи 1

```
proc task1
a1:
   mov  ah, 71h ; Атрибут символов
   mov  cx, 2
a2:
   push cx
   mov  di, 800
   mov  cx, mes1_len
   mov  si, offset mes1
   cld
   ;Вывод на экран
a3:
   lodsb
   stosw
   loop a3
```

```
   ;Задержка
   mov  cx, 0FFFFh
a4:
   loop a4
   pop  cx
   mov  ah, 17h
   loop a2
   jmp  a1
   mes1 db 22,22,22," task_1 ",22,22,22 ; Строка выводимая задачей
1
  mes1_len = $-mes1                    ; Длинна строки
endp
```

Аналогично выглядит и задача task2.

Введем теперь в программу защиту от повторного переключения на текущую задачу. Фактически нам надо после каждого нажатия клавиш <0>, <1> или <2> выяснять, какая задача является текущей и не допускать переключения, если с клавиатуры случайно подана команда переключения на ту задачу, которая как раз выполняется. Узнать, какая задача является текущей, просто: достаточно прочитать содержимое регистра задачи TR. В нем всегда находится селектор сегмента состояния текущей задачи. Преобразуем процедуру обработчика прерывания от клавиатуры, введя в нее анализ регистра TR.

Ниже приводится только модифицированный обработчик прерываний от клавиатуры. Остальной текст программы не изменился.

Пример 8.5 – Обработчик прерывания от клавиатуры с анализом регистра состояния задачи

```
proc new_09h
   push ax
   push dx
   ;Чтение скан-кода
   in   al, 60h ; Прочитать скан-код нажатой клавиши,
   mov  dl, al
   ;Обработка прерывания клавиатуры
   in   al, 61h
   or   al, 80h ; Установить бит 7 порта 61h
   out  61h, al
   and  al, 7Fh ; Восстанавливаем бит 7
   out  61h, al
   ;Послать EOI контроллеру прерываний
   mov  al, 20h
   out  20h, al
   ;Проверяем введенный символ
   cmp  dl, 0Bh ; 0Bh = Клавиша <0>
   je   tsk0
```

```
   cmp  dl, 2    ; 02h = Клавиша <1>
   je   tsk1
   cmp  dl, 3    ; 03h = Клавиша <2>
   je   tsk2
   jmp  out1
tsk0:
   ;Переключение на задачу 0
   str  ax       ; Получим текущее значение TR
   cmp  ax, 40   ; Сревним с селектором TSS 0
   je   out1
   pop  dx
   pop  ax
   jmp  [dword ptr cs:t0]
   jmp  out_09h
tsk1:
   ;Переключение на задачу 1
   str  ax       ; Получим текущее значение TR
   cmp  ax, 48   ; Сревним с селектором TSS 1
   je   out1
   pop  dx
   pop  ax
   jmp  [dword ptr cs:t1]
   jmp  out_09h
tsk2:
   ;Переключение на задачу 2
   str  ax       ; Получим текущее значение TR
   cmp  ax, 56   ; Сревним селектором TSS 2
   je   out1
   pop  dx
   pop  ax
   jmp  [dword ptr cs:t2]
out_09h:
   db 66h        ; Префикс замены размера операнда
   iret
out1:
   pop  dx
   pop  ax
   jmp  out_09h

   t0   dw 0,40
   t1   dw 0,48
   t2   dw 0,56
endp
```

Если обработчик фиксирует нажатие клавиши <0> (из порта 60h получен скен-код 0Bh), выполняется чтение TR, для чего предусмотрена специальная команда str. Полученное значение сравнивается с известным нам селектором сегмента состояния задачи 0 (число 40). В случае равенства осуществляется выход из обработчика без выполнения каких-либо действий. Точно так же обрабатываются команды <1> и <2>.

4 Контрольные вопросы

1. Каково взаимодействие вычислительных объектов в процессе переключения задач?
2. Как осуществляется вызов обработчика прерываний от клавиатуры?
3. Как инициализируются сегменты состояния задачи?
4. Как ввести в программу защиту от повторного переключения на текущую задачу?
5. Какова роль флага NT в многозадачной среде?

5 Задание для самостоятельной работы

1. Изучить теоретические основы использования многозадачного режима и переключения задач по прерыванию от клавиатуры.
2. Набрать и отладить и протестировать предложенную программу (пример 8.1) с использованием переключения задач. При этом обратить внимание на следующие этапы:
   - 2.1. Определение таблицы дескрипторов и дескрипторов используемых сегментов.
   - 2.2. Определение таблицы прерываний.
   - 2.3. Определение переменных для переключения задач и маскирования прерываний.
   - 2.4. Определение и структура обработчика системного таймера (IRQ0).
   - 2.5. Определение и структура обработчика прерывания клавиатуры (IRQ1).
   - 2.6. Подготовка дескрипторов сегментов TSS и TSS задач
   - 2.7. Перепрограммирование ведущего контроллера прерываний.
   - 2.8. Маскирование прерывания ведущего и ведомого контроллера.
   - 2.9. Определение и структура задач.
   - 2.10. Процесс переключение между задачами.
   - 2.11. Восстановление настроек контроллера прерываний.
   - 2.12. Восстановление масок прерываний.
3. Отладить и протестировать полученную программу.
4. Модифицировать пример 8.1 в соответствии с примером 8.2, отладить и протестировать программу.
5. Модифицировать пример 8.1 в соответствии с примером 8.3, отладить и протестировать программу.
6. Оформить отчёт.

5 Содержание отчета

1. Титульный лист.
2. Краткое теоретическое описание.
3. Задание на лабораторную работу.
4. Листинг программы.
5. Результаты выполнения работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Аблязов Р.З. Программирование на ассемблере на платформе x86-64 [Электронный ресурс]/ Аблязов Р.З.— Электрон. текстовые данные.— Саратов: Профобразование, 2017.— 304 c.— Режим доступа: http://www.iprbookshop.ru/63951.html.— ЭБС «IPRbooks», по паролю
2. Борисенко В.В. Основы программирования [Электронный ресурс]/ Борисенко В.В.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 323 c.— Режим доступа: http://www.iprbookshop.ru/22427.— ЭБС «IPRbooks», по паролю
3. Введение в программные системы и их разработку [Электронный ресурс]/ С.В. Назаров [и др.].— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 649 c.— Режим доступа: http://www.iprbookshop.ru/52145.— ЭБС «IPRbooks», по паролю
4. Журавлёва И.А. Системное и прикладное программное обеспечение [Электронный ресурс]: лабораторный практикум/ Журавлёва И.А., Корнеев П.К.— Электрон. текстовые данные.— Ставрополь: Северо-Кавказский федеральный университет, 2017.— 132 c.— Режим доступа: http://www.iprbookshop.ru/69432.html.— ЭБС «IPRbooks»
5. Котельников Е.В. Введение во внутреннее устройство Windows [Электронный ресурс]/ Котельников Е.В.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 260 c.— Режим доступа: http://www.iprbookshop.ru/16700.— ЭБС «IPRbooks», по паролю
6. Крищенко, В.А. Основы программирования в ядре операционной системы GNU/Linux: учеб.пособие [Электронный ресурс] : учебное пособие / В.А. Крищенко, Н.Ю. Рязанова. — Электрон.дан. — М. : МГТУ им. Н.Э. Баумана (Московский государственный технический университет имени Н.Э. Баумана), 2010. — 36 с. — Режим доступа: http://e.lanbook.com/books/element.php?pl1_id=58435, по паролю
7. Малявко А.А. Системное программное обеспечение. Формальные языки и методы трансляции. Часть 3 [Электронный ресурс]:

учебное пособие/ Малявко А.А.— Электрон. текстовые данные.— Новосибирск: Новосибирский государственный технический университет, 2012.— 120 с.— Режим доступа: http://www.iprbookshop.ru/45019.— ЭБС «IPRbooks», по паролю

8. Мамойленко С.Н. Системное программное обеспечение [Электронный ресурс]: учебно-методическое пособие/ Мамойленко С.Н., Ефимов А.В.— Электрон. текстовые данные.— Новосибирск: Сибирский государственный университет телекоммуникаций и информатики, 2018.— 33 с.— Режим доступа: http://www.iprbookshop.ru/84080.html.— ЭБС «IPRbooks», по паролю

9. Назаров С.В. Современные операционные системы [Электронный ресурс]/ Назаров С.В., Широков А.И.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 351 с.— Режим доступа: http://www.iprbookshop.ru/15837.— ЭБС «IPRbooks», по паролю

10. Привалов И.М. Основы аппаратного и программного обеспечения [Электронный ресурс]: учебное пособие/ Привалов И.М.— Электрон. текстовые данные.— Ставрополь: Северо-Кавказский федеральный университет, 2015.— 145 с.— Режим доступа: http://www.iprbookshop.ru/63113.html.— ЭБС «IPRbooks», по паролю

11. Управление процессами в операционных системах Windows и Linux [Электронный ресурс]: методические указания к выполнению лабораторных работ для студентов бакалавриата по направлению подготовки 09.03.01 Информатика и вычислительная техника/ — Электрон. текстовые данные.— М.: Московский государственный строительный университет, ЭБС АСВ, 2015.— 48 с.— Режим доступа: http://www.iprbookshop.ru/30450.— ЭБС «IPRbooks», по паролю