



ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
“МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПРИБОРОСТРОЕНИЯ И ИНФОРМАТИКИ”

**Кафедра “Персональные
компьютеры и сети”**



РОЩИН А.В.

СИСТЕМНОЕ ПРОГРАММНОЕ
ОБЕСПЕЧЕНИЕ

**Особенности программирования 32-разрядных
процессоров**

Учебное пособие



Москва
2008

УДК 681.3

Рецензенты:
профессор Зеленко Г.В.
доцент Булгаков М.В.

Системное программное обеспечение. Особенности программирования 32-разрядных процессоров. Учебное пособие/ А.В.Рощин. – М.: МГУПИ, 2008. – 148 с.: ил.

Рекомендовано Ученым Советом МГУПИ в качестве учебного пособия для специальности 230101.

Настоящее учебное пособие предназначено для подготовки студентов различных вычислительных специальностей, изучающих работу с 32-разрядными микропроцессорами семейства x86 в реальном режиме. Для специальности 230101 эта работа может использоваться в курсах "Системное программное обеспечение", "Проектирование микропроцессорных систем", "Организация ввода-вывода".

В пособии описаны особенности работы с 32-разрядными микропроцессорами семейства x86 с точки зрения программиста, рассмотрены расширенные возможности адресации таких процессоров – использование 32-разрядных операндов и режима линейной адресации в реальном режиме. Даны основные сведения по отладке программ на ассемблере, как с использованием программ-отладчиков, так и с использованием аппаратных средств процессора. Рассмотрены соответствующие примеры.

© А.В.Рощин. 2008
© МГУПИ. 2008

		Содержание	Стр.
		Введение	4
Глава	1	Особенности работы с 32-разрядными процессорами	5
	1.1	Особенности 32-разрядных процессоров	5
	1.2	Первое знакомство с защищенным режимом	17
	1.3	Вопросы для самопроверки	38
Глава	2	Использование 32-разрядной адресации в реальном режиме	39
	2.1	Линейная адресация данных в реальном режиме DOS	39
	2.2	Вопросы для самопроверки	74
Глава	3	Определение параметров системы из программы пользователя	76
	3.1	Определение версии операционной системы	76
	3.2	Определение наличия в системе мыши	79
	3.3	Определение частоты процессора	85
	3.4	Определение объема оперативной памяти	86
	3.5	Определение объема доступного дискового пространства	88
	3.6	Вопросы для самопроверки	93
Глава	4	Отладка и тестирование программ	95
	4.1	Отладка и турбо дебаггер	96
	4.2	Меню и диалоговые окна	100
	4.3	Команды локальных меню	112
	4.4	Пример отладки простой программы	122
	4.5	Процессорные средства тестирования программ	136
	4.6	Вопросы для самопроверки	144
		Литература	147

Введение

Предлагаемая работа может рассматриваться как пособие-справочник для студентов осваивающих основы системного программирования для 32-разрядных процессоров.

Пособие состоит из четырех частей. Первая из них посвящена особенностям программирования на языке ассемблера для 32-разрядных микропроцессоров. Дано краткое описание дополнительных команд 32-разрядных микропроцессоров, дополнительные возможности, связанные с использованием 32-разрядных операндов, а также с работой в защищенном режиме.

Во второй части приведены возможности использования 32-разрядной адресации в реальном режиме, а также примеры использования линейной 32-разрядной адресации в реальном режиме.

Третья часть пособия посвящена определению параметров системы из программы пользователя.

В четвертой части изложены основные сведения по отладке программ с помощью турбо дебаггера фирмы Borland, а также некоторые возможности использования аппаратных средств процессора для отладки программ.

1 Особенности работы с 32-разрядными процессорами

1.1 Особенности 32-разрядных процессоров

С появлением 32-разрядных процессоров корпорации Intel (80386, i486, Pentium) значительно расширился спектр возможностей программистов. Официально эти процессоры могут работать в трех режимах: реальном, защищенном и виртуального процессора 8086 (как будет показано ниже, это далеко не все возможные режимы работы) [8].

Каждая следующая модель микропроцессора оказывается значительно совершеннее предыдущей. Так, начиная с процессора i486, арифметический сопроцессор, ранее выступавший в виде отдельной микросхемы, реализуется на одном кристалле с центральным процессором; улучшаются характеристики встроенной кэш-памяти; быстро растет скорость работы процессора. Однако все эти усовершенствования мало отражаются на принципах и методике программирования. Приводимые здесь программы будут одинаково хорошо работать на любом 32-разрядном процессоре. В дальнейшем под термином "процессор" мы будем понимать любую модификацию 32-разрядных процессоров корпорации Intel – от 80386 до Pentium, а также многочисленные разработки других фирм, совместимые с исходными процессорами Intel.

Процессор содержит около 40 программно адресуемых регистров (не считая регистров сопроцессора), из которых шесть являются 16-разрядными, а большая часть остальных – 32-разрядными. Регистры принято объединять в группы: регистры данных, регистры-указатели, сегментные регистры, управляющие регистры, регистры системных адресов, отладочные регистры и регистры тестирования. Кроме того, в отдельную группу выделяют счетчик команд и регистр флагов. На рисунке 1.1 показаны регистры, чаще других используемые в прикладных программах.

Регистры общего назначения и регистры-указатели отличаются от аналогичных регистров процессора 8086 тем, что они являются 32-

разрядными.

Для сохранения совместимости с ранними моделями процессоров допускается обращение к младшим половинам всех регистров, которые имеют те же мнемонические обозначения, что и в микропроцессоре 8086/88 (*AX, BX, CX, DX, SI, DI, BP* и *SP*). Естественно, сохранена возможность работы с младшими (*AL, BL, CL* и *DL*) и старшими (*AH, BH, CH* и *DH*) половинками регистров МП 8086/88. Однако старшие половины 32-разрядных регистров процессора не имеют мнемонических обозначений и непосредственно недоступны. Для того, чтобы прочитать, например, содержимое старшей половины регистра *EAX* (биты 31...16) придется сдвинуть все содержимое *EAX* на 16 разрядов вправо (в регистр *AX*) и прочесть затем содержимое регистра *AX*.

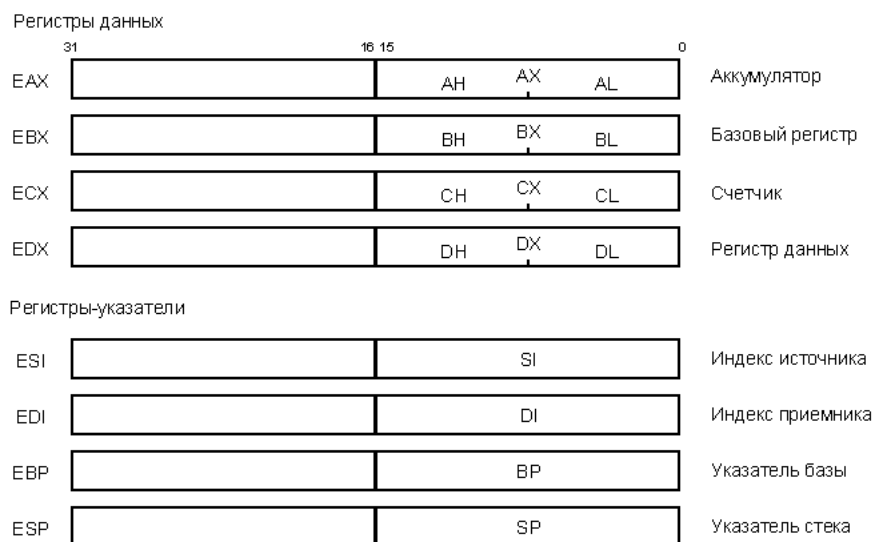


Рисунок 1.1 – Регистры общего назначения

Все регистры общего назначения и указатели программист может использовать по своему усмотрению для временного хранения адресов и данных размером от байта до двойного слова. Так, например, возможно использование следующих команд:

```
mov EAX, 0FFFFFFFFh ; Работа с двойным словом (32 бита)
mov BX, 0FFFFFFFFh ; Работа со словом (16 бит)
```

mov CL,0FFh ; Работа с байтом (8 бит)

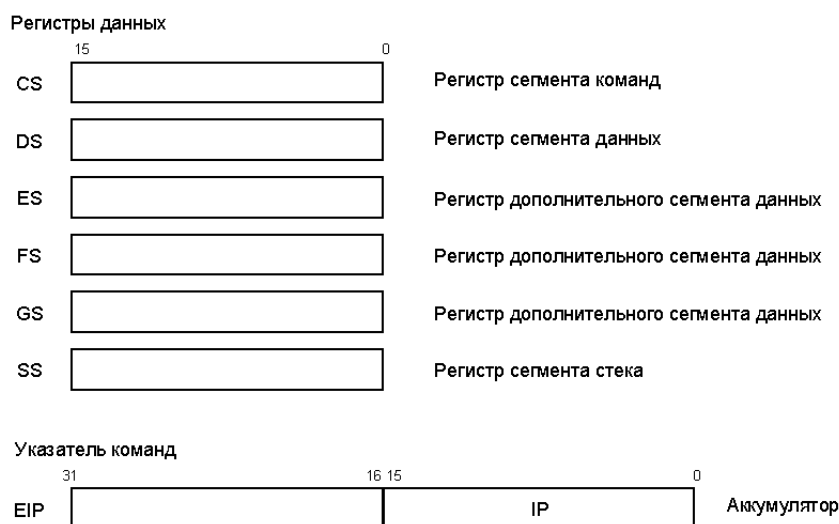


Рисунок 1.2 – Сегментные регистры и указатель команд

Все сегментные регистры, как и в процессоре 8086, являются 16-разрядными. В их состав включено еще два регистра – *FS* и *GS*, которые могут использоваться для хранения сегментных адресов двух дополнительных сегментов данных. Таким образом, при работе в реальном режиме из программы можно обеспечить доступ одновременно к четырем сегментам данных, а не к двум, как при использовании МП 8086.

Регистр указателя команд также является 32-разрядным и обычно при описании процессора его называют *EIP*. Младшие шестнадцать разрядов этого регистра соответствуют регистру *IP* процессора 8086.

Регистр флагов процессоров, начиная с 486 принято называть *EFLAGS*. Дополнительно к шести флагам состояния (*CF*, *PF*, *AF*, *ZF*, *SF* и *OF*) и трем флагам управления состоянием процессора (*TF*, *IF* и *DF*), назначение которых было описано в предыдущих пособиях, он включает три новых флага *NT*, *RF* и *VM* и двухбайтовое поле *IOPL* (рисунок 1.3).

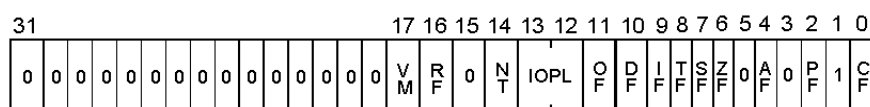


Рисунок 1.3 – Регистр флагов *EFLAGS*

Новые флаги *NT*, *RF* и *VM*, а также поле *IOPL* используются процессором только в защищенном режиме.

Двухразрядное поле привилегий ввода-вывода *IOPL* (Input/Output Privilege Level) указывает на максимальное значение уровня текущего приоритета (от 0 до 3), при котором команды ввода-вывода выполняются без генерации исключений.

Флаг вложенной задачи *NT* (Nested Task) показывает, является ли текущая задача вложенной в выполнение другой задачи. В этом случае *NT*=1. Флаг устанавливается автоматически при переключении задач. Значение *NT* проверяется командой *iret* для определения способа возврата в вызвавшую задачу.

Управляющий флаг рестарта *RF* (Restart Flag) используется совместно с отладочными регистрами. Если *RF*=1, то ошибки, возникшие во время отладки при исполнении команды, игнорируются до выполнения следующей команды.

Управляющий флаг виртуального режима *VM* (Virtual Mode) используется для перевода процессора из защищенного режима в режим виртуального процессора 8086. В этом случае процессор функционирует как быстродействующий МП 8086, но реализует механизмы защиты памяти, страничной адресации и ряд других возможностей.

При работе с процессором программист имеет доступ к четырем управляющим регистрам *CR0...CR3*, в которых содержится информация о состоянии компьютера. Эти регистры доступны только в защищенном режиме для программ, имеющих уровень привилегий 0. Нас будет интересовать лишь регистр *CR0* (рисунок 1.4), представляющий собой слово состояния системы. Более подробно этот управляющий регистр будет рассмотрен в следующей главе.

Для управления режимом работы процессора и указания его состояния используются следующие шесть битов регистра *CR0*:

содержит полный линейный адрес, вызвавший исключительную ситуацию на последней странице, а *CR3* – адрес, указывающий базу каталога страницы.

Регистры системных адресов (см. рисунок 1.5) используются в защищенном режиме работы процессора. Они задают расположение системных таблиц, служащих для организации сегментной адресации в защищенном режиме.

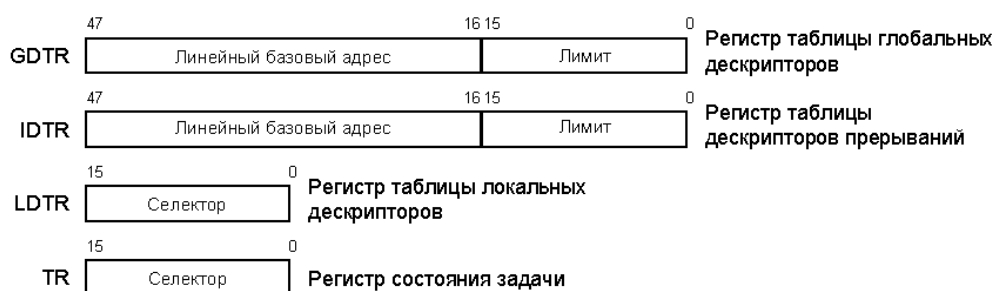


Рисунок 1.5 – Регистры системных адресов

В состав процессора входят четыре регистра системных адресов:

GDTR (Global Descriptor Table Register) – регистр таблицы глобальных дескрипторов для хранения линейного базового адреса и границы таблицы глобальных дескрипторов.

IDTR (Interrupt Descriptor Table Register) – регистр таблицы дескрипторов прерываний для хранения линейного базового адреса и границы таблицы дескрипторов прерываний.

LDTR (Local Descriptor Table Register) – регистр таблицы локальных дескрипторов для хранения селектора сегмента таблицы локальных дескрипторов.

TR (Task Register) – регистр состояния задачи для хранения селектора сегмента состояния задачи.

Отладочные регистры и регистры тестирования будут рассмотрены в главе 4.

Рассмотрим пример простой программы для 32-разрядного

процессора.

Пример 1.1. Программа сложения 32-разрядных операндов

```
.386
Assume CS:code, DS:data, SS:stk

; Простая программа сложения 32-разрядных чисел
data segment para public "data"      ; Сегмент данных
sum dd 0                             ; Переменная для суммы
data ends

stk segment para stack "stack"        ; Сегмент стека
db 256 dup (?)                        ; Буфер для стека
stk ends

code segment para public "code" use16 ; Сегмент кода
begin:
    mov ax,data    ; Адрес сегмента данных в регистр AX
    mov ds,ax      ; Запись AX в DS
; Основной фрагмент программы
    mov eax,12345678h ; Первый 32-разрядный операнд
    add eax,87654321h ; Второй 32-разрядный операнд
    mov dword ptr sum,eax ; Запись результата в sum
; Завершение программы
    mov ax,4C00h ; Функция завершения программы
    int 21h      ; Функция Dos
code ends
END begin
```

Поскольку в данном примере обрабатываются 32-разрядные числа, в текст программы необходимо включить директиву `.386`, разрешающую использование команд 32-разрядных процессоров. Кроме того, при компоновке программы с помощью программы **tlink.exe** следует указать ключ `/3` для разрешения 32-разрядных операций.

Если рассмотреть листинг этой программы, можно увидеть как команды МП 8086 для работы с 16-разрядными операндами, так и команды МП 386 для работы с 32-разрядными операндами. Для облегчения текста из протокола трансляции удалены строчные комментарии.

```
1  .386
2  Assume    CS:code, DS:data, SS:stk
3
```

```

4      ; Простая программа сложения 32-разрядных чисел
5      00000000 data segment para public "data"
6      00000000 00000000 sum dd 0
7      00000004 data ends
8
9      00000000 stk segment para stack "stack"
10     00000000 0100*(??) db 256 dup (?)
11     00000100 stk ends
12
13     0000 code segment para public "code" use16
14     0000 begin:
15     0000 B8 0000s mov ax,data
16     0003 8E D8 mov ds,ax
17     ; Основной фрагмент программы
18     0005 66| B8 12345678 mov eax,12345678h
19     000B 66| 05 87654321 add eax,87654321h
20     0011 66| 67| A3 00000000r mov dword ptr sum,eax
21     ; Завершение программы
22     0018 B8 4C00 mov ax,4C00h
23     001B CD 21 int 21h
24     001D code ends
25     END begin

```

В строках 15 и 16 листинга используется команда засылки операнда в аккумулятор (B8h). Однако в строке 18 наличие перед кодом этой команды префикса замены размера операнда (код 66h) определяет, что длина операнда равна 32 бита, и, следовательно, используется регистр *EAX*. Префикс замены размера операнда включается в объектный модуль транслятором автоматически, если в программе указано мнемоническое обозначение 32-разрядного регистра, например, *EAX*.

При отладке этой программы используется отладчик фирмы Borland (турбо дебаггер), использование которого для отладки программ подробно описывается в главе 4.

Для индикации содержимого 32-разрядных регистров требуется провести дополнительную настройку отладчика. Запустив отладчик, надо выбрать *Основное меню*→*View*→*Registers*. При этом откроется окно индикации содержимого регистров процессора. Затем необходимо вызвать локальное меню этого окна, нажав *ALT-F10*, выбрать в открывшемся меню

пункт *Registers 32 bit* и нажать *Enter*. После этого стоявшее по умолчанию в этом пункте *No* сменится на *Yes*. Это обеспечит вывод на экран содержимого полных 32-разрядных регистров *EAX...ESP* взамен 16-разрядных регистров *AX...SP*. Окно отладчика с исходным состоянием программы и переменных показано на рисунке 1.6.



Рисунок 1.6 – Окно отладчика с исходным состоянием программы и переменных

Для иллюстрации выполнения 32-разрядного сложения надо выполнить программу до команды пересылки содержимого *EAX* в переменную *sum* включительно (строка программы 20). Для этого следует 5 раз нажать клавишу F7, которая вызывает покомандное выполнение программы. Результат такого выполнения показан на рисунке 1.7.

На рисунке 1.6 видно, что содержимое аккумулятора в окне просмотра регистров процессора и содержимое переменной *sum* в окне просмотра переменных нулевые. На рисунке 1.7 содержимое и аккумулятора и указанной переменной уже равно 99999999h (или 2576980377 десятичных), что является результатом сложения 12345678h и 87654321h.

Кроме значения рассматриваемой переменной в окне просмотра

переменных указан еще тип переменной (dword) и ее адрес (6015:0000).

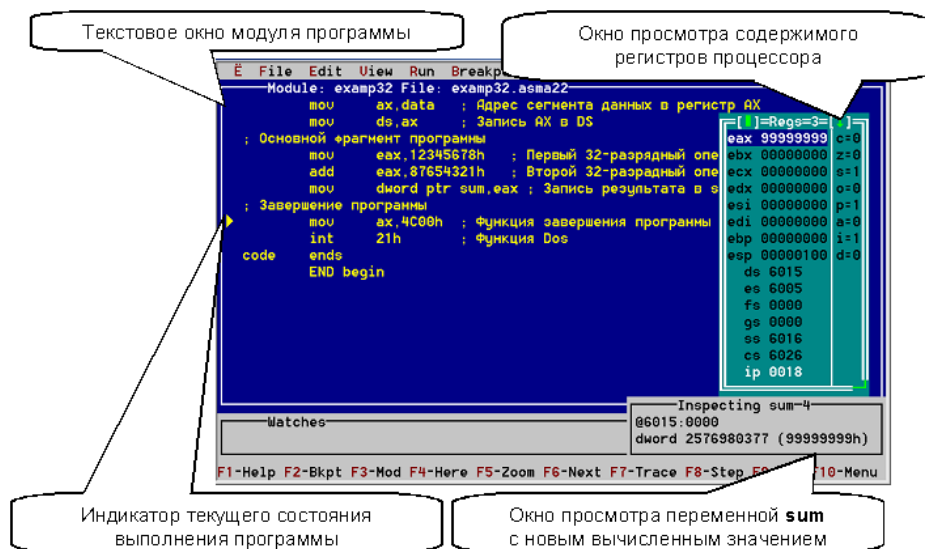


Рисунок 1.7 – Окно отладчика с результатом выполнения 32-разрядного сложения

В рассмотренном примере используются уже известные нам команды. Однако в систему команд современных процессоров включен ряд новых команд, выполнение которых не поддерживается процессором 8086. Некоторые из этих команд впервые появились в процессоре 80386, другие – в процессорах i486 или Pentium. Ниже приведен список этих команд.

Команды общего назначения

bound – проверка индекса массива относительно границ массива.

bsf/bsr – команды сканирования битов.

bt/btc/btr/bts – команды выполнения битовых операций.

bswap – изменение порядка байтов операнда.

cdq – преобразование двойного слова в четверное.

cmpsd – сравнение строк по двойным словам.

cmpxchg – сравнение и обмен операндов.

cmpxchg8b – сравнение и обмен 8-битовых операндов.

cuid – идентификация процессора

cwde – преобразование слова в двойное слово с расширением.

enter – создание кадра стека для параметров процедур.

imul reg,imm – умножение операнда со знаком на непосредственное значение.

ins/outs – ввод/вывод из порта в строку.

iretd – возврат из прерывания в 32-разрядном режиме.

j(cc) – команды условного перехода, допускающие 32-битовое смещение.

leave – выход из процедуры с удалением кадра стека, созданного командой *enter*.

lss/lfs/lgs – команды загрузки сегментных регистров.

mov DRx,reg; reg,DRx

mov CRx,reg; reg,CRx

mov TRx,reg; reg,TRx – команды обмена данными со специальными регистрами. В качестве источника или приемника могут быть использованы регистры *CR0...CR3*, *DR0...DR7*, *TR3...TR5*.

movsx/movzx – знаковое/беззнаковое расширение до размера приемника и пересылка.

popa – извлечение из стека всех 16-разрядных регистров общего назначения (*AX, BX, CX, DX, SP, BP, SI, DI*).

popad – извлечение из стека всех 32-разрядных регистров общего назначения (*EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI*).

push imm – запись в стек непосредственного операнда размером байт, слово или двойное слово (например, *push 0FFFFFFFh*).

pusha – запись в стек всех 16-разрядных регистров общего назначения (*AX, BX, CX, DX, SP, BP, SI, DI*).

pushad – запись в стек всех 32-разрядных регистров общего назначения (*EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI*).

rcl/rcl/ror/rol reg/mem,imm – циклический сдвиг на непосредственное

значение.

sar/sal/shr/shl *reg/mem,imm* — арифметический сдвиг на непосредственное значение.

scasd — сканирование строки двойных слов с целью сравнения.

set(cc) — установка байта по условию.

shrd/shld — логический сдвиг с двойной точностью.

stosd — запись двойного слова с строку.

xadd — обмен и сложение.

xlatb — табличная трансляция.

Команды защищенного режима

arpl — корректировка поля *RPL* селектора

clts — сброс флага переключения задач в регистре *CR0*.

lar — загрузка байта разрешения доступа.

lgdt — загрузка регистра таблицы глобальных дескрипторов.

lidt — загрузка регистра таблицы дескрипторов прерываний.

lldt — загрузка регистра таблицы локальных дескрипторов.

lmsw — загрузка слова состояния машины.

lsl — загрузка границы сегмента.

ltr — загрузка регистра задачи.

rdmsr — чтение особого регистра модели.

sgdt — сохранение регистра таблицы глобальных дескрипторов.

sidt — сохранение регистра таблицы дескрипторов прерываний.

sldt — сохранение регистра таблицы локальных дескрипторов.

smsw — сохранение слова состояния.

ssl — сохранение границы сегмента

str — сохранение регистра задачи.

verr — проверка доступности сегмента для чтения.

verw — проверка доступности сегмента для записи.

1.2 Первое знакомство с защищенным режимом

Как уже отмечалось, современные процессоры могут работать в трех режимах: реальном, защищенном и виртуального 86-го процессора. В реальном режиме процессоры функционируют фактически так же, как МП 8086 с повышенным быстродействием и расширенным набором команд. Многие весьма привлекательные возможности процессоров принципиально не реализуются в реальном режиме, который введен лишь для обеспечения совместимости с предыдущими моделями. Все программы, приведенные в предыдущих пособиях по системному программному обеспечению, относятся к реальному режиму и могут с равным успехом выполняться на любом из этих процессоров без каких-либо изменений. Характерной особенностью реального режима является ограничение объема адресуемой оперативной памяти величиной 1 Мбайт.

Только 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 4 Гбайт;
- возможность работы в виртуальном адресном пространстве, превышающем максимально возможный объем физической памяти и достигающей огромной величины 64 Тбайт. 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). Собственно говоря, 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 процессора в нем автоматически устанавливается реальный режим. Переход в защищенный режим осуществляется программно путем выполнения соответствующей последовательности команд. Поскольку многие детали функционирования процессора в реальном и защищенном режимах существенно различаются, программы, предназначенные для защищенного режима, должны быть написаны особым образом. Реальный и защищенный режимы не совместимы! Архитектура современного микропроцессора необычайно сложна. Столь же сложными оказываются и программы, использующие средства защищенного режима. К счастью, однако, 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. Все эти ограничения существенно упрощают освоение защищенного режима

Начнем изучение защищенного режима с рассмотрения простейшей (но, к сожалению, все же весьма сложной) 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 [8]. Рассматривая эту программу, мы познакомимся с основополагающей особенностью защищенного режима –

сегментной адресацией памяти, которая осуществляется совсем не так, как в реальном режиме.

Следует заметить, что для выполнения рассмотренной ниже программы необходимо, чтобы на компьютере была установлена система MS-DOS "в чистом виде" (не в виде сеанса DOS системы Windows). Перед запуском программ защищенного режима следует выгрузить как систему Windows, так и драйверы обслуживания расширенной памяти HIMEM.SYS и EMM386.EXE.

Листинг 1.1 – Программа, работающая в защищенном режиме

```
1 ;*****
2 ; Программа, работающая в защищенном режиме *
3 ;*****
4 .386P ; Разрешение трансляции всех (в том
5       ; числе и привилегированных команд
6       ; процессоров 386 и 486
7 ; Структура для дескриптора сегмента
8 descr  struc
9  lim    dw      0 ; Граница (биты 0 - 15)
10 base_1  dw      0 ; База (биты 0 - 15)
11 base_m  db      0 ; База (биты 16 - 23)
12 attr_1  db      0 ; Байт атрибутов 1
13 attr_2  db      0 ; Граница (биты 16 - 19)
14         ; и атрибуты 2
15 base_h  db      0 ; База (биты 24 - 31)
16 descr  ends
17
18 data    segment    use16
19 ; Таблица глобальных дескрипторов GDT
20 ; Селектор 0 - обязательный нулевой дескриптор
21 gdt_null descr <0,0,0,0,0,0>
22 ; Селектор 8 - сегмент данных
23 gdt_data  descr <data_size-1,0,0,92h,0,0>
24 ; Селектор 16 - сегмент кода
25 gdt_code  descr <code_size-1,0,0,98h,0,0>
26 ; Селектор 24 - сегмент стека
27 gdt_stack descr <255,0,0,92h,0,0>
28 ;Селектор 32 - видеобуфер
29 gdt_screen descr <4095,8000h,0bh,92h,0,0>
30 gdt_size=$-gdt_null ; Размер GDT
31 ;=====
32 ; Поля данных программы
33 pdescr  dq      0 ; Псевдодескриптор для команды lgdt
34 sym     db      1 ; Символ для вывода на экран
35 attr    db     1ah ; Атрибут символа
```

```

36 mes      db      27,'[31;42mReal mode now',27,'[0m',10,13,'$'
37 mes1     db      26 dup(32),'A message in protected mode',27 dup(32),0
38 data_size=$-gdt_null      ; Размер сегмента данных
39 data      ends
40 ;=====
41 text      segment      'code'  use16 ; По умолчанию 16-разрядный режим
42          assume      cs:text,ds:data
43 main      proc
44          xor eax,eax  ; Очистка 32-разр. EAX
45          mov ax,data  ; Инициализация сегментного регистра
46          mov ds,ax    ; для реального режима
47 ; Вычисление 32-битного линейного адреса сегмента данных и загрузка
48 ; его в дескриптор (в EAX уже находится его сегментный адрес) .
49 ; Для умножения его на 16 сдвинем его влево на 4 разряда
50          shl eax,4    ; В EAX - линейный базовый адрес
51          mov ebp,eax  ; Сохранение его в EBP
52          mov bx,offset gdt_data      ; В BX адрес дескриптора
53          mov [bx].base_1,ax  ; Мл. часть базы
54          rol eax,16         ; Обмен старшей и младшей половины EAX
55          mov [bx].base_m,al   ; Средняя часть базы
56 ;Вычисление и загрузка 32-битного линейного адреса сегмента команд
57          xor eax,eax  ; Очистка 32-разр. EAX
58          mov ax,cs    ; Адрес сегмента команд
59          shl eax,4    ; В EAX - линейный базовый адрес
60          mov bx,offset gdt_code      ; В BX адрес дескриптора
61          mov [bx].base_1,ax  ; Мл. часть базы
62          rol eax,16         ; Обмен старшей и младшей половины EAX
63          mov [bx].base_m,al   ; Средняя часть базы
64 ;Вычисление и загрузка 32-битного линейного адреса сегмента стека
65          xor eax,eax
66          mov ax,ss
67          shl eax,4
68          mov bx,offset gdt_stack
69          mov [bx].base_1,ax
70          rol eax,16
71          mov [bx].base_m,al
72 ;Подготовка псевдодескриптора и загрузка его в регистр GDTR
73          mov dword ptr pdescr+2,ebp      ; База GDT (0-31)
74          mov word ptr pdescr,gdt_size-1  ; Граница GDT
75          lgdt  pdescr                    ; Загрузка регистра GDTR
76 ;Подготовка к переходу в защищенный режим
77          cli                            ; Запрет маскир. прерываний
78          mov al,80h                     ; Запрет NMI
79          out 70h,al                     ; Порт КМОП микросхемы
80 ;Переход в защищенный режим
81          mov eax,cr0 ; Чтение регистра состояния
82          or  eax,1   ; Взведение бита 0
83          mov cr0,eax
84 ;*****
85 ;* Теперь процессор работает в защищенном режиме *
86 ;*****

```

```

87 ; Загрузка в CS селектор сегмента кода, а в IP смещения следующей
88 ; команды (при этом и очищается очередь команд)
89     db 0eah                ; Код команды far jmp
90     dw offset continue    ; Смещение
91     dw 16                 ; Селектор сегмента команд
92 continue:
93 ;Инициализация селектора сегмента данных
94     mov ax,8              ; Селектор сегмента данных
95     mov ds,ax
96 ;Инициализация селектора сегмента стека
97     mov ax,24             ; Селектор сегмента стека
98     mov ss,ax
99 ;Инициализация селектора ES и вывод символов на экран
100    mov ax,32             ; Селектор сегмента видеобuffers
101    mov es,ax
102    mov ebx,800           ; Начальное смещение на экране
103;Вывод сообщения на экран
104    lea esi,mes1
105    mov ah,attr
106screen:
107    mov al,[esi]
108    or al,al
109    jz scend ; Выход, если нуль (терминатор сообщения)
110    mov es:[bx],ax ; Вывод символа в видеобuffer
111    add ebx,2           ; Следующий адрес на экране
112    inc esi             ; Следующий символ
113    jmp screen          ; Цикл
114 scend:                ; Конец вывода сообщения
115 ;Подготовка перехода в реальный режим
116 ;*****
117 ;Формирование и загрузка дескрипторов для реального режима
118     mov gdt_data.lim,0ffffh ; Запись значения
119     mov gdt_code.lim,0ffffh ; границы в 4 ис-
120     mov gdt_stack.lim,0ffffh ; пользуемых нами
121     mov gdt_screen.lim,0ffffh ; дескриптора
122 ; Для перенесения этих значений в теневые регистры необходимо
123 ; записать в сегментные регистры соответствующие селекторы
124     mov ax,8 ; Загрузка теневого регистра
125     mov ds,ax ; сегмента данных
126     mov ax,24 ; Загрузка теневого регистра
127     mov ss,ax ; сегмента стека
128     mov ax,32 ; Загрузка теневого регистра
129     mov es,ax ; дополнительного сегмента
130 ;Сегментный регистр CS программно недоступен, поэтому его
131 ; загрузку опять выполняем косвенно с помощью искусственно
132 ; сформированной команды дальнего перехода
133     db 0eah ; Код команды дальнего перехода
134     dw offset go ; Смещение
135     dw 16 ; Селектор сегмента кода
136 ;Переключение режима процессора
137 go: mov eax,cr0 ; Чтение cr0

```

```

138     and eax,0fffffffh          ; Сброс бита 0
139     mov cr0,eax                ; Запись cr0
140     db 0eah                    ; Код команды дальнего перехода
141     dw return                  ; Смещение
142     dw text                    ; Сегмент кода
143 ;*****
144 ;* Теперь процессор опять работает в реальном режиме *
145 ;*****
146 ;Восстановление операционной среды реального режима
147 return:
148     mov ax,data ; Инициализация сегментных регистров
149     mov ds,ax   ; данных и
150     mov ax,stk  ; стека
151     mov ss,ax   ; в реальном режиме
152 ;Мы не восстанавливает содержимое SP, так как при таком мягком (без
153 ; сброса) переходе в реальный режим SP не разрушается
154 ;Разрешение всех прерываний
155     sti          ; Разрешение маск. прерываний
156     mov al,0     ; Сброс бита 7 порта 70 КМОП -
157     out 70h,al   ; разрешение NMI
158 ;Вывод сообщения в реальном режиме
159     mov ah,9     ; Вывод сообщения
160     mov dx,offset mes ; функцией DOS
161     int 21h
162 ;Ожидание нажатия клавиши
163     xor ah,ah
164     int 16h
165 ; Завершение программы
166     mov ax,4c00h
167     int 21h
168 main     endp
169 code_size=$-main
170 text     ends
171 stk      segment      stack 'stack'
172          db           256 dup(0)
173 stk      ends
174          end          main

```

К тексту программы добавлены номера строк, которые облегчат описание отдельных команд в тексте. Следует иметь в виду, что перед трансляцией показанного текста, необходимо удалить все номера строк, так как компилятор на каждый номер будет выдавать ошибку.

32-разрядные микропроцессоры отличаются расширенным набором команд, часть которых относится к привилегированным. Для того, чтобы разрешить транслятору обрабатывать эти команды, в текст программы

необходимо включить директиву ассемблера .386P.

Программа начинается с описания структуры дескриптора сегмента. В отличие от реального режима, в котором сегменты определяются их базовыми адресами, задаваемыми программистом в явной форме, в защищенном режиме для каждого сегмента программы должен быть определен дескриптор – 8-байтовое поле, в котором в определенном формате записываются базовый адрес сегмента, его длина и некоторые другие характеристики (рисунок 1.8) [8].

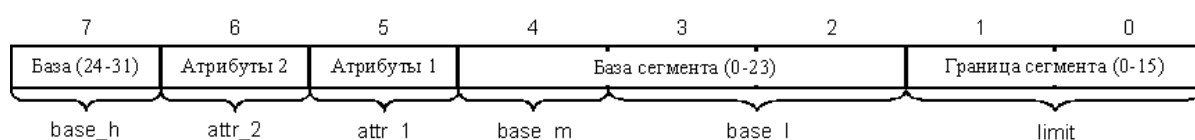


Рисунок 1.8 – Дескриптор сегмента

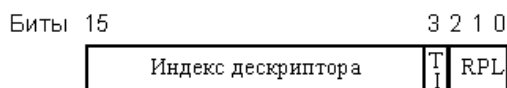


Рисунок 1.9 – Селектор дескриптора

Теперь для обращения к требуемому сегменту программист заносит в сегментный регистр не сегментный адрес, а так называемый селектор (рисунок 1.9), в состав которого входит номер (индекс) соответствующего сегмента 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. Другие поля селектора, которые для нашего случая принимают значения 0, будут описаны ниже.

Структура *descr* (строка 8 листинга 1.1) предоставляет шаблон для дескрипторов сегментов, облегчающий их формирование. Сравнивая описание структуры *descr* в программе с рисунком 1.8, нетрудно заметить их соответствие друг другу.

Рассмотрим вкратце содержимое дескриптора. 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. Как видно из рисунка 1.8, 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 старший бит байта атрибутов 2, называемый битом дробности. Если он равен 0, граница указывается в байтах; если 1 – в блоках по 4 килобайта.

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. Однако если 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 рассмотренном примере 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* – строка 10 листинга) и двумя байтами: средним (*base_m* – строка 11 листинга) и старшим (*base_h* – строка 15 листинга).

В байте атрибутов 1 задается ряд характеристик сегмента. Не вдаваясь пока в подробности этих характеристик, укажем, что в рассмотренном примере используются сегменты двух типов: сегмент команд, для которого байт *attr_1* (строка 12 листинга) 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, бит дробности). Для всех наших сегментов значение этого полубайта равно 0.

Сегмент данных *data* (строка 18 листинга), который для удобства изучения функционирования программы расположен в начале программы, до сегмента команд, объявлен с типом использования *use16* (так же будет объявлен и сегмент команд). Этот описатель объявляет, что в данном сегменте будут по умолчанию использоваться 16-битовые адреса. Если бы мы готовили нашу программу для работы под управлением операционной системы защищенного режима, реализующей все возможности микропроцессора, тип использования был бы *use32*. Однако наша программа будет запускаться под управлением DOS, которая работает в реальном режиме с 16-битовыми адресами и операндами.

Сегмент данных начинается с описания важнейшей системной структуры – таблицы глобальных дескрипторов. Как уже отмечалось выше, обращение к сегментам в защищенном режиме возможно только 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, обозначаемой *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 эти 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 [8], например:

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

В дескрипторе *gdt_data* (строка 23 листинга), описывающем сегмент данных программы, заполняется поле границы сегмента (фактическое значение размера сегмента *data_size* будет вычислено компилятором, строка 30 листинга), а также байт атрибутов 1. Код 92h говорит о том, что это сегмент данных с разрешением записи и чтения. Базу сегмента, т.е. физический адрес его начала, придется вычислить программно и занести в дескриптор уже на этапе выполнения.

Дескриптор *gdt_code* (строка 25 листинга) 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* (строка 27 листинга) сегмента стека имеет, как и любой сегмент данных, код атрибута 92h, что разрешает его чтение и запись, и явным образом заданную границу 255 байтов, что соответствует размеру стека. Базовый адрес сегмента стека так же будет вычислен на этапе выполнения программы.

Последний дескриптор *gdt_screen* (строка 29 листинга) описывает страницу 0 видеобуфера. Размер видеостраницы, как известно, составляет 4096 байтов, поэтому в поле границы указано число 4095. Базовый

физический адрес страницы известен, он равен B8000h. Младшие 16 разрядов базы (число 8000h) заполняют слово *base_1* 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 ее размер (точнее, границу). Размер GDT определяется на этапе трансляции в строке 30.

Назначение оставшихся строк сегмента данных станет ясным в процессе рассмотрения программы.

Сегмент команд *text* (строка 41 листинга) начинается, как и всегда, оператором *segment*, в котором указывается тип использования *use16*, так как мы составляем 16-разрядное приложение. Указание описателя *usel6*. Не запрещает использовать в программе 32-битовые регистры.

Фактически вся программа примера, 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. Сначала производится очистка 32-разрядного аккумулятора (строка 44), так как в нем будет сформирован позднее базовый 32-разрядный адрес (фактически эта команда нужна для очистки старшего слова расширенного аккумулятора). После обычной инициализации сегментного регистра *DS* (строки 45, 46), которая позволит нам обращаться к полям данных программы (в реальном режиме!) выполняется сдвиг на 4 разряда содержимого регистра *EAX*. Эта операция выполняется командой *shl EAX,4*. Команда сдвигает влево содержимое 32-разрядного аккумулятора

на указанное константой число бит (4). Следующая команда сохраняет получившееся 32-разрядное значение адреса в регистре *EBP*. После этого в регистр *BX* помещается смещение дескриптора сегмента кода. Следующими тремя командами (строки 53 – 55) младшее и старшее слова регистра *EAX* отправляется в поля *base_l* и *base_m* дескриптора *gdt_data*, соответственно. Аналогично вычисляются 32-битовые адреса сегментов команд и стека, помещаемые в дескрипторы *gdt_code* (строки 57 – 63) и *gdt_stack* (строки 65 – 71).

Следующий этап подготовки к переходу в защищенный режим – загрузка в регистр процессора *GDTR* (Global Descriptor Table Register, регистр 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 псевдодескриптором. Для загрузки *GDTR* предусмотрена 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 1.10.

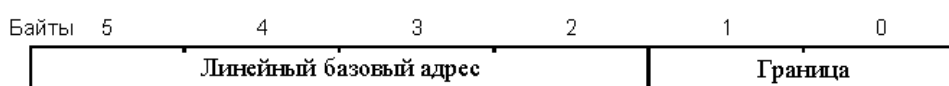


Рисунок 1.10 – Формат псевдодескриптора

В нашем примере 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 строках 73, 74 базовый адрес и граница помещаются в требуемые поля *pdescr*, а в строке 75 командой *lgdt* загружается регистр *GDTR*, сообщая, таким образом, процессору о местонахождении и размер

GDT.

В принципе теперь можно перейти в защищенный режим. Однако мы запускаем нашу программу под управлением DOS (а как ее еще можно запустить?) 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 *int* с определенными номерами, а в защищенном режиме эти команды приведут к совершенно другим результатам. Следовательно, программу, работающую в защищенном режиме, нельзя завершить средствами DOS. Сначала ее надо вернуть в реальный режим.

Возврат в реальный режим можно осуществить сбросом процессора. Действия процессора после сброса определяются одной из ячеек КМОП-микросхемы – байтом состояния отключения, располагаемым по адресу Fh. В частности, если в этом байте записан код Ah, 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 необходимо в ячейку 40h:67h записать адрес возврата, а в байт Fh КМОП-микросхемы занести код Ah. Приведенный способ возврата в реальный режим использовался в процессорах i286 (так как другого способа возврата в нем предусмотрено не было).

В процессорах, начиная с i386, переход из реального режима в защищенный и обратно может осуществляться с использованием

управляющего регистра *CR0*. Так как встретить сейчас «живой» 286 процессор практически не реально, воспользуемся именно таким способом.

Всего в микропроцессорах i386 и выше имеется 4 программно адресуемых управляющих регистра *CR0*, *CR1*, *CR2* и *CR3* [8]. Регистр *CR1* зарезервирован, регистры *CR2* и *CR3* управляют страничным преобразованием, которое в рассматриваемой программе не используется, а регистр *CR0* содержит набор управляющих битов, из которых нам интересны биты 0 (включение и выключение защищенного режима) и 31 (разрешение страничного преобразования).

После сброса процессора оба эти бита сброшены, благодаря чему процессор начинает работать в реальном режиме с выключенным страничным преобразованием. Установка младшего бита *CR0* в 1 переводит процессор в защищенный режим, а сброс его возвращает процессор в реальный режим. Следует отметить, что младшая половина регистра *CR0* совпадает со словом состояния 286 процессора, поэтому команды чтения и записи в регистр *CR0* аналогичны по результату командам *smsw* и *lmsw*, которые сохранены в старших процессорах из соображений совместимости.

Еще один важный шаг, который необходимо выполнить перед переходом в защищенный режим, заключается в запрете всех аппаратных прерываний. Дело в том, что в защищенном режиме процессор выполняет процедуру обработки прерывания иначе, чем 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 аналогично таблице глобальных дескрипторов и располагаемой в программе пользователя (или в операционной системе). В нашем примере такой таблицы нет, и на время

работы программы прерывания придется запретить. Запрет всех аппаратных прерываний осуществляется командой *cli* (строка 77).

В строках 78, 79 в порт 70h засылается код 80h, который запрещает немаскируемые прерывания (которые не запрещаются командой *cli*).

В строках 81...83 осуществляется перевод процессора в защищенный режим. Этот перевод можно выполнить различными способами. В рассматриваемом примере для этого используется команда *mov*. Сначала содержимое управляющего *CR0* регистра считывается в аккумулятор *EAX*, затем его младший бит устанавливается в 1 с помощью команды *or*, затем содержимое аккумулятора опять записывается в управляющий регистр *CR0* с уже модифицированным младшим битом. Все последующие команды выполняются уже в защищенном режиме.

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.

Сегментные регистры		Теневые регистры		
CS	Селектор	База	Граница	Атрибуты
SS	Селектор	База	Граница	Атрибуты
DS	Селектор	База	Граница	Атрибуты
ES	Селектор	База	Граница	Атрибуты
FS	Селектор	База	Граница	Атрибуты
GS	Селектор	База	Граница	Атрибуты

Рисунок 1.11 – Сегментные и теневые регистры

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 (рисунок 1.11). 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, прежде всего, 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 (строки 93 – 101). Но как загрузить селектор в регистр

CS? Для этого можно воспользоваться искусственно сконструированной командой дальнего перехода, которая, как известно, приводит к смене содержимого и *IP*, и *CS*. Строки 89 – 91 демонстрируют эту методику. В реальном режиме мы поместили бы во второе слово адреса 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 программы (строки 100 – 113) является чисто иллюстративным. В нем инициализируется (по правилам защищенного режима!) сегментный регистр *ES* и в видеобуфер выводится сообщение '*A message in protected mode*' (зеленым цветом на синем фоне), так, что оно располагается в середине пятой строки экрана, чем подтверждается правильное функционирование программы в защищенном режиме.

Как уже отмечалось выше, для того, чтобы не нарушить работоспособность DOS, процессор следует вернуть в реальный режим,

после чего можно будет завершить программу обычным образом. Перейти в реальный режим можно разными способами. Можно, например, осуществить сброс процессора, заслав команду FEh в порт 64h контроллера клавиатуры. Эта команда возбуждает сигнал на одном из выводов контроллера клавиатуры, который, в конечном счете, приводит к появлению сигнала сброса на выводе RESET микропроцессора. Этот способ (единственный для 286 процессора) неудобен тем, что для выполнения сброса необходимо несколько микросекунд. Если выполнять таким образом переход в реальный режим достаточно часто, можно потерять много времени.

В нашем примере переход в реальный режим осуществляется простым сбросом младшего бита в управляющем регистре *CR0*.

Если просто перейти в реальный режим сбросом бита 0 в регистре *CR0*, в теневых регистрах останутся дескрипторы защищенного режима, и при первом же обращении к любому сегменту программы возникнет исключение общей защиты, так как ни один из определенных ранее сегментов не имеет границы FFFh. Так как обработка исключений не производится, произойдет сброс процессора и перезагрузка компьютера, так как в данном случае не настраивался байт состояния перезагрузки, и не заполнялись соответствующие ячейки области данных BIOS. Следовательно, перед переходом в реальный режим необходимо исправить дескрипторы всех используемых сегментов: кодов, данных, стека и видеобuffers. Сегментные регистры *FS* и *GS* не использовались, поэтому о них можно не заботиться.

В строках 118 – 121 в поля границ всех четырех дескрипторов записывается значение FFFFh, а в строках 124 – 129 выполняется загрузка селекторов в сегментные регистры, что приводит к перезаписи содержимого теневых регистров. Так как сегментный регистр *CS* программно недоступен, его загрузку приходится опять выполнять с

помощью искусственно сформированной команды дальнего перехода (строки 133 – 135).

После настройки всех использованных в защищенном режиме сегментных регистров, можно сбрасывать бит 0 управляющего регистра *CR0* (строки 137 – 139). После перехода в реальный режим опять надо выполнить искусственно сформированную команду дальнего перехода для того, чтобы очистить очередь команд в блоке предвыборки процессора и загрузить в регистр *CS* вместо хранящегося там селектора обычный сегментный адрес регистра команд (строки 140 – 142).

Искусственно сформированная команда дальнего перехода передает управление на метку *return*.

Теперь процессора опять работает в реальном режиме. При этом, хотя в сегментных регистрах *DS*, *ES* и *SS* остались недействительные для реального режима селекторы, программа пока работает корректно, так как в теневых регистрах находятся правильные линейные адреса (оставшиеся от защищенного режима) и законные для реального режима границы (загруженные в строках 118 – 121). Однако, если в программе встретится любая команда сохранения или восстановления какого-либо сегментного регистра, нормальное выполнение программы нарушится, так как в сегментном регистре окажется не сегментный адрес, как это должно быть в реальном режиме, а селектор. Это значение будет трактоваться процессором как сегментный адрес, что приведет в дальнейшем к неверной адресации соответствующего сегмента.

Если даже в оставшемся тексте программы и нет команд чтения или записи сегментных регистров, неприятности все равно возникнут, так как простой вызов DOS'овского прерывания *int 21h* приведет к этим неприятностям, так как диспетчер DOS выполняет сохранение и восстановление регистров (в том числе и сегментных) при выполнении функций DOS.

Поэтому после перехода в реальный режим необходимо загрузить в используемые далее сегментные регистры соответствующие сегментные адреса. В строках 148 – 151 в регистры *DS* и *SS* записываются сегментные адреса соответствующих сегментов.

При рассмотренном варианте возврата в реальный режим (без сброса процессора) не надо сохранять кадр стека, так как содержимое регистра *SP* в этом случае не разрушается, а регистр *SS* мы уже инициализировали.

Для восстановления работоспособности системы следует также разрешить прерывания (маскируемые – строка 155, немаскируемые – строки 156, 157), после чего программа может продолжаться уже в реальном режиме. В рассмотренном примере для проверки работоспособности системы в этом режиме на экран выводится сообщение '*Real mode now*' с помощью функции DOS 09h. Для наглядности в сообщение включены Esc последовательности для смены цвета символов и фона (красные символы на зеленом фоне). Осуществлена смена цвета символов и фона может быть лишь в том случае, если в DOS установлен драйвер ANSI.SYS. Если после перехода в реальный режим при установленном драйвере ANSI.SYS сообщение будет выведено без изменения цветов, это может говорить об ошибках защищенного режима.

Перед завершением программы, она ожидает ввода с клавиатуры (функция 0 прерывания *int* 16h), чтобы можно было успеть увидеть содержимое экрана.

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

1.3 Вопросы для самопроверки

1. Какие режимы работы поддерживают 32-разрядные процессоры x86?
2. Какие регистры в 32-разрядных микропроцессорах x86 являются 16-разрядными?
3. Какие новые флаги добавились у 32-разрядных микропроцессоров x86?
4. Какие разряды управляющего регистра *CR0* микропроцессора указывают состояние и режимы работы процессора?
5. Что такое бит страничного преобразования?
6. Что такое бит сопроцессора?
7. Для чего нужен бит переключения задачи?
8. Что такое бит эмуляции сопроцессора?
9. Для чего нужен бит присутствия сопроцессора?
10. Что такое бит разрешения защиты?
11. Какие регистры микропроцессора используются для поддержки страничного преобразования?
12. Что такое регистры системных адресов?
13. Для чего нужен регистр таблицы глобальных дескрипторов?
14. Для чего нужен регистр таблицы дескрипторов прерываний?
15. Для чего нужен регистр таблицы локальных дескрипторов?
16. Для чего нужен регистр состояния задачи?
17. Какие действия надо выполнить, чтобы в компилируемой программе можно было использовать 32-разрядные операнды?
18. Какие команды появились в 32-разрядных микропроцессорах (примеры)?
19. Каково главное ограничение реального режима работы процессора?
20. Какие дополнительные возможности появляются в защищенном режиме работы микропроцессора?

2 Использование 32-разрядной адресации в реальном режиме

Большое количество процессоров, используемых в настоящее время, ставит перед программистами проблемы оптимального использования ресурсов конкретного процессора в своих разработках. У изготовителей микропроцессоров стало традицией публиковать описания регистров и команд через Интернет в виде pdf-файлов, но не давать при этом рекомендаций по их применению. Хорошо, если из названия (или описания) можно сделать совершенно определенные выводы о назначении команды или регистра. А если нет?

Столь же вредная традиция — не описывать в общедоступной документации режимы работы, которых современные процессоры имеют великое множество. Безусловным чемпионом в этой области является Intel — значительная часть потенциальных возможностей процессоров класса Pentium и последующих модификаций не используется потребителями, поскольку эти возможности в документации только упоминаются, но не рассматриваются. Программистам приходится искать наработки энтузиастов, которые тратят свое время на углубленное исследование режимов работы процессоров и применения конкретных, плохо описанных изготовителями, команд и регистров процессора [1].

2.1 Линейная адресация данных в реальном режиме DOS

В литературе по программированию описано три режима работы микропроцессоров серии 80x86 — реальный режим (режим совместимости с архитектурой 8086), защищенный режим и режим виртуальных процессоров 8086 (являющийся неким подвидом защищенного режима).

Основной недостаток реального режима состоит в том, что адресное пространство имеет размер всего в 1 Мбайт и при этом сегментировано — «нарезано» на кусочки размером по 64 Кбайт. Одного мегабайта очень мало для современных ресурсоемких прикладных программ (текстовых и

графических редакторов, геоинформационных систем, систем проектирования и т. д.), а сегментация не позволяет нормально работать с видеопамтью и большими массивами данных.

Что можно сказать о защищенном и виртуальном режимах? Многие книги и учебники по микропроцессорам Intel *заканчиваются* главой «Переход в защищенный режим». Недостаток этого режима — необходимость *заново* создавать программное обеспечение для работы с периферийными устройствами на низком уровне, то есть фактически полностью переписывать *все* основные функции DOS. Можно, конечно, использовать Windows, но эта операционная система предназначена для офисных целей и плохо адаптируется к решению задач оперативного управления техническими системами. Кроме того, Windows забирает для собственных нужд изрядную часть ресурсов компьютера и ограничивает доступ к периферийным устройствам.

В некоторых случаях универсальные многозадачные операционные системы типа Windows и Unix неприменимы по причинам, не относящимся напрямую к области вычислительной техники. Первая причина — лицензионные соглашения между изготовителями и потребителями программ. Прочтите внимательно любую лицензию: разработчик программы не несет ответственности *ни за что*. Следовательно, за все сбои и неисправности расплачивается потребитель. Например, за аварию в системе управления транспортом разработчикам этой системы придется отвечать по статьям Уголовного кодекса. Что касается систем военного назначения, то вообще сомнительно, что на таких лицензионных условиях какая-либо программа может быть официально принята в эксплуатацию на территории России.

Вторая причина — огромный объем универсальных операционных систем — десятки миллионов строк на языках высокого уровня! Полностью протестировать такие системы невозможно — у фирмы

Microsoft, например, хватает сил только на доскональную проверку небольшого ядра Windows! Тем более на это не способен потребитель, у которого нет всей документации. Даже в случае открытой системы типа Linux, если документация есть и все исходные коды доступны — попробуйте *доказать* военным или банкирам, что в системе нет скрытых ловушек и «черного хода»!

Создать собственную программу для переключения в защищенный режим и работы в нем — непростая задача. При работе с аппаратурой в защищенном режиме программист должен четко понимать, какими возможностями аппаратуры пользоваться опасно. Например, приводимые в учебниках примеры программ для защищенного режима часто проявляют несовместимость с определенными конфигурациями оборудования, поскольку их авторы не имели достаточно широкой лабораторной базы для тестирования. Дело в том, что периферийные устройства всегда имеют какие-нибудь нестандартные особенности, добавляемые их изготовителями в рекламных целях. При работе в реальном режиме DOS такие особенности не применяются и потому никак не проявляются. Однако они могут показать себя с самой неприятной стороны при переключении в защищенный режим, когда программисту приходится перенастраивать периферийные устройства на новую модель организации оперативной памяти, перезаписывая при этом множество различных регистров аппаратуры. Возможно две ситуации: либо в стандартных регистрах некоторые разряды применяются нестандартным образом, но программисту об этом ничего не известно, либо вообще имеются какие-либо дополнительные регистры, не описанные в документации, но влияющие на режим работы системы. Возникает абсурдная ситуация: простой (реальный) режим работы задается процедурами BIOS фирмы-изготовителя системной платы, которая обычно хорошо осведомлена об особенностях применяемого на этой плате чипсета, а программы для

перехода в защищенный режим вынуждены писать совершенно посторонние люди, не располагающие документацией в полном объеме. В BIOS включено некоторое количество процедур для работы в защищенном режиме, но они охватывают лишь часть необходимых операций.

Вообще говоря, изобилие управляющих регистров в современных персональных компьютерах (их общее количество достигает нескольких тысяч) — явление совершенно ненормальное, теоретически приводящее к увеличению количества возможных режимов работы до бесконечности. Поскольку протестировать функционирование системы в миллиардах различных режимов технически невозможно, разработчики программного обеспечения не могут использовать дополнительные средства, и ограничены несколькими общепринятыми (стандартными) режимами. Чтобы убедиться в этом, достаточно сравнить полный набор команд любого периферийного устройства с реально используемым (например, в BIOS) подмножеством команд данного набора. Большая часть регистров в настоящее время в принципе не нужна — установкой режима работы периферийного устройства должен заниматься его встроенный специализированный процессор, а не центральный процессор компьютера. Однако переход на новые технологии произойдет, вероятно, только после очередного кризиса в развитии компьютерной индустрии, а пока что приходится приспосабливаться к сложившейся ситуации.

Изложенные выше причины приводят к тому, что программисты вынуждены искать различные обходные пути. Один из возможных приемов — использование линейной адресации памяти. Линейная адресация — это наиболее простой, с точки зрения программиста, способ работы непосредственно с аппаратурой ЭВМ (логические адреса при этом совпадают с физическими). Различия в организации памяти в реальном, защищенном и линейном режимах работы процессора иллюстрирует рисунок 2.1.

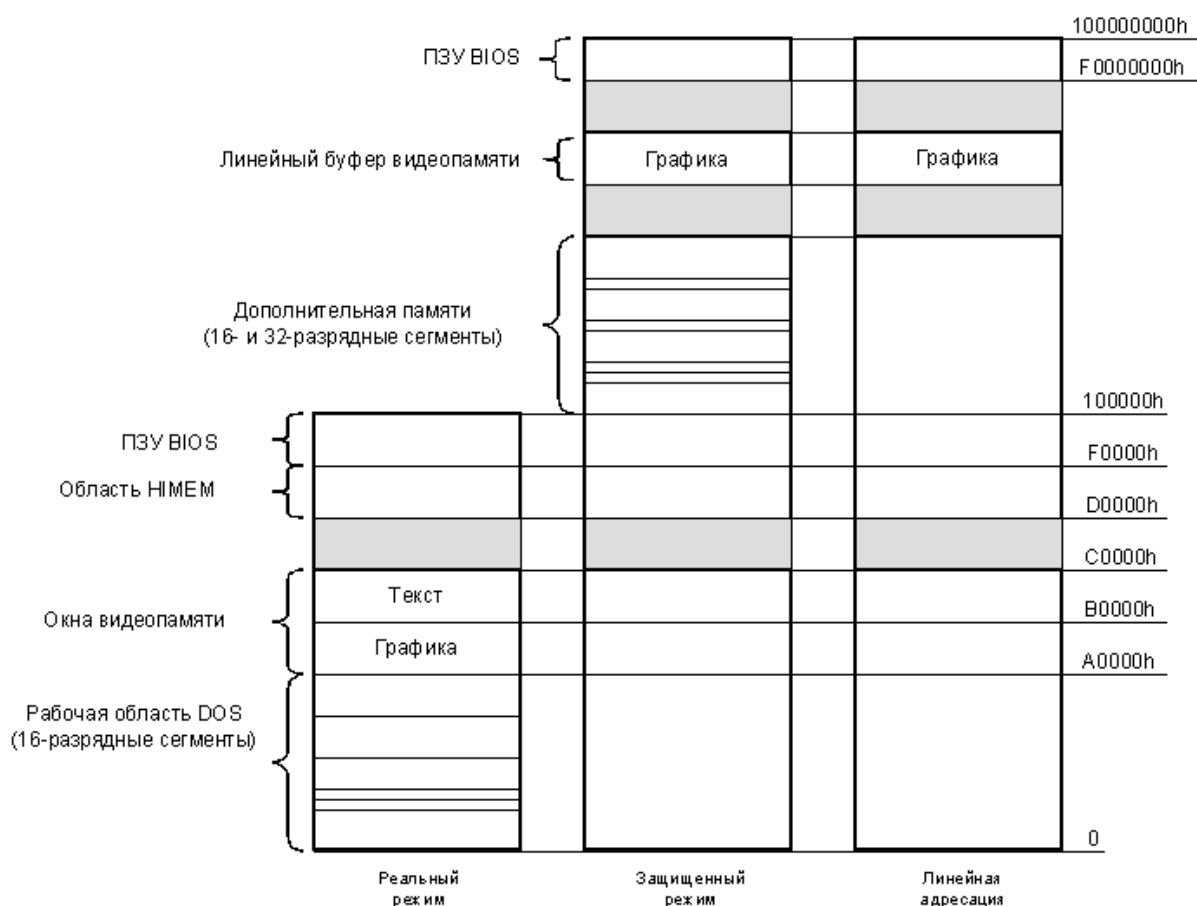


Рисунок 2.1 – Организация адресного пространства в реальном, защищенном и линейном режимах работы процессора x86

Линейную адресацию можно использовать в специализированных программах, активно эксплуатирующих ресурсы ЭВМ — как в компьютерных играх, так и в системах автоматики, измерительных системах, системах управления, связи и т. п. Применение линейной адресации целесообразно в том случае, если проектируемая система предназначена для выполнения ограниченного, заранее известного набора функций и требует высокого быстродействия и надежности. Разработчики процессоров начали внедрять линейную адресацию (в качестве одного из возможных режимов работы) при переходе с 16-разрядной архитектуры на 32-разрядную. Фирма Intel ввела такой режим в процессоре 80386, после чего он стал фактически стандартным (поддерживается не только всеми последующими моделями, но и всеми клонами архитектуры x86), однако остался недокументированным

(почти не описан в литературе и не рассматривается в фирменном руководстве по программированию).

Для пользователей обычных персональных компьютеров линейная адресация в чистом виде интереса не представляет по тем же причинам, что и защищенный режим: DOS и BIOS функционируют только в реальном режиме с 64-килобайт-ными сегментами, и при переходе в любой другой режим программист оказывается один на один с аппаратурой ЭВМ — без документации. Однако кроме чистых режимов процессоры Intel способны работать и в режимах гибридных. Еще в 1989 году Томас Роден (Thomas Roden) предложил использовать интересную комбинацию сегментной (для кода и данных) и линейной (только для данных) адресации [2]. Предложенный им метод позволяет, находясь в обычном режиме DOS, работать со всей доступной памятью в пределах четырехгигабайтного адресного пространства процессора Intel 80386. Чтобы включить режим линейной адресации данных, необходимо снять ограничения на размер сегмента в теновом регистре, соответствующем одному из дополнительных сегментных регистров FS или GS (при необходимости описание архитектуры процессора Pentium можно найти в документации [3-5], размещенной в Интернете на сервере Intel для разработчиков). Через избранный регистр можно обращаться к любой области памяти с помощью прямой адресации или используя в качестве индексного любой 32-разрядный регистр общего назначения. После снятия ограничения запись в выделенный для линейной адресации сегментный регистр выполнять нельзя, иначе нарушится информация в соответствующем ему теновом регистре (предел сегмента сохранится, но начальный адрес будет перезаписан новым значением). Однако стандартные компиляторы и функции DOS с регистрами FS и GS не работают, и соответственно, при вызове процедур эти регистры можно вообще «не трогать» — их не нужно сохранять и восстанавливать. Достаточно один раз снять ограничение на размер адресного пространства, и после выхода из

программы (до перезагрузки компьютера) линейную адресацию можно будет использовать из любой другой программы DOS, как поступил в своем примере Томас Роден.

Рассмотрим более подробно процедуру переключения одного из дополнительных сегментных регистров в режим линейной адресации. Каждый сегментный регистр, как указано в документации [5], состоит из видимой и невидимой (теневой) частей. Информацию в видимую часть можно записывать напрямую при помощи обычных команд пересылки данных (MOV и др.), а для записи в невидимую часть применяются специальные команды, которые доступны только в защищенном режиме. Теневая часть представляет собой так называемый дескриптор (описатель) сегмента, длина которого равна 8 байтам.

При переходе от 16-разрядной архитектуры к 32-разрядной (то есть от i286 к i386) разработчики нового процессора попытались сохранить совместимость снизу вверх по структуре системных регистров, в результате чего дескрипторы сегментов приобрели довольно уродливый (с точки зрения технической эстетики) вид — поля предела и базового адреса разделены на несколько частей. Кроме того, поле предела оказалось ограничено 20 разрядами, что вынудило разработчиков применить еще один радиолобительский трюк — ввести бит гранулярности G, чтобы можно было задавать размер сегмента, превышающий 16 Мбайт.

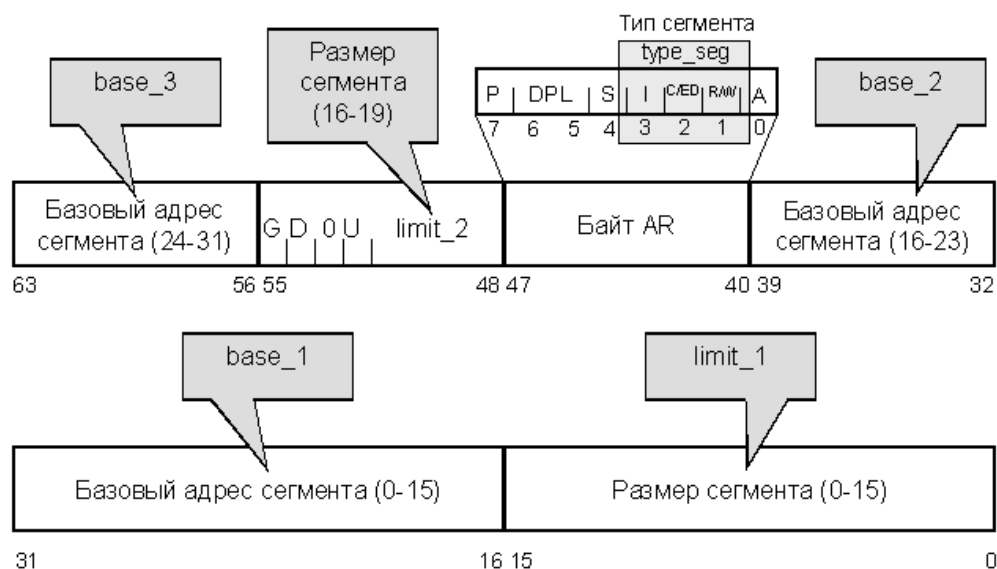


Рисунок 2.2 – Формат дескриптора сегмента

Формат дескриптора сегмента показан на рисунке 2.2. Дескриптор состоит из следующих полей.

- Базовый адрес — 32-разрядное поле, задающее начальный адрес сегмента (в линейном адресном пространстве).
- Предел сегмента — 20-разрядное поле, которое определяет размер сегмента в байтах или 4-килобайтных страницах (в зависимости от значения бита гранулярности G). Поле предела содержит значение, которое должно быть на единицу меньше реального размера сегмента в байтах или страницах.
- Тип — 4-разрядное поле, определяющее тип сегмента и типы операций, которые допустимо с ним выполнять.
- Бит S — признак системного объекта (0 — дескриптор описывает системный объект, 1 — назначение сегмента описывается полем типа).
- DPL — 2-разрядное поле, определяющее уровень привилегий описываемого дескриптором сегмента.

- Бит P — признак присутствия сегмента в оперативной памяти компьютера (0 — сегмент «сброшен» на диск, 1 — сегмент присутствует в оперативной памяти).
- Бит AVL — свободный (available) бит, который может использоваться по усмотрению системного программиста.
- Бит D — признак используемого по умолчанию режима адресации данных (0 — 16-разрядная адресация, 1 — 32-разрядная).
- Бит G — гранулярности сегмента (0 — поле предела задает размер сегмента в байтах, 1 — в 4-килобайтных страницах).

В нашем случае признак используемого по умолчанию режима адресации данных D можно установить в 0 (использовать по умолчанию 16-разрядные операнды), но особой роли его значение не играет — в смешанном режиме сегментно-линейной адресации при работе с линейным сегментом строковые команды, использующие значение этого разряда, применять нельзя. Бит гранулярности G должен быть установлен в 1, чтобы обеспечить охват всего адресного пространства процессора.

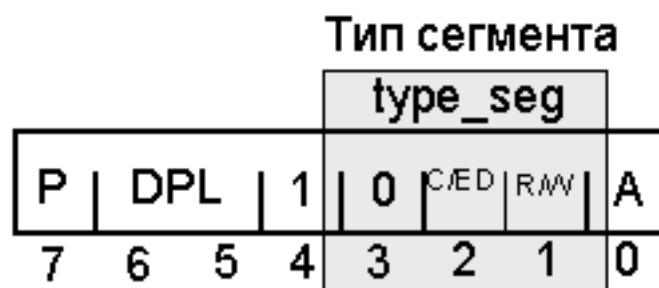
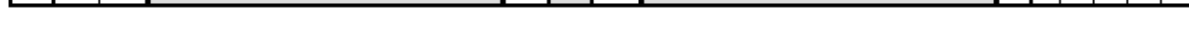


Рисунок 2.3 – Формат прав доступа для сегмента данных

Для сегментов данных формат байта прав доступа (включающего поле типа сегмента) имеет вид, показанный на рисунке 2.3. Как видно из рисунка, поле S для сегментов данных должно быть установлено в 1, а старший разряд поля типа должен иметь значение 0. Поля P и DPL уже упоминались выше. Бит присутствия сегмента P следует установить в 1 (сегмент присутствует в


$$DE(\text{Protection End}) = \frac{Y_{\text{Protection End}} - 1}{Y_{\text{Protection End}} + 1}$$

процессоров 286/287 и 386/387 на процессорах 486 DX и старше бит *MP* должен быть установлен.

- *EM* (Processor Extension Emulated) — эмуляция математического сопроцессора. Установка этого флага вызывает исключение *#NM* при каждой команде, относящейся к сопроцессору, что позволяет прозрачно осуществлять его программную эмуляцию.
- *TS* (Task Switched) — признак переключения задачи (флаг устанавливается в 1 при каждом переключении задач и проверяется перед выполнением команд математического сопроцессора).
- *ET* (Extension Type) — индикатор поддержки набора инструкций математического сопроцессора (0 — выключена, 1 — включена). В процессорах P6 флаг всегда установлен в 1.
- *NE* (Numeric Error) — встроенный механизм контроля ошибок математического сопроцессора (0 — выключен, 1 — включен).
- *WP* (Write Protect) — защита от записи информации в страницы уровня пользователя из процедур супервизора (0 — выключена, 1 — включена).
- *AM* (Alignment Mask) — разрешение контроля выравнивания (контроль выравнивания выполняется только на уровне привилегий 3 при *AM*=1 и флаге *AC*=1. (0 — запрещен, 1 — разрешен).
- *NW* (Not Writethrough) — запрещение сквозной записи и циклов аннулирования. (0 — сквозная запись разрешена, 1 — запрещена).
- *CD* (Cache Disable) — запрет заполнения кэш-памяти (попадание в ранее заполненные строки при этом обслуживаются кэшем). (0 — использование кэш-памяти разрешено, 1 — запрещено).
- *PG* (Paging) — включение страничного преобразования памяти (0 — запрещено, 1 — разрешено).

Набор подпрограмм, необходимых для переключения сегментного регистра GS в режим линейной адресации, показан в листинге 2.1 [1]. Как сказано выше, перезапись содержимого теневого регистра процессора возможна только в защищенном режиме, а переход в этот режим, как видно из листинга, требует ряда дополнительных операций, выполняемых процедурой *Initialization*. В частности, нужно перенастроить на специально выделенные в кодовом сегменте области данных регистры DS, SS и SP. В момент перенастройки регистров стека должны быть запрещены прерывания, поскольку некоторые обработчики прерываний пишут информацию в стек прерываемой программы.

Процедура **SetLAddrModeForGS**, непосредственно осуществляющая перенастройку регистра GS в режим линейной адресации, воспроизводит (с незначительными изменениями) метод Родена. Прежде чем осуществить переключение, нужно вначале подготовить таблицу *GDT* (настроить на текущие сегменты кода и данных) и загрузить ее. Затем нужно войти в защищенный режим — установить в единицу бит *PE* регистра CR0, а остальные разряды сохранить без изменений (в том виде, в котором они находились при работе в реальном режиме). В защищенном режиме необходимо перезагрузить сегментные регистры, сняв при этом ограничения с GS, и сразу же вернуться в реальный режим DOS, сбросив в ноль бит *PE*. Длительное пребывание в защищенном режиме нежелательно, поскольку переключение в него выполнялось по упрощенной схеме: таблица прерываний не создавалась, а сами прерывания были просто отключены.

После выполнения процедуры **SetLAddrModeForGS** обязательно следует отменить замыкание адресного пространства, то есть разблокировать адресную линию *A20*, которая управляется контроллером клавиатуры. Для этого необходимо послать в порт *A* контроллера соответствующую команду. Посылка команды осуществляется при помощи **Enable_A20** и **Wait8042Buffer Empty**.

Листинг 2.1 – Подпрограмма, устанавливающая режим линейной
адресации данных

```
; Порт, управляющий запретом немаскируемых прерываний
CMOS_ADDR      equ  0070h
CMOS_DATA      equ  0071h
; Селекторы сегментов
SYS_PROT_CS    equ  0008h
SYS_REAL_SEG   equ  0010h
SYS_MONDO_SEG  equ  0018h

CODESEG
;*****
;* ВКЛЮЧЕНИЕ РЕЖИМА ЛИНЕЙНОЙ АДРЕСАЦИИ ПАМЯТИ *
;*      (процедура параметров не имеет)      *
;*****
PROC Initialization NEAR
    pushad
; Сохранить значения сегментных регистров в
; реальном режиме (кроме GS)
    mov     [CS:Save_SP],SP
    mov     AX,SS
    mov     [CS:Save_SS],AX
    mov     AX,DS
    mov     [CS:Save_DS],AX
; (работаем теперь только с кодовым сегментом)
    mov     AX,CS
    mov     [word ptr CS:Self_Mod_CS],AX
    mov     DS,AX
    cli
    mov     SS,AX
    mov     SP,offset Local_Stk_Top
    sti

; Установить режим линейной адресации
    call    SetLAddrModeForGS

; Восстановить значения сегментных регистров
    cli
    mov     SP,[CS:Save_SP]
    mov     AX,[CS:Save_SS]
    mov     SS,AX
    mov     AX,[CS:Save_DS]
    mov     DS,AX
    sti

; Разрешить работу линии A20
    call    Enable_A20
    popad
    ret
```

ENDP Initialization

```

; Область сохранения значений сегментных регистров
Save_SP DW ?
Save_SS DW ?
Save_DS DW ?
; Указатель на GDT
GDTPtr DQ ?
; Таблица дескрипторов сегментов для
; входа в защищенный режим
GDT DW 00000h,00000h,00000h,00000h ;не используется
      DW 0FFFFh,00000h,09A00h,00000h ;сегмент кода CS
      DW 0FFFFh,00000h,09200h,00000h ;сегмент данных DS
      DW 0FFFFh,00000h,09200h,0008Fh ;сегмент GS
; Локальный стек для защищенного режима
; (организован внутри кодового сегмента)
label GDTEnd word
      DB 255 DUP(0FFh)
Local_Stk_Top DB (0FFh)

;*****
;*          ОТМЕНИТЬ ПРЕДЕЛ СЕГМЕНТА GS          *
;* Процедура изменяет содержимое теневого        *
;* регистра GS таким образом, что становится     *
;* возможной линейная адресация через него     *
;* 4 Gb памяти в реальном режиме                *
;*****
PROC SetLAddrModeForGS near
; Вычислить линейный адрес кодового сегмента
      mov     AX,CS
      movzx   EAX,AX
      shl     EAX,4 ;умножить номер параграфа на 16
      mov     EBX,EAX ;сохранить линейный адрес в EBX
; Занести младшее слово линейного адреса в дескрипторы
; сегментов кода и данных
      mov     [word ptr CS:GDT+10],AX
      mov     [word ptr CS:GDT+18],AX
      ; Переставить местами старшее и младшее слова
      ror     EAX,16
; Занести биты 16-23 линейного адреса в дескрипторы
; сегментов кода и данных
      mov     [byte ptr CS:GDT+12],AL
      mov     [byte ptr CS:GDT+20],AL
; Установить предел (Limit) и базу (Base) для GDTR
      lea     ax,[GDT] ;*****
      movzx   eax,ax ;*****
      add     EBX,EAX; offset GDT
      mov     [word ptr CS:GDTPtr],(offset GDTEnd-GDT-1)
      mov     [dword ptr CS:GDTPtr+2],EBX

```

```

; Сохранить регистр флагов
    pushf
; Запретить прерывания, так как таблица прерываний IDT
; не сформирована для защищенного режима
    cli
; Запретить немаскируемые прерывания NMI
    in     AL,CMOS_ADDR
    mov    AH,AL
    or     AL,080h      ;установить старший разряд
    out    CMOS_ADDR,AL ;не затрагивая остальные
    and    AH,080h
    ; Запомнить старое состояние маски NMI
    mov    CH,AH
; Перейти в защищенный режим
    lgdt   [fword ptr CS:GDTPtr]
    mov    BX,CS        ;запомнить сегмент кода
    mov    EAX,CR0
    or     AL,01b       ;установить бит PE
    mov    CR0,EAX      ;защита разрешена
    ; Безусловный дальний переход на метку SetPMode
    ; (очистить очередь команд и перезагрузить CS)
    DB     0EAh
    DW     (offset SetPMode)
    DW     SYS_PROT_CS

SetPMode:
    ; Подготовить границы сегментов
    mov    AX,SYS_REAL_SEG
    mov    SS,AX
    mov    DS,AX
    mov    ES,AX
    mov    FS,AX
    ; Снять ограничения с сегмента GS
    mov    AX,SYS_MONDO_SEG
    mov    GS,AX
; Вернуться в реальный режим
    mov    EAX,CR0
    and    AL,11111110b ;сбросить бит PE
    mov    CR0,EAX      ;защита отключена

    ; Безусловный дальний переход на метку SetRMode
    ; (очистить очередь команд и перезагрузить CS)
    DB     0EAh
    DW     (offset SetRMode)
Self_Mod_CS DW ?

SetRMode:
    ; Регистры стека и данных
    ; настроить на сегмент кода
    mov    SS,BX
    mov    DS,BX
    ; Обнулить дополнительные сегментные

```

```

        ; регистры данных (GS не трогать!)
        xor     AX,AX
        mov     ES,AX
        mov     FS,AX
        ; Возврат в реальный режим,
        ; прерывания снова разрешены
        in      AL,CMOS_ADDR
        and     AL,07Fh
        or      AL,CH
        out     CMOS_ADDR,AL
        popf
        ret
ENDP SetLAddrModeForGS

;*****
;* Разрешить работу с памятью выше 1 Мб *
;*****
PROC Enable_A20 near
        call    Wait8042BufferEmpty
        mov     AL,0D1h ;команда управления линией A20
        out     64h,AL
        call    Wait8042BufferEmpty
        mov     AL,0DFh ;разрешить работу линии
        out     60h,AL
        call    Wait8042BufferEmpty
        ret
ENDP Enable_A20

;*****
;* ОЖИДАНИЕ ОЧИСТКИ ВХОДНОГО БУФЕРА I8042 *
;* При выходе из процедуры: *
;* флаг ZF установлен - нормальное завершение, *
;* флаг ZF сброшен - ошибка тайм-аута. *
;*****
proc Wait8042BufferEmpty near
        push    CX
        mov     CX,0FFFFh ;задать число циклов
@@kb:   in      AL,64h      ;получить статус
        test    AL,10b      ;буфер i8042 свободен?
        loopnz  @@kb        ;если нет, то цикл
        pop     CX
        ; (если при выходе сброшен флаг ZF - ошибка)
        ret
endp Wait8042BufferEmpty
ENDS

```

ВНИМАНИЕ! Как уже было сказано, после выхода из защищенного режима нельзя перезаписывать регистр GS, иначе будет полностью или частично стерта информация в соответствующем теневом регистре. В частности, нельзя выполнять операции сохранения/восстановления содержимого регистра при помощи команд работы со стеком *push* и *pop*.

При использовании нестандартных режимов работы возникают определенные трудности в процессе отладки программ: стандартные программы-отладчики становятся неудобными. Во многих случаях, однако, достаточно использовать простую отладочную печать. В листинге 2.2 [1] приведена подпрограмма ShowRegs, отображающая на экране содержимое регистров общего назначения, сегментных регистров, регистра флагов и регистра CR0. Недостаток этого упрощенного примера заключается в том, что ShowRegs *не сохраняет содержимое видеопамати*. Однако при использовании линейной адресации программу не трудно усовершенствовать, если есть достаточный запас оперативной памяти: в текстовом режиме для сохранения одной страницы нужно менее 4 Кбайт, а в графическом режиме TrueColor32 с разрешением 1920x1280 требуется уже 9,5 Мбайт.

Листинг 2.2 – Отладочная подпрограмма, предназначенная для отображения на экран содержимого регистров процессора

```
DATASEG
label REGROW_386 byte
    DB 0,0,'EAX =',0
    DB 1,0,'EBX =',0
    DB 2,0,'ECX =',0
    DB 3,0,'EDX =',0
    DB 4,0,'ESI =',0
    DB 5,0,'EDI =',0
    DB 6,0,'EBP =',0
    DB 7,0,'ESP =',0
    DB 8,0,'IP  =',0
    DB 9,0,'CS  =',0
    DB 10,0,'DS  =',0
```

```

DB 11,0,'ES  =',0
DB 12,0,'FS  =',0
DB 13,0,'GS  =',0
DB 14,0,'SS  =',0
DB 16,8,'          AVR  NIOODIT SZ A P C',0
DB 17,8,'          CMF  TPLFFFF FF F F F',0
DB 18,0,'Флаги:',0
DB 20,8,'PCN          A V          NETEMP',0
DB 21,8,'GDW          M P          ETSMP',0
DB 22,0,'CR0:',0
DB 24,15
DB 'Для продолжения работы нажмите любую клавишу',0

```

CODESEG

```

;*****
;* ВЫВЕСТИ НА ЭКРАН ДАМП РЕГИСТРОВ ПРОЦЕССОРА *
;*      (процедура параметров не имеет)      *
;*****
PROC ShowRegs FAR
    pushad
    pushfd
    push    DS
    mov     BP,SP
    mov     AX,DGROUP
    mov     DS,AX
; Сохраняем глобальные переменные
    mov     AL,[TextColorAndBackground]
    push    AX
    push    [ScreenString]
    push    [ScreenColumn]
; Очищаем экран
    call    ClearScreen
; Вывести 21 строку текста
    mov     [TextColorAndBackground],YELLOW
    mov     SI, offset REGROW_386
    mov     CX,22
@@GLB:    call ShowString
    loop    @@GLB

    mov     [TextColorAndBackground],WHITE

    mov     EAX,[BP+34] ;Показать EAX
    mov     [ScreenString],0
    mov     [ScreenColumn],6
    call    ShowHexDWord
    mov     EAX,[BP+22] ;Показать EBX
    inc     [ScreenString]
    mov     [ScreenColumn],6
    call    ShowHexDWord
    mov     EAX,[BP+30] ;Показать ECX

```



```

inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexDWord
mov      EAX,[BP+26] ;Показать EDX
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexDWord
mov      EAX,[BP+10] ;Показать ESI
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexDWord
mov      EAX,[BP+6]  ;Показать EDI
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexDWord
mov      EAX,[BP+14] ;Показать EBP
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexDWord
mov      EAX,[BP+18] ;Показать ESP
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexDWord
mov      AX,[BP+38]  ;Показать IP
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexWord
mov      AX,[BP+40]  ;Показать CS
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexWord
mov      AX,[BP]      ;Показать DS
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexWord
mov      AX,ES        ;Показать ES
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexWord
mov      AX,FS        ;Показать FS
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexWord
mov      AX,GS        ;Показать GS
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexWord
mov      AX,SS        ;Показать SS
inc      [ScreenString]
mov      [ScreenColumn],6
call     ShowHexWord

```

```

        add     [ScreenString],4
        mov     [ScreenColumn],8
        mov     EAX,[BP+2]
        call    ShowBinDWord
        add     [ScreenString],4
        mov     [ScreenColumn],8
        mov     EAX,CRO
        call    ShowBinDWord
; Ожидаем нажатия любого символа на клавиатуре
        call    GetChar
; Очищаем экран
        call    ClearScreen

; Восстановить глобальные переменные
        pop     [ScreenColumn]
        pop     [ScreenString]
        pop     AX
        mov     [TextColorAndBackground],AL
        pop     DS
        popfd
        popad
        ret
ENDP ShowRegs
ENDS

```

В программе LAddrTest, показанной в листинге 2.3 [1], используются процедуры из листингов 2.1 и 2.2 для включения режима линейной адресации и демонстрации изменения содержимого сегментных регистров, которое при этом происходит (процедура установки линейного режима перезаписывает теневой регистр у регистра GS, а регистры ES и FS просто обнуляет). После выполнения программы режим линейной адресации данных *сохраняется*, и любая другая программа, в том числе написанная на языке высокого уровня, может через GS обращаться к любой области памяти по физическому адресу.

Листинг 2.3 – Включение режима линейной адресации

```

IDEAL
P386
LOCALS
MODEL MEDIUM

; Подключить файл мнемонических обозначений
; кодов управляющих клавиш

```

```

include "lst_2_03.inc"
; Подключить файл мнемонических обозначений цветов
include "lst_2_05.inc"

SEGMENT sseg para stack 'STACK'
DB 400h DUP(?)
ENDS

DATASEG
; Текстовые сообщения
Text1 DB 0,19,"Включение режима "
      DB "линейной адресации данных",0
      DB 11,0,"Для просмотра "
      DB "содержимого регистров процессора",0
      DB 12,0,"перед запуском процедуры "
      DB "перехода в режим",0
      DB 13,0,"линейной адресации нажмите "
      DB "любую клавишу.",0
Text2 DB 11,0,"Произведено переключение в "
      DB "режим линейной адресации.",0
      DB 12,0,"Для просмотра содержимого "
      DB "регистров процессора",0
      DB 13,0,"нажмите любую клавишу.",0
Text3 DB 11,0,"После завершения данной "
      DB "программы регистр GS",0
      DB 12,0,"может использовать для "
      DB "линейной адресации",0
      DB 13,0,"любая другая программа.",0
      DB 24,18,"Для выхода из программы "
      DB "нажмите любую клавишу.",0
ENDS

CODESEG
;*****
;* Основной модуль программы *
;*****
PROC LAddrTest
      mov     AX,DGROUP
      mov     DS,AX
; Установить текстовый режим и очистить экран
      mov     AX,3
      int     10h
; Скрыть курсор - убрать за нижнюю границу экрана
      mov     [ScreenString],25
      mov     [ScreenColumn],0
      call    SetCursorPosition
; Вывести первое текстовое сообщение
; на экран зеленым цветом
      mov     [TextColorAndBackground],LIGHTGREEN
      mov     CX,4

```

```

        mov     SI,offset Text1
@@NextString1:
        call    ShowString
        loop    @@NextString1
        ; Ожидать нажатия любой клавиши
        call    GetChar
; Занести контрольное число в дополнительные
; сегментные регистры данных
        mov     AX,0ABCDh
        mov     ES,AX
        mov     FS,AX
        mov     GS,AX
; Показать содержимое регистров процессора
        call    far ShowRegs

; Установить режим прямой адресации памяти
call    Initialization

; Вывести второе текстовое сообщение
; на экран голубым цветом
        mov     [TextColorAndBackground],LIGHTCYAN
        mov     CX,3
        mov     SI,offset Text2
@@NextString2:
        call    ShowString
        loop    @@NextString2
        ; Ожидать нажатия любой клавиши
        call    GetChar
; Показать содержимое регистров процессора
        call    far ShowRegs

; Вывести третье текстовое сообщение
; на экран желтым цветом
        mov     [TextColorAndBackground],YELLOW
        mov     CX,4
        mov     SI,offset Text3
@@NextString3:
        call    ShowString
        loop    @@NextString3
        ; Ожидать нажатия любой клавиши
        call    GetChar

; Установить текстовый режим
        mov     ax,3
        int     10h
; Выход в DOS
        mov     AH,4Ch
        int     21h
ENDP LAddrTest
ENDS

```

```

; Подключить набор процедур вывода/вывода данных
include "lst_2_02.inc"
; Подключить подпрограмму, переводящую сегментный
; регистр GS в режим линейной адресации
include "lst_3_01.inc"
; Подключить подпрограмму, отображающую на экране
; содержимое регистров процессора
include "lst_3_02.inc"

END

```

Листинг 2.4 [1] демонстрирует использование линейной адресации для отображения содержимого памяти компьютера на экране, то есть выдачи дампа памяти. Программа *MemoryDump* позволяет просматривать все адресное пространство, а не только оперативную память. Можно, например, считывать память видеоконтроллера или вообще неиспользуемые области.

Кроме процедур ввода/вывода общего назначения, в *MemoryDump* используются также следующие подпрограммы:

- процедура *ShowASCIIChar* осуществляет вывод символа в ASCII-коде в заданную позицию экрана;
- процедура *HexToBin32* осуществляет перевод числа (введенного с клавиатуры адреса) из шестнадцатеричного кода в двоичный;
- процедура *GetAddressOrCommand* принимает команды, вводимые с клавиатуры (введенное число воспринимается как линейный адрес памяти в шестнадцатеричном коде, нажатие на управляющие клавиши — как команда).

Листинг 2.4 – Использование линейной адресации для отображения на экран содержимого оперативной памяти

```

IDEAL
P386
LOCALS
MODEL MEDIUM

; Подключить файл мнемонических обозначений
; кодов управляющих клавиш
include "lst_2_03.inc"

```

```

; Подключить файл мнемонических обозначений цветов
include "lst_2_05.inc"

SEGMENT sseg para stack 'STACK'
DB 400h DUP(?)
ENDS

DATASEG
; Текстовые сообщения
Txt1 DB LIGHTMAGENTA,0,28,"Дамп оперативной памяти",0
      DB YELLOW,2,0,"Адрес:",0
      DB LIGHTGREEN,2,11
      DB "Шестнадцатеричное представление:",0
      DB LIGHTCYAN,2,61,"ASCII-коды:",0
      DB LIGHTRED,21,0,"Введите число "
      DB "или нажмите управляющую клавишу:",0
Txt2 DB 23,0, "Стрелка вниз - следующие 256 байт;",0
      DB 23,35, "Стрелка вверх - предыдущие 256 байт;",0
      DB 24,0, "Enter - завершение ввода адреса;",0
      DB 24,33, "Esc - отмена ввода адреса;",0
      DB 24,60, "F10 - выход.",0
; Количество введенных символов числа
CharacterCounter DB 0
; Позиция для ввода адреса на экране
OutAddress DB 21,47
; Строка для ввода адреса
AddressString DB 9 DUP(0)
; Строка пробелов для "затирания" числа
SpaceString DB 21,47,9 DUP(' '),0
; Начальный адрес
StartAddress DD 0
; Код команды
CommandByte DB 0
ENDS

CODESEG
;*****
;* Основной модуль программы *
;*****
PROC MemoryDump
      mov     AX,DGROUP
      mov     DS,AX
; Устанавливаем режим прямой адресации памяти
      call    Initialization
; Установить текстовый режим и очистить экран
      mov     AX,3
      int     10h
; Скрыть курсор - убрать за нижнюю границу экрана
      mov     [ScreenString],25
      mov     [ScreenColumn],0

```

```

        call    SetCursorPosition
; Вывести текстовые сообщения на экран
        mov     CX,5
        mov     SI,offset Txt1
@@NextString1:
        call    ShowColorString
        loop    @@NextString1
        mov     [TextColorAndBackground],WHITE
        mov     CX,5
        mov     SI,offset Txt2
@@NextString2:
        call    ShowString
        loop    @@NextString2

; Установить белый цвет символов и черный фон
        mov     [TextColorAndBackground],WHITE
; Отобразить символы-разделители колонок
        mov     AL,0B3h
        mov     [ScreenString],2
        mov     [ScreenColumn],9
        call    ShowASCIICChar
        mov     [ScreenColumn],59
        call    ShowASCIICChar
        mov     [ScreenString],3
        mov     [ScreenColumn],9
        call    ShowASCIICChar
        mov     [ScreenColumn],59
        call    ShowASCIICChar

; Инициализируем переменные
        mov     [StartAddress],0
        mov     [CommandByte],0
; ВНЕШНИЙ ЦИКЛ
@@q0:    mov     EBX,[StartAddress]
        mov     [ScreenString],4
        mov     DX,16
@@q1:    mov     [ScreenColumn],0
; Отобразить линейный адрес первого байта в группе
        mov     [TextColorAndBackground],YELLOW
        mov     EAX,EBX
        call    ShowHexDWord
; Отобразить символ-разделитель колонок
        mov     [TextColorAndBackground],WHITE
        inc     [ScreenColumn]
        mov     AL,0B3h
        call    ShowASCIICChar
        inc     [ScreenColumn]
; Отобразить очередную группу байт
; в шестнадцатеричном коде
        mov     CX,16
        mov     [TextColorAndBackground],LIGHTGREEN

```

```

@@q2:    mov     AL,[GS:EBX]
         call    ShowByteHexCode
         inc     [ScreenColumn]
         inc     EBX
         loop    @@q2
; Отобразить символ-разделитель колонок
         mov     [TextColorAndBackground],WHITE
         mov     AL,0B3h
         call    ShowASCIIChar
         inc     [ScreenColumn]
         ; Вернуться назад на 16 символов
         sub     EBX,16
; Отобразить очередную группу байт в кодах ASCII
         mov     CX,16
         mov     [TextColorAndBackground],LIGHTCYAN
@@q3:    mov     AL,[GS:EBX]
         call    ShowASCIIChar
         inc     EBX
         loop    @@q3
         inc     [ScreenString]
         dec     DX
         jnz     @@q1

         ; Ожидать нажатия любой клавиши
         call    GetAddressOrCommand
         cmp     [CommandByte],F10
         jne     @@q0

@@End:   ; Установить текстовый режим
mov     ax,3
int     10h
; Выход в DOS
mov     AH,4Ch
int     21h
ENDP MemoryDump

```

```

;*****
;*          ВЫВОД БАЙТА НА ЭКРАН В КОДЕ ASCII          *
;* Подпрограмма выводит содержимое регистра AL в коде *
;* ASCII в указанную позицию экрана.                  *
;* Координаты позиции передаются через глобальные    *
;* переменные ScreenString и ScreenColumn. После      *
;* выполнения операции вывода происходит автомати-    *
;* ческое приращение значений этих переменных.        *
;*****
PROC ShowASCIIChar near
    pusha
    push    DS
    push    ES
    mov     DI,DGROUP

```



```

        mov     DS,DI
        cld
; Настроить пару ES:DI для прямого вывода в видеопамять
        push    AX
        ; Загрузить адрес сегмента видеоданных в ES
        mov     AX,0B800h
        mov     ES,AX
        ; Умножить номер строки на длину строки в байтах
        mov     AX,[ScreenString]
        mov     DX,160
        mul     DX
        ; Прибавить номер колонки (дважды)
        add     AX,[ScreenColumn]
        add     AX,[ScreenColumn]
        ; Переписать результат в индексный регистр
        mov     DI,AX
        pop     AX
        mov     AH,[TextColorAndBackground]
        stosw

; Подготовка для вывода следующих байтов
        ; Перевести текущую позицию на 2 символа влево
        inc     [ScreenColumn]
        ; Проверить пересечение правой границы экрана
        cmp     [ScreenColumn],80
        jb      @@End
        ; Если достигнута правая граница экрана -
        ; перейти на следующую строку
        sub     [ScreenColumn],80
        inc     [ScreenString]
@@End:   pop     ES
        pop     DS
        popa
        ret
ENDP ShowASCIIChar

```

```

;*****
;* ПЕРЕВОД ЧИСЛА ИЗ ШЕСТНАДЦАТЕРИЧНОГО КОДА В ДВОИЧНЫЙ *
;* DS:SI - число в коде ASCII.                               *
;* Результат возвращается в EAX.                             *
;*****
PROC HexToBin32 near
        push    EBX
        push    CX
        push    SI
        cld
        xor     EBX,EBX ;обнуляем накопитель
        xor     CX,CX   ;обнуляем счетчик цифр
@@h0:   lodsb
        ; Проверка на ноль (признак конца строки)

```

```

        and     AL,AL
        jz      @@h4
        ; Проверка на диапазон '0'-'9'
        cmp     AL,'0'
        jnb     @@Error
        cmp     AL,'9'
        ja      @@h1
        sub     AL,'0'
        jmp short @@h3
@@h1:    ; Проверка на диапазон 'A'-'F'
        cmp     AL,'A'
        jnb     @@Error
        cmp     AL,'F'
        ja      @@h2
        sub     AL,'A'-10
        jmp short @@h3
@@h2:    ; Проверка на диапазон 'a'-'f'
        cmp     AL,'a'
        jnb     @@Error
        cmp     AL,'f'
        ja      @@Error
        sub     AL,'a'-10
@@h3:    ; Дописать к результату
        ; очередные 4 разряда справа
        shl     EBX,4
        or      BL,AL
        inc     CX
        cmp     CX,8
        jbe     @@h0
        ; Если в числе больше 8 цифр - ошибка
        jmp short @@Error
@@h4:    ; Успешное завершение - результат в EAX
        mov     EAX,EBX
        jmp short @@End
@@Error: ; Ошибка - обнулить результат
        xor     EAX,EAX
@@End:   pop     SI
        pop     CX
        pop     EBX
        ret
ENDP HexToBin32

```

```

;*****
;* ПРИНЯТЬ С КЛАВИАТУРЫ НОВЫЙ АДРЕС ИЛИ КОМАНДУ *
;*****
PROC GetAddressOrCommand near
    pushad
    ; Использовать при выводе белый цвет, черный фон
    mov     [TextColorAndBackground],WHITE
    ; Установить номер строки поля ввода

```

```

        mov     [ScreenString],21
@@GetAddressOrCommand:
; Инициализировать переменные
        ; Обнулить счетчик цифр
        mov     [CharacterCounter],0
        ; Очистить строку
        mov     DI,offset AddressString
        mov     [byte ptr DS:DI],0
        ; Очистить позицию ввода (забить пробелами)
        mov     SI,offset SpaceString
        call    ShowString
        ; Установить курсор в позицию ввода
        mov     [ScreenColumn],47
        mov     AL,[CharacterCounter]
        add     [byte ptr ScreenColumn],AL
        call    SetCursorPosition
        ; Ввести цифру или команду
        call    GetChar
        ; Адрес или команда?
        cmp     AL,0
        jz      @@Command
        ; Введена первая цифра числа

; ВВОД АДРЕСА В ШЕСТНАДЦАТЕРИЧНОМ КОДЕ
@@Address:
        ; Проверка на диапазон '0'-'9'
        cmp     AL,'0'
        jb      @@AddressError
        cmp     AL,'9'
        jbe     @@WriteChar
        ; Проверка на диапазон 'A'-'F'
        cmp     AL,'A'
        jb      @@AddressError
        cmp     AL,'F'
        jbe     @@WriteChar
        ; Проверка на диапазон 'a'-'f'
        cmp     AL,'a'
        jb      @@AddressError
        cmp     AL,'f'
        ja      @@AddressError
@@WriteChar:
        ; Проверяем количество цифр
        cmp     [CharacterCounter],8
        jae     @@AddressError
        inc     [CharacterCounter]
        ; Записываем цифру в число
        mov     [DS:DI],AL
        inc     DI
        ; Передвинуть признак конца строки
        ; в следующий разряд
        mov     [byte ptr DS:DI],0

```

```

        ; Отобразить число на экране
mov     SI,offset SpaceString
call    ShowString
mov     SI,offset OutAddress
call    ShowString
@@GetNextChar:
        ; Отобразить курсор в новой позиции ввода
mov     [ScreenColumn],47
mov     AL,[CharacterCounter]
add     [byte ptr ScreenColumn],AL
call    SetCursorPosition
        ; Ожидать ввода следующего символа
call    GetChar
cmp     AL,0
jne     @@Address

; Проанализировать код нажатой клавиши
cmp     AH,B_Esc          ;отмена ввода адреса
je      @@GetAddressOrCommand

@@TestF10:
cmp     AH,F10            ;"Выход"
jne     @@TestRubout
mov     [CommandByte],AH
jmp     @@End

@@TestRubout:
cmp     AH,B_RUBOUT      ;"Забой"
jne     @@TestEnter
cmp     [CharacterCounter],0
je      @@AddressError
        ; Передвинуть признак конца строки
        ; на разряд влево
dec     DI
dec     [CharacterCounter]
mov     [byte ptr DS:DI],0
        ; Отобразить число на экране
mov     SI,offset SpaceString
call    ShowString
mov     SI,offset OutAddress
call    ShowString
jmp     @@GetNextChar

@@TestEnter:
cmp     AH,B_Enter       ;завершение ввода числа
jne     @@AddressError
mov     [CommandByte],AH
mov     SI,offset AddressString
call    HexToBin32
mov     [StartAddress],EAX
jmp     short @@End

```

```

@@AddressError:
    call    Beep
    jmp     @@GetNextChar

; ОБРАБОТКА "КОМАНД"
@@Command:
    cmp     AH,F10            ;"Выход"
    jne     @@TestDn
    mov     [CommandByte],AH
    jmp     short @@End

@@TestDn:
    cmp     AH,B_DN          ;"Стрелка вниз"
    jne     @@TestUp
    mov     [CommandByte],AH
    add     [StartAddress],256
    jmp     short @@End

@@TestUp:
    cmp     AH,B_UP          ;"Стрелка вверх"
    jne     @@CommandError
    mov     [CommandByte],AH
    sub     [StartAddress],256
    jmp     short @@End

@@CommandError:
    call    Beep
    jmp     @@GetAddressOrCommand
@@End:    popad
         ret
ENDP GetAddressOrCommand
ENDS

; Подключить подпрограмму, переводящую сегментный
; регистр GS в режим линейной адресации
include "lst_3_01.inc"
; Подключить набор процедур вывода/вывода данных
include "lst_2_02.inc"

END

```

Метод Родена проверен не только на процессорах Intel, но и на клонах, изготовленных AMD, Cyrix, IBM, TI [1]. На всех протестированных компьютерах переход в режим линейной адресации данных проходил нормально, то есть метод не только работоспособен, но и универсален. Метод Родена в свое время не был оценен по достоинству, поскольку

обычный объем памяти персональных компьютеров составлял тогда 1-2 Мбайт, и преимущества линейной адресации не были очевидными. Резкое увеличение объема памяти в устройствах массового применения произошло гораздо позже — начиная с 1995 года. В это же время был внедрен новый стандарт на видеоконтроллеры (VESA 2.0) и появилась возможность линейной адресации видеопамати, однако о методе Родена программисты уже успели забыть. Между тем, совместное использование линейной адресации данных в оперативной памяти и линейного пространства видеопамати дает наибольший выигрыш по скорости выполнения программ и позволяет сильно упростить алгоритмы построения изображений.

Таким образом, метод Томаса Родена обладает следующими основными преимуществами [1]:

- имеется свободный доступ ко всем аппаратным ресурсам компьютера;
- возможна линейная адресация всей оперативной памяти и памяти видеоконтроллера;
- логические и физические адреса отображенной на шину процессора памяти периферийных устройств совпадают;
- метод совместим с клонами процессоров Intel;
- сохраняется возможность использования всех функций DOS и BIOS, как в обычном реальном режиме работы процессора.

Последнее свойство особенно важно: не нужно разрабатывать собственные программы для работы с периферийными устройствами на уровне регистров, следовательно, не проявляются и не создают лишних проблем нестандартные особенности оборудования.

Основной недостаток метода Родена — существенное ослабление защиты памяти. Поскольку отменен контроль границы сегмента данных, работающая с линейным пространством подпрограмма в случае ошибки адресации или заикливания может не только разрушить смежные данные, но

и вообще стереть все содержимое оперативной памяти, в том числе все программы и резидентную часть операционной системы. Чаще всего стирается таблица векторов прерываний, размещенная в начале адресного пространства. Следовательно, необходимо ограничивать число подпрограмм, работающих с линейной адресацией, и очень тщательно их отлаживать.

Второй недостаток прямо вытекает из первого — работа в реальном режиме DOS и ослабление защиты не позволяют реализовать многозадачность. Однако для решения прикладных задач часто вполне достаточно фоново-оперативного режима работы, когда всеми ресурсами системы распоряжается один программный модуль, а остальные предназначены для узкоспециальных целей и вызываются на короткие промежутки времени через механизм прерываний. Иными словами, доступ к видеопамяти и всей оперативной памяти должен быть только у основной программы, а вспомогательные процедуры и драйверы периферийных устройств могут хранить свои данные только в основной области памяти DOS (то есть, в пределах первого мегабайта адресного пространства). Линейная адресация, сама по себе, не накладывает слишком жестких ограничений на работу системы, поскольку персональные компьютеры вообще функционируют в основном в однозадачном режиме: аппаратные средства для реализации многозадачности имеются уже давно, но сильные ограничения создают физиологические и психологические особенности человека, который сидит за компьютером. Любая серьезная работа требует от оператора полной концентрации внимания на одном процессе. То же самое относится к компьютерным играм — невозможно одновременно играть в Quake и редактировать текст.

Третий недостаток: строковые команды процессора x86 в реальном режиме не пригодны для работы с сегментом, настроенным на линейную адресацию памяти. Это не очень существенный недостаток, поскольку внутренняя RISC-архитектура современных процессоров позволяет выполнять

группу из нескольких простых команд с той же скоростью, что и одну сложную составную команду, выполняющую аналогичную операцию. Кроме того, процессор выполняет внутренние операции быстрее, чем операции обращения к оперативной памяти, и гораздо быстрее, чем операции чтения/записи в видеопамять.

В целом можно сказать, что предложенный Роденом режим — это в первую очередь режим учебно-отладочный. Его очень удобно применять в процессе освоения методов непосредственной работы с периферийными устройствами. Во-первых, линейная адресация абсолютно прозрачна — область памяти устройства можно просматривать прямо по физическому адресу. Во-вторых, исследуемое устройство можно рассматривать изолированно, исключив опасность возникновения паразитных взаимодействий с другими аппаратными компонентами и посторонним программным обеспечением.

Ниже приведены файлы, включаемые в программу, приведенную в листинге 2.4 [1].

Листинг 2.3 – Мнемонические обозначения кодов управляющих клавиш

```
; Для клавиш, традиционно выполняющих определенные  
; функции, приведены краткие комментарии справа.  
  
; Для "текстовых" управляющих клавиш вместо скан-кодов  
; используются ASCII-коды:  
B_RUBOUT equ 8 ;забой  
B_TAB equ 9 ;табуляция  
B_LF equ 10 ;перевод строки  
B_ENTER equ 13 ;возврат каретки  
B_ESC equ 27 ;"Esc"  
  
; Скан-коды функциональных клавиш:  
F1 equ 59 ;вызов подсказки на экран  
F2 equ 60  
F3 equ 61  
F4 equ 62  
F5 equ 63  
F6 equ 64  
F7 equ 65  
F8 equ 66
```



```

F9          equ    67
F10         equ    68 ;выход из программы

; Скан-коды клавиш дополнительной клавиатуры:
B_HOME      equ    71 ;перейти в начало
B_UP        equ    72 ;стрелка вверх
B_PGUP      equ    73 ;на страницу вверх
B_BS        equ    75 ;стрелка влево
B_FWD       equ    77 ;стрелка вправо
B_END       equ    79 ;перейти в конец
B_DN        equ    80 ;стрелка вниз
B_PGDN      equ    81 ;на страницу вниз
B_INS       equ    82 ;переключить режим (вставка/замещение)
B_DEL       equ    83 ;удалить символ над курсором

; Скан-коды часто используемых комбинаций клавиш:
ALT_F1      equ    104
ALT_F2      equ    105
CTRL_C      equ     3
CTRL_BS     equ    115
CTRL_FWD    equ    116
CTRL_END    equ    117
CTRL_PGDN   equ    118
CTRL_HOME   equ    119
CTRL_PGUP   equ    122

```

Листинг 2.5 – Мнемонические обозначения цветов для цветного текстового видеорежима

```

; "Темные" цвета (можно использовать для фона и текста)
BLACK       equ    0 ;черный
BLUE        equ    1 ;темно-синий
GREEN       equ    2 ;темно-зеленый
CYAN        equ    3 ;бирюзовый (циан)
RED         equ    4 ;темно-красный
MAGENTA     equ    5 ;темно-фиолетовый
BROWN       equ    6 ;коричневый
LIGHTGREY   equ    7 ;серый
; "Светлые" цвета (только для текста)
DARKGREY    equ    8 ;темно-серый
LIGHTBLUE   equ    9 ;синий
LIGHTGREEN  equ   10 ;зеленый
LIGHTCYAN   equ   11 ;голубой
LIGHTRED    equ   12 ;красный
LIGHTMAGENTA equ  13 ;фиолетовый
YELLOW      equ   14 ;желтый
WHITE       equ   15 ;белый

```

2.2 Вопросы для самопроверки

1. В каком режиме работает микропроцессор x86 сразу после сброса?
2. Как осуществляется сегментная адресация памяти в защищенном режиме?
3. Что такое привилегированные команды процессора?
4. Какие действия надо предпринять, чтобы в программе выполнялись привилегированные команды?
5. Какова структура дескриптора сегмента?
6. Для чего используются сегментные регистры в защищенном режиме?
7. Что такое селектор дескриптора?
8. Что такое линейный адрес?
9. Что такое базовый адрес сегмента?
10. Что такое лимит сегмента?
11. Для чего нужны атрибуты сегмента?
12. Как выполняется перевод процессора в защищенный режим из реального?
13. Что такое псевдодескриптор?
14. Какая команда используется для загрузки таблицы глобальных дескрипторов?
15. Что такое теневые регистры дескрипторов?
16. Почему перед переводом процессора в защищенный режим надо запретить все прерывания?
17. Почему нельзя корректно завершить программу, находясь в защищенном режиме?
18. Когда загружаются теневые регистры дескрипторов процессора?
19. Каким образом обнуляется стек предвыбранных команд при переходе в защищенный режим?
20. Зачем обнуляется стек предвыбранных команд при переходе в защищенный режим?

21. Как производится возврат из защищенного режима работы процессора в реальный?
22. Что такое линейный режим адресации памяти в реальном режиме?
23. С каким объемом памяти позволяет работать режим линейной адресации в реальном режиме?
24. Чем различается организация памяти в реальном, защищенном и линейном режимах адресации?
25. Почему при работе в линейном режиме адресации необходимо включить адресный сигнал A_{20} ?
26. Можно ли в режиме линейной адресации работать с памятью видеоконтроллера?
27. Можно ли в режиме линейной адресации выполнять функции DOS?
28. Работают ли механизмы защиты памяти при использовании метода линейной адресации в реальном режиме?
29. Можно ли использовать строковые команды с сегментом, настроенным на линейную адресацию памяти?

3 Определение параметров системы из программы пользователя

При запуске программы пользователя часто бывает необходимо выяснить, насколько параметры системы, на которой запущена программа, соответствуют требованиям программы.

Ниже приведен пример определения таких параметров системы для программы пользователя, которая должна выполняться под управлением операционной системы Windows 98 SR2, иметь не менее 512 мегабайт оперативной памяти, иметь мышь

3.1 Определение версии операционной системы

Для определения версии запущенной операционной системы может быть использована функция API Win32 GetVersionEx.

```
BOOL GetVersionEx(  
    LPOSVERSIONINFO lpVersionInformation // указатель на  
); // структуру
```

Параметры

lpVersionInformation

Указатель на структуру данных OSVERSIONINFO, которую функция заполняет информацией о версии операционной системы.

Перед вызовом функции GetVersionEx необходимо в поле dwOSVersionInfoSize структуры OSVERSIONINFO занести значение sizeof(OSVERSIONINFO).

Возвращаемые значения

При успешном выполнении функции возвращается ненулевое значение.

При возникновении ошибки возвращается нуль. Для получения расширенной информации об ошибке необходимо вызвать функцию

GetLastError. Ошибка возникает, если неверно указано поле dwOSVersionInfoSize структуры OSVERSIONINFO.

Структура данных OSVERSIONINFO содержит сведения о версии операционной системы. Информация включает в себя старшую и младшую части версии, номер сборки, идентификатор платформы и строку описания операционной системы.

```
typedef struct _OSVERSIONINFO{
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[ 128 ];
} OSVERSIONINFO;
```

Поля структуры

dwOSVersionInfoSize

Определяет размер структуры в байтах. Перед вызовом функции GetVersionEx следует заполнить это поле значением sizeof(OSVERSIONINFO).

dwMajorVersion

Идентифицирует старшую часть версии операционной системы. Например, для Windows NT версии 3.51, старшая часть версии – 3, а для Windows NT версии 4.0 – 4.

dwMinorVersion

Идентифицирует младшую версии операционной системы. Например, для Windows NT версии 3.51, младшая часть версии – 51, а для Windows NT версии 4.0 – 0.

dwBuildNumber

Windows NT: Идентифицирует номер сборки операционной системы.

Windows 95: В младшем слове идентифицирует номер сборки операционной системы. Старшее слово содержит старшую и младшую части версии.

dwPlatformId

Идентифицирует платформу операционной системы. Это поле может содержать одно из следующих значений:

VER_PLATFORM_WIN32s	Win32s on Windows 3.1.
VER_PLATFORM_WIN32_WINDOWS	Win32 on Windows 95.
VER_PLATFORM_WIN32_NT	Win32 on Windows NT.

szCSDVersion

Windows NT: Содержит строку, завершающуюся нулем, такую, как "Service Pack 3", которая индицирует последний сервис-пак, установленный в систему. Если нет установленных сервис-паков, строка пустая.

Windows 95: Содержит строку, завершающуюся нулем, которая дает необязательную дополнительную информацию об операционной системе.

```
// Проверка версии операционной системы
OSVERSIONINFO ver;
DWORD c;
char* d;
ver.dwOSVersionInfoSize=sizeof(OSVERSIONINFO);
GetVersionEx(&ver);
if((ver.dwPlatformId!=
    VER_PLATFORM_WIN32_WINDOWS)|| // He Windows 95/98
    ver.dwMajorVersion!=4)||      // He Windows 98
```

```

        (ver.dwMinorVersion<10))          // Ниже SR2
    { AnsiString f1,f2;
      f1="Для работы программы необходима операционная система
не ниже Windows 98 SR2";
      f2="Системные требования";
      Application->MessageBox(f1.c_str(),f2.c_str(),MB_OK);
      Application->Terminate();
    }

```

3.2 Определение наличия в системе мыши

Для определения различных системных параметров и установок может быть использована функция API Win32 GetSystemMetrics.

Системные параметрами являются, например, размеры (ширина и высота) отображаемых окон. Все размеры функция GetSystemMetrics возвращает в пикселях.

```

int GetSystemMetrics(
    int nIndex // Возвращаемые системные параметры
);           // или установки

```

Параметры

nIndex

Определяет возвращаемые системные параметры или установки конфигурации. Все параметры SM_CX* обозначают значения ширины. Все параметры SM_CY* обозначают значения высоты. Определены следующие значения:

SM_ARRANGE	Флаг, определяющий, как система размещает минимизированные окна.
SM_CLEANBOOT	Значение, определяющее, как система стартовала: <ol style="list-style-type: none"> 0. Нормальный старт 1. Старт в безопасном режиме 2. Старт в безопасном режиме с сетевыми подключениями.

SM_CMOUSEBUTTONS

Количество кнопок на мыши, или нуль, если мыши нет.

SM_CXBORDER

SM_CYBORDER

Ширина и высота рамки окна в пикселях. Этот параметр эквивалентен **SM_CXEDGE** 3-D окон.

SM_CXCURSOR

SM_CYCURSOR

Ширина и высота курсора в пикселях. Размеры курсора поддерживаются текущим драйвером. Система не может создать курсор другого размера.

SM_CXDLGFRAME

SM_CYDLGFRAME

То же, что **SM_CXFIXEDFRAME** и **SM_CYFIXEDFRAME**.

SM_CXDOUBLECLK

SM_CYDOUBLECLK

Ширина и высота прямоугольника локализации вокруг первого и двойного кликов. Чтобы система распознала второй клик двойного клика, второй клик должен попасть внутрь этого прямоугольника. (Второй клик должен удовлетворять и определенным временным требованиям.)

SM_CXDRAG SM_CYDRAG

Ширина и высота прямоугольника, лимитирующего перемещения указателя мыши перед началом операции перетаскивания. Это позволяет пользователю легко нажимать и отпускать кнопки мыши без случайного выполнения операции перетаскивания.

SM_CXEDGE SM_CYEDGE

Размерность в пикселях 3-D рамки. Это 3-D эквивалент **SM_CXBORDER** и **SM_CYBORDER**.

SM_CXFIXEDFRAME

SM_CYFIXEDFRAME

Толщина рамки вокруг немасштабируемого окна с заголовком. **SM_CXFIXEDFRAME** – это ширина горизонтальной рамки, а **SM_CYFIXEDFRAME** – высота вертикальной. То же, что **SM_CXDLGFRAME** и **SM_CYDLGFRAME**.

SM_CXFRAME SM_CYFRAME

То же, что **SM_CXSIZEFRAME** и **SM_CYSIZEFRAME**.

SM_CXFULLSCREEN SM_CYFULLSCREEN	Ширина и высота клиентской области для полноэкранного режима. Для получения координат части экрана, не заслоняемой треем, следует вызывать функцию SystemParametersInfo со значением SPI_GETWORKAREA .
SM_CXHSCROLL SM_CYHSCROLL	Ширина в пикселях изображения стрелки на горизонтальной полосе прокрутки и высота в пикселях горизонтальной полосы прокрутки.
SM_CXHNTUMB	Ширина в пикселях ползунка горизонтальной полосы прокрутки.
SM_CXICON SM_CYICON	Ширина и высота пиктограммы по умолчанию в пикселях. Типовое значение 32x32, однако, оно сильно зависит от аппаратуры видеотракта. Функция LoadIcon может загружать пиктограммы только этого размера.
SM_CXICONSPACING SM_CYICONSPACING	Ширина и высота прямоугольника, используемого для размещения пиктограммы с заголовком. Это значение больше или равно SM_CXICON и SM_CYICON .
SM_CXMAXIMIZED SM_CYMAXIMIZED	Размер по умолчанию максимизированного окна верхнего уровня.
SM_CXMAXTRACK SM_CYMAXTRACK	Максимальный размер в пикселях по умолчанию окна, имеющего заголовок и изменяемые границы.
SM_CXMENUCHECK SM_CYMENUCHECK	Размер по умолчанию в пикселях изображения маркера меню.
SM_CXMENUSIZE SM_CYMENUSIZE	Размер в пикселях по умолчанию кнопок меню.
SM_CXMIN SM_CYMIN	Минимальные ширина и высота окна в пикселях.
SM_CXMINIMIZED SM_CYMINIMIZED	Размер в пикселях нормально минимизированного окна.

SM_CXMINSPACING
SM_CYMINSPACING

Размер в пикселях сетки для минимизированных окон. Каждое расставляемое минимизированное окно помещается в прямоугольник этого размера. Эти значения меньше или равны **SM_CXMINIMIZED** и **SM_CYMINIMIZED**.

SM_CXMINTRACK
SM_CYMINTRACK

Минимальная отслеживаемая ширина и высота окна в пикселях. Пользователь не может уменьшить окно до размеров, меньше указанных.

SM_CXSCREEN **SM_CYSCREEN**

Ширина и высота экрана в пикселях.

SM_CXSIZE
SM_CYSIZE

Ширина и высота кнопки в пикселях на полосе заголовка.

SM_CXSIZEFRAME
SM_CYSIZEFRAME

Толщина в пикселях размерной рамки по периметру окна, если размер окна может быть изменен. **SM_CXSIZEFRAME** это ширина горизонтальной границы, а **SM_CYSIZEFRAME** это высота вертикальной границы. То же, что **SM_CXFRAME** и **SM_CYFRAME**.

SM_CXSMICON **SM_CYSMICON**

Рекомендованный размер малой пиктограммы в пикселях. Малые пиктограммы обычно изображаются в заголовке окна и в режиме отображения маленьких иконок.

SM_CXSMSIZE **SM_CYSMSIZE**

Размер в пикселях малой кнопки заголовка.

SM_CXVSCROLL **SM_CYVSCROLL**

Ширина в пикселях вертикальной полосы прокрутки и высота в пикселях изображения стрелки вертикальной полосы прокрутки.

SM_CYCAPTION

Высота в пикселях нормального заголовка.

SM_CYKANJIWINDOW

Для двухбайтовой версии набора символов Windows высота и ширина окна Kanji в нижней части экрана.

SM_CYMENU

Высота в пикселях однострочной полосы меню.

SM_CYSMCAPTION	Высота в пикселях малого заголовка.
SM_CYVTHUMB	Высота в пикселях ползунка вертикальной полосы прокрутки.
SM_DBCSENABLED	TRUE или не нуль, если установлена двухбайтовая версия набора символов (DBCS) USER.EXE; FALSE или нуль, в противном случае.
SM_DEBUG	TRUE или не нуль, если установлена отладочная версия USER.EXE is installed; FALSE или нуль, в противном случае.
SM_MENUDROPALIGNMENT	TRUE или не нуль, если выпадающее меню выровнено справа относительно соответствующе элемента строки меню; FALSE или нуль, если левое выравнивание.
SM_MIDEASTENABLED	TRUE, если система разрешает использование восточных языков.
SM_MOUSEPRESENT	TRUE или не нуль, если установлена мышь; FALSE или нуль, в противном случае.
SM_MOUSEWHEELPRESENT	Только Windows NT: TRUE или не нуль, если установлена мышь с колесом прокрутки; FALSE или нуль, в противном случае.
SM_NETWORK	Младший значащий бит установлен, если сеть представлена; в противном случае, сброшен. Остальные разряды зарезервированы для дальнейшего использования.
SM_PENWINDOWS	TRUE или не нуль, если установлено Microsoft Windows световое перо; нуль или FALSE, в противном случае.
SM_SECURE	TRUE, если представлена система безопасности, FALSE, в противном случае.

SM_SHOWSOUNDS	TRUE или не нуль, если пользователю требуется визуальная информация, которая по умолчанию предоставляется в аудио-форме; FALSE или нуль, в противном случае.
SM_SLOWMACHINE	TRUE, если компьютер имеет медленный процессор, FALSE, в противном случае.
SM_SWAPBUTTON	TRUE или не нуль, если назначение левой и правой кнопок мыши поменяны местами; FALSE или нуль, в противном случае.

Возвращаемые значения

Если функция выполнена успешно, возвращается значение запрошенного системного параметра или установки конфигурации.

Если возникла ошибка при выполнении функции, возвращается значение «нуль». Функция GetLastError предоставляет дополнительную информацию об ошибке.

Примечания. Системные параметры зависят от типа видеоподсистемы.

SM_ARRANGE установки определяют, как система располагает минимизированные окна, в зависимости от стартовой позиции и направления. Стартовая позиция может принимать одно из следующих значений.

ARW_BOTTOMLEFT Начинает с левого нижнего угла экрана (позиция по умолчанию).

ARW_BOTTOMRIGHT Начинает с правого нижнего угла экрана. Эквивалентно **ARW_STARTRIGHT**.

ARW_HIDE Скрывать минимизированные окна при перемещении их за пределы видимой области экрана.

ARW_TOPLEFT Начинает с левого верхнего угла экрана.
Эквивалентно **ARV_STARTTOP**.

ARW_TOPRIGHT Начинает с правого верхнего угла экрана.
Эквивалентно **ARW_STARTTOP** | **SRW_STARTRIGHT**.

Направление, в котором могут располагаться окна, может принимать одно из следующих значений.

ARW_DOWN Вертикальное расположение, сверху вниз.

ARW_LEFT Горизонтальное расположение, слева направо.

ARW_RIGHT Горизонтальное расположение, справа налево.

ARW_UP Вертикальное расположение, снизу вверх.

```
// Проверка наличия мыши
if (GetSystemMetrics(SM_MOUSEPRESENT)==0)
{ Application->MessageBox("Для работы программы необходима
мышь", "Системные требования", MB_OK);
Application->Terminate();
}
```

3.3 Определение частоты процессора

Для определения частоты процессора можно использовать ассемблерную команду **RDTSC** (Read Time Stamp Counter). Это ассемблерная инструкция для платформы x86, читающая счётчик TSC (Time Stamp Counter) и возвращающая в регистрах EDX:EAX 64-битное количество тактов с момента последнего сброса процессора.

```
// Определение частоты процессора =====
AnsiString a;

try
{ unsigned __int64 count_1, count_2;
  unsigned int lo, hi;
  fasm(&lo, &hi);
  count_1=((unsigned __int64)hi)*(((unsigned
__int64)1)<<32)+(unsigned __int64)lo;
  Sleep(500);
  fasm(&lo, &hi);
```

```

        count_2=((unsigned __int64)hi)*(((unsigned
__int64)1)<<32)+(unsigned __int64)lo;
        frequency_proc=(int)(count_2-count_1)/(500000); //
Частота в МГц
    }
    catch(...) // Реакция на системное исключение [не может
выполниться
        // функция Fasm()]
    { AnsiString f1,f2;
      f1="Не тот тип процессора";
      f2="Системные требования";
      Application->MessageBox(f1.c_str(),f2.c_str(),MB_OK);

      Application->Terminate();
    }

// Системного исключения не было, вычисляем частоту процессора
if(frequency_proc<500)
{ AnsiString a,f2;
  f2="Системные требования";
  a="Частота процессора ЭВМ менее 500 МГц"+
    +AnsiString((int)frequency_proc)+ "МГц)\n";
  Application->MessageBox(a.c_str(),f2.c_str(),MB_OK);
}

```

Ассемблерная функция

```

//---- «Fasm» -----
void fasm(unsigned int *plo,unsigned int *phi)
{ unsigned int lo,hi;
  asm
  {
      rdtsc
      mov lo,eax
      mov hi,edx
  }
  *plo=lo;
  *phi=hi;
}

```

3.4 Определение объема оперативной памяти

Для получения информации о доступной оперативной памяти можно использовать функцию API Win32 GlobalMemoryStatus. Она возвращает информацию, как о физической, так и о виртуальной памяти.

```

VOID GlobalMemoryStatus(

```

```

LPMEMORYSTATUS lpBuffer    // Указатель на структуру,
    );                      // описывающую состояние памяти

```

Параметр:

LpBuffer

Указывает на структуру MEMORYSTATUS, которая возвращает информацию о текущей доступной памяти. Перед вызовом этой функции вызывающий процесс должен установить поле dwLength этой структуры

```

typedef struct _MEMORYSTATUS { // mst
    DWORD dwLength;           // sizeof(MEMORYSTATUS)
    DWORD dwMemoryLoad;       // Процент используемой памяти
    DWORD dwTotalPhys;        // Байтов физической памяти
    DWORD dwAvailPhys;        // Байтов свободной физической памяти
    DWORD dwTotalPageFile;    // Байтов в файле подкачки
    DWORD dwAvailPageFile;    // Свободных байтов в файле подкачки
    DWORD dwTotalVirtual;     // Байтов адресн. пространства пользов.
    DWORD dwAvailVirtual;     // Свободных байтов пользователя
} MEMORYSTATUS, *LPMEMORYSTATUS;

```

Поля

dwLength

Показывает размер структуры. Перед вызовом этой функции вызывающий процесс должен установить поле dwLength этой структуры (значением sizeof(MEMORYSTATUS)).

dwMemoryLoad

Число от 0 до 100, указывающее процент использования памяти, 0 говорит о том, что память не используется, 100 говорит о полном использовании памяти.

dwTotalPhys

Показывает количество байтов физической памяти.

dwAvailPhys

Показывает доступное количество байтов физической памяти.

dwTotalPageFile

Показывает количество байтов, которое может быть сохранено в файле подкачки. Следует иметь в виду, что это число не характеризует текущий размер физического файла подкачки на диске.

dwAvailPageFile

Показывает количество доступных байтов в файле подкачки.

dwTotalVirtual

Показывает общее количество байтов, которое может быть описано в виртуальном адресном пространстве пользователя вызывающего процесса.

dwAvailVirtual

Показывает количество незарезервированных и неподключенных байтов в виртуальном адресном пространстве пользователя вызывающего процесса.

```
// Определение объема установленной памяти =====
int mem_size;
MEMORYSTATUS mem;
mem.dwLength=sizeof(MEMORYSTATUS);
GlobalMemoryStatus(&mem);

mem_size=mem.dwTotalPhys;
mem_size=mem_size/(1024*1024);
a=AnsiString(mem_size)+" Мбайт";
// Application->MessageBox(a.c_str(),"Объем установленной
памяти",MB_OK);
if(mem_size<500)
{ AnsiString f1;
  f1="Системные требования";
  a="объем памяти ЭВМ менее 500 МГц"+AnsiString(mem_size)+
" Мбайт)\n";
  Application->MessageBox(a.c_str(),f1.c_str(),MB_OK);
}
```

3.5 Определение объема доступного дискового пространства

Для определения доступного дискового пространства может быть использована функция API Win32 GetLogicalDrives, которая возвращает битовую маску, представляющую текущие доступные диски.

DWORD GetLogicalDrives (VOID)

Параметр

Эта функция не имеет параметров.

Возвращаемые значения

При успешном выполнении функции она возвращает битовую маску, представляющую текущие доступные диски. Бит 0 (младший значащий бит) представляет диск A, бит 1 – диск B, бит 2 – диск C, и так далее.

Если функция завершилась неудачно, возвращается нулевое значение.

Для определения типа диска – сменный диск, фиксированный, CD-ROM, RAM диск или сетевой диск – может быть использована функция API Win32 **GetDriveType**.

```
UINT GetDriveType (  
    LPCTSTR lpRootPathName    // адрес пути к корневому  
    );                        // каталогу диска
```

Параметры

lpRootPathName

Указывает на нуль-терминированную строку, определяющую путь к корневому каталогу диска, о котором необходимо получить информацию. Если **lpRootPathName** – NULL, функция использует корневой каталог текущего диска.

Возвращаемые значения

Возвращаемые значения определяют тип диска. Возможно одно из следующих значений:

0 Тип диска не может быть определен.

1 Корневой каталог не существует.

DRIVE_REMOVABLE Диск может быть удален из дисковода.

DRIVE_FIXED Диск не может быть удален из дисковода.

DRIVE_REMOTE Удаленный (сетевой) диск.

DRIVE_CDROM Диск представляет собой CD-ROM.

DRIVE_RAMDISK Диск представляет собой RAM диск.

Для определения наличного свободного дискового пространства может быть использована функция API Win32 **GetDiskFreeSpace**, которая возвращает информацию об определенном диске, включая объем свободного пространства на диске.

```
BOOL GetDiskFreeSpace(  
    LPCTSTR lpRootPathName,    // адрес пути к корневому  
                               // каталогу диска  
    LPDWORD lpSectorsPerCluster,    // адрес переменной,  
                               // указывающей количество секторов в кластере  
    LPDWORD lpBytesPerSector,    // адрес переменной,  
                               // указывающей количество байтов в секторе  
    LPDWORD lpNumberOfFreeClusters, // адрес переменной,  
                               // указывающей общее количество свободных  
                               // кластеров на диске  
    LPDWORD lpTotalNumberOfClusters    // адрес переменной,  
                               // указывающей общее количество  
                               // кластеров на диске  
);
```

Параметры

lpRootPathName

Указывает на нуль-терминированную строку, определяющую путь к корневому каталогу диска, о котором необходимо получить информацию. Если **lpRootPathName** – NULL, функция использует корневой каталог текущего диска.

lpSectorsPerCluster

Указывает на переменную, содержащую количество секторов в кластере.

lpBytesPerSector

Указывает на переменную, содержащую количество байтов в секторе.

lpNumberOfFreeClusters

Указывает на переменную, содержащую общее количество свободных кластеров на диске.

lpTotalNumberOfClusters

Указывает на переменную, содержащую общее количество кластеров на диске.

Возвращаемые значения

Если функция выполнена успешно, возвращается ненулевое значение.

Если функция выполнена с ошибкой, возвращается нулевое значение. Для получения расширенной информации об ошибке следует вызвать функцию **GetLastError**.

Примечания

Windows 95:

Функция **GetDiskFreeSpace** возвращает некорректные значения для томов, объемом более 2 гигабайт. Функция обрезает значения, сохраняемые в ***lpNumberOfFreeClusters** и ***lpTotalNumberOfClusters** так, чтобы информация об объеме никогда не была более 2 гигабайт.

Даже для томов, меньших 2 гигабайт, значения, записываемые в ***lpSectorsPerCluster**, ***lpNumberOfFreeClusters**, и ***lpTotalNumberOfClusters** могут быть некорректными. Операционная система делает это для того, чтобы вычисленных из этих параметров объем тома получился корректным.

Windows 95 OSR 2:

Функция **GetDiskFreeSpaceEx** доступна в операционных системах Windows 95, начиная с OEM Service Release 2 (OSR 2). Функция **GetDiskFreeSpaceEx** возвращает корректные значения всех томов, включая тома, превышающие 2 гигабайта.

Для определения количества свободных байтов на определенном диске может быть использована более простая функция, включенная в Builder C++ **DiskFree**.

Эта функция возвращает количество свободных байтов на диске, причем 0 = текущий диск, 1 = A, 2 = B, и так далее.

Если указан неверный номер диска, функция возвращает значение -1.

```
//===== Определение дискового пространства =====
DWORD disk_present;
disk_present=GetLogicalDrives();
int n_disk=31; // Количество дисков вообще
for(int i=0;i<32;i++)
{ if(((disk_present<<i)&(1<<31))==0)
    n_disk--; // Количество дисков вообще (начиная со
              //старшего диска)
    else
        break; // Выход, как только найден старший диск
}
num_disk=0; // Количество разделов жестких дисков
for(int i=0;i<n_disk;i++)
{ a.sprintf("%c:\\",i+'c'); // Например, для I=2
                              //формируется строка E:\
    if(GetDriveType(a.c_str())==DRIVE_FIXED) // Проверка на
                                              // фикс. диск
        num_disk++; // Переход к следующему диску
    else
        break; // Выход с последним найденным фикс. диском
}
a="";
for(int i=0; i<num_disk;i++)
{ if(DiskFree(i+3)/(1024*1024)>1000) // В мегабайтах
    { a="aa"; // Взведение флага, если свободно
              // более 1 гигабайта
        break;
    }
}
if(a!="aa")
```

```

{ a="На дисках недостаточно свободного места. \n";
  "Всего на жестких дисках свободно: \n ";
  for(int i=0; i<num_disk;i++)

      { AnsiString b; // Формирование сообщения C:\ - XXX MB
        b.sprintf("%c:\ - ",i+'c'); // D:\ - YYY MB
                                     // и так далее
        a+=b+AnsiString(DiskFree(i+3)/(1024*1024)).c_str()+"
MB\n";
      }
  Application->MessageBox(a.c_str(),"Системные
требования",MB_OK);
}
//=====

```

Приведенные примеры, конечно, не охватывают все системные параметры, которые можно выяснить из программы пользователя. Здесь показано, как используя функции API Win32 и C++ Builder, можно определить такие важные для прикладной программы параметры, как наличный объем оперативной и дисковой памяти, параметры процессора и наличие мыши.

3.6 Вопросы для самопроверки

1. Как определить из программы пользователя номер версии операционной системы?
2. Что необходимо сделать перед вызовом функции API GetVersionEX?
3. Что возвращает функция API GetVersionEX при завершении ее с ошибкой?
4. Как узнать, с какой именно ошибкой завершилась функция API GetVersionEX?
5. Что такое старшая часть версии операционной системы?
6. Что такое младшая часть версии операционной системы?
7. Что такое номер сборки операционной системы?
8. Что такое платформа операционной системы?
9. Как определить из программы пользователя наличие мыши в системе?

10. Какие системные параметры возвращает функция API `GetSystemMetrics`?
11. Как определить тактовую частоту процессора из программы пользователя?
12. Как в программе пользователя предотвратить вывод сообщений об общесистемных исключениях?
13. Как из программы пользователя определить доступный объем оперативной памяти системы?
14. Как из программы пользователя определить процент использования памяти в системе?
15. Как из программы пользователя определить доступный для использования объем файла подкачки?
16. Как из программы пользователя определить доступный объем дискового пространства?
17. Как из программы пользователя определить наличие и количество имеющихся логических дисков?
18. Как из программы пользователя определить тип конкретного логического диска?
19. Какого типа диски позволяет определить функция API `GetDriveType`?
20. Как определить наличие свободного дискового пространства на каждом конкретном диске системы?
21. Как определить общий объем свободного дискового пространства в системе?
22. В каких операционных системах функция API `GetDiskFreeSpace` дает корректную информацию о томах, объемом более 2 гигабайт?
23. Какие функции имеются в составе пакета `Builder C++` для определения наличного свободного дискового пространства?
24. Что возвращает функция `Builder C++ DiskFree` при указании неверного номера запрашиваемого диска?

4 Отладка и тестирование программ

Турбо дебаггер (Turbo Debugger) – это отладчик, позволяющий отлаживать программы на уровне исходного текста. Многочисленные перекрывающиеся друг друга окна, а также сочетание выпадающих и раскрывающихся меню обеспечивают удобный пользовательский интерфейс. Интерактивная, контекстно-зависимая система подсказки обеспечит вас подсказкой на всех стадиях работы.

Ниже перечислены некоторые свойства турбо дебаггера:

- использование расширенной памяти типа EMS для отладки больших программ;
- вычисление любых выражений языка Си, C++, Паскаль и Ассемблера;
- настраиваемое размещение информации на экране;
- доступ к Ассемблеру и процессору по мере необходимости;
- мощные средства использования точек останова и протокола регистрации;
- запись нажатий клавиш (макрокоманды);
- средства обратной трассировки отлаживаемой программы;
- использование удаленной системы для отладки больших программ;
- поддержка процессора 80386 и аппаратных отладчиков прочих изготовителей;
- возможности отладки резидентных в памяти программ и драйверов устройств;
- возможности отладки прикладных программ Microsoft Windows.

4.1 Отладка и турбо дебаггер

Отладка – это процесс нахождения и исправления ошибок в программе. Нет ничего необычного в том, что поначалу поиск и устранение ошибок занимают больше времени, чем написание программы.

Отладка не является точной наукой. Часто лучшее средство отладки находится в голове у программиста. Тем не менее, систематический метод отладки может дать некоторые преимущества.

Процесс отладки в общем случае можно разделить на четыре этапа:

1. Обнаружение ошибки.
2. Поиск ее местонахождения.
3. Определение причины ошибки.
4. Исправление ошибки.

Обнаружение ошибки

Первый этап является наиболее очевидным. Компьютер либо "зависает" во время работы программы, либо происходит сбой, который проявляется в выдаче на экран бессмысленной информации. Однако, в некоторых случаях ошибка не проявляется так очевидно. Программа может работать хорошо до тех пор, пока не будет введено некоторое число (например, 0 или отрицательное число), или пока не будет тщательно проверена выдаваемая ею информация. Только после такой проверки можно обнаружить, что результат отличается от ожидаемого в 2 раза, или что в середине списка имен стоят неправильные инициалы.

Нахождение ошибки

Второй этап иногда является самым трудным. Он заключается в том, чтобы найти место в программе, где находится ошибка. Просто невозможно держать в голове всю программу сразу (если эта программа не очень маленькая). Лучший подход – это "разделяй и властвуй", то есть разбивать программу на части и отлаживать их отдельно друг от друга. Структурное программирование идеально подходит для такой отладки.

Определение причины ошибки

Третий этап, выяснение причины ошибки, возможно, является второй наиболее трудной стадией отладки. После того, как будет определено местонахождение ошибки, обычно становится несколько проще определить причину неправильной работы программы. Например, если вы определили, что ошибка находится в процедуре с именем *mes*, вам достаточно просмотреть текст только этой процедуры, а не всей программы. Но даже в этом случае ошибка может оказаться настолько "неуловимой", что вам придется немного поэкспериментировать, прежде чем вы сможете ее найти.

Исправление ошибки

Последний этап заключается в исправлении ошибки. Вооружившись знанием языка программирования и знанием местонахождения ошибки, вы устраняете ее. После этого вы снова запускаете программу, ждете появления следующей ошибки, и процесс отладки начинается снова.

При написании программы процесс, состоящий из этих четырех этапов, повторяется многократно. Например, многочисленные синтаксические ошибки не позволяют откомпилировать программу, пока все они не будут исправлены. Компиляторы фирмы Borland имеют встроенные средства проверки синтаксиса, которые информируют программиста об ошибках такого типа и позволяют тут же их исправлять.

Однако есть ошибки гораздо более тонкие и коварные, чем синтаксические. Они не проявляются до тех пор, пока вы не введете отрицательное число, либо являются настолько неуловимыми, что загоняют вас в тупик. Здесь-то вам и придет на помощь турбо дебаггер.

Что может дать турбо дебаггер?

Автономный турбо дебаггер обеспечивает доступ к гораздо более мощным средствам отладки, чем те, которые имеются в самом компиляторе.

Турбо дебаггер можно использовать для отладки любой программы на языке Си или C++ для компилятора семейства Borland, Турбо Паскале, Турбо Ассемблере.

Турбо дебаггер можно использовать для решения двух труднейших проблем процесса отладки: поиска места нахождения ошибки и ее причины. Он поможет преодолеть эти трудности с помощью исключительных возможностей по замедлению выполнения программы, благодаря чему можно исследовать состояние программы в любой заданной точке. Можно даже тестировать новые значения переменных, чтобы увидеть, как они воздействуют на программу. Эти возможности реализуются с помощью трассировки, пошагового выполнения, просмотра, изменения и прослеживания.

Трассировка	Позволяет выполнять программу по одному оператору.
Обратная трассировка	Позволяет выполнить код в обратном порядке.
Пошаговое выполнение	Позволяет выполнять программу по одному оператору, но пропускать вызовы процедур и функций. Если есть уверенность, что в процедурах и функциях нет ошибок, то пропуск их вызовов увеличит скорость отладки.
Просмотр	Позволяет запросить турбо дебаггер создать специальное окно для показа самых различных вещей – переменных, их значений, точек останова, содержимого стека, файлов регистрации, данных, файлов исходных текстов, кодов ЦП, памяти, регистров, информации процессора, арифметики с плавающей точкой, вывода программы.

Проверка	Позволяет запросить у турбо дебаггера содержимое сложных структур данных из программы.
Изменение	Позволяет изменить значение переменной (как локальной, так и глобальной).
Прослеживание	Позволяет выделить некоторые программные переменные и проследить изменение их значений в процессе работы программы.

Эти средства можно использовать для разделения программы на некоторые фрагменты, проверять работоспособность которых можно поочередно. Таким образом можно просмотреть всю программу, независимо от ее величины и сложности, и найти ошибку. Может быть, найдется функция, которая неверно изменяет значение переменной, или неправильная рекурсия, или попадание программы в бесконечный цикл. В любом случае турбо дебаггер помогает найти и определить тип ошибки.

Что турбо дебаггер не сможет сделать

Познакомившись со всеми этими возможностями турбо дебаггера, можно подумать, что он может все. В действительности, есть как минимум три вещи, которые турбо дебаггер не сможет сделать:

- Он не имеет встроенного редактора текстов для изменения исходной программы. Большинство программистов имеет свои излюбленные редакторы, и вполне довольны ими. Было бы расточительством памяти включать какой-либо редактор в турбо дебаггер. Однако вы можете легко передать управление вашему текстовому редактору, выбрав глобальную команду Edit (Редактирование) окна File (Файл) (о локальных командах рассказывается далее). Турбо дебаггер использует редактор, который задается с помощью программы установки TDINST.

- Турбо дебаггер не может перекомпилировать вашу программу. Для этого необходим соответствующий компилятор.
- Турбо дебаггер не сможет заменить процесс обдумывания. Во время отладки вашим главным инструментом является мысль. Турбо дебаггер является мощным средством, но при отсутствии ее, он не экономит ни времени, ни усилий.

4.2 Меню и диалоговые окна

В турбо дебаггере, как и во многих продуктах фирмы Borland, используются удобная система глобальных меню, доступных из строки меню, выводимой в верхней части экрана. За исключением того момента, когда активно диалоговое окно, система меню всегда доступна.

Главное меню

Главное меню доступно всегда, независимо от того, какое окно активно (т.е. в каком окне находится курсор).

Для каждого пункта (команды) главного меню имеется выпадающее меню. С помощью выпадающего меню можно:

- Выполнить команду.
- Открыть всплывающее меню. Это меню выводится после выбора элемента меню, за которым следует отметка (>).
- Открыть диалоговое окно. Это окно открывается, когда вы выбираете элемент меню, за которым следует многоточие (...).

Использование меню

Существует 4 способа выбора пунктов из главного меню:

- Нажать F10, подвести курсор к требуемому элементу меню и нажать клавишу Enter.
- Нажать клавишу F10 и клавишу с первой буквой элемента меню (F, V, R, B, D, O, W, H).

- Нажать клавишу Alt одновременно с первой клавишей пункта (элемента) меню (F, V, R, B, D, O, W, H) для активизации выбранного меню команд. Например, в любом месте системы нажатие клавиш Alt-F переместит вас в меню File (Файл). Системное меню будет открыто при нажатии клавиш Alt-пробел.
- Выбрать элемент (пункт) меню с помощью манипулятора типа "мышь" и нажать кнопку.

Для перемещения по всем меню, кроме главного, используйте следующие клавиши:

- *стрелки вправо/влево* – для перемещения от одного выпадающего меню к другому (например, когда вы находитесь в меню File, нажатие стрелки влево переместит вас в меню View),
- *стрелки вверх и вниз* – для перемещения по командам конкретного меню,
- *клавиши Home и End* для перемещения к первой и последней альтернативам (командам) меню, соответственно.

Для перемещения в меню или диалоговое окно более низкого уровня (всплывающее меню) используется клавиша Enter.

С этой же целью можно щелкнуть левой кнопкой мыши на нужной команде.

Выйти из меню или системы меню можно следующим образом:

- Нажать клавишу Esc для выхода из меню и возврата в предыдущее меню.
- Нажать клавишу Esc – если вы находитесь в меню второго уровня, это позволяет выйти из системного меню и возвратиться в предыдущее активное окно.
- F10 для возврата из меню любого уровня в предыдущее активное окно.

- Щелкнуть на активном окне левой кнопкой мыши для выхода из системы меню и возврата в это окно.

Некоторые команды главного (основного) меню соответствуют оперативным клавишам (сокращения команд). Там, где возможно, обозначения оперативных клавиш изображаются справа от альтернативы меню. В таблице 4.1 показано соответствие функциональных клавиш и команд.

Таблица 4.1 – Функциональные клавиши и соответствующие команды

Клавиша	Команда меню	Функция
F1		Выводит на экран контекстно-зависимую справочную информацию
F2	Breakpoints / Toggle (Точки останова / переключение)	Устанавливает в позиции курсора точку останова
F3	View/Module (Обзор / Модуль)	Выводит список для выбора модуля
F4	Run / Go to Cursor (Выполнение / Переход к курсору)	Выполняет программу до позиции курсора
F5	Window / Zoom (Окно / Переключение)	Переключает текущее окно
F6	Window / Next Window (Окно / Следующее окно)	Выполняет переход к следующему окну
F7	Run / Trace Into (Выполнение / Трассировка вглубь)	Выполняет одну исходную строку или инструкцию
F8	Run / Step Over (Выполнение / Шаг)	Выполняет одну исходную строку или инструкцию, пропуская вызовы
F9	Run / Run (Выполнение / Выполнение)	Выполняет программу
F10		Активизирует основное меню
Alt-F1	Help / Previous Topic (Справка / Предыдущая тема)	Выводит последний экран со справочной информацией

Alt-F2	Breakpoints / At (Точка останова / На ...)	Устанавливает точку останова по заданному адресу
Alt-F3	Window / Close (Окно / Закрытие)	Закрывает текущее окно
Alt-F4	Run / Back Trace (Выполнение / Обратная трассировка)	Выполняет программу «в обратном направлении»
Alt-F5	Window / User Screen (Окно / Экран пользователя)	Показывает экран вывода программы
Alt-F6	Window / Undo Close (Окно / Отменить закрытие)	Вновь открывает последнее закрытое окно
Alt-F7	Run / Instruction trace (Выполнение / Трассировка инструкции)	Выполняет одну инструкцию
Alt-F8	Run / Until Return (Выполнение / До возврата)	Выполняет программу до возврата управления из функции
Alt-F9	Run / Execute To (Выполнение / Выполнение до ...)	Выполняет программу до заданного адреса
Alt-F10		Вызывает локальное меню окна
Alt-1÷9		Переводит в окно заданным номером
Alt-Пробел		Активизирует системное меню
Alt-B		Активизирует меню Breakpoints (Точки останова)
Alt-D		Активизирует меню Data (Данные)
Alt-F		Активизирует меню File (Файл)
Alt-H		Активизирует меню Help (Справка)
Alt-O		Активизирует меню Options (Параметры)
Alt-R		Активизирует меню Run (Выполнение)
Alt-V		Активизирует меню View (Обзор)
Alt-W		Активизирует меню Window (Окно)
Alt-X		Выполняет выход из турбо дебаггера и возврат в DOS

Alt-=	Options / Macros / Create (Параметры / Макрокоманды / Создание)	Определяет клавиатурную макрокоманду
Alt-Минус	Options / Macros / Stop Recording (Параметры / Макрокоманды / Остановить запись)	Завершает запись макрокоманды
Ctrl-F2	Run / Program Reset (Выполнение / Сброс программы)	Останавливает сеанс отладки и сбрасывает состояние программы для повторного выполнения
Ctrl-F4	Data / Evaluate (Данные / Вычисления)	Вычисляет выражение
Ctrl-F5	Window / Size, Move (Окно / Размер, перемещение)	Инициализирует перемещение или изменение размера окна
Ctrl-F7	Data / Add Watch (Данные / Добавить выражение)	Добавляет переменную в окно просмотра (Watch)
Ctrl-F8	Breakpoints / Toggle (Точка останова / Переключение)	Переключает состояние точки останова в месте расположения курсора
Ctrl-F9	Run / Run (Выполнение / Выполнение)	Запускает программу на выполнение
Ctrl-F10		Вызывает локальное меню окна
Ctrl→		Сдвигает начальный адрес в области кода, данных или стека окна CPU (ЦП) на 1 байт вверх
Ctrl←		Сдвигает начальный адрес в области кода, данных или стека окна CPU (ЦП) на 1 байт вниз
Ctrl-A		Перемещение к предыдущему слову
Ctrl-C		Прокручивает вниз на один экран
Ctrl-D		Перемещает вправо на одну позицию
Ctrl-E		Перемещает вверх на одну позицию
Ctrl-F		Перемещает к следующему слову
Ctrl-R		Прокручивает вверх на один экран
Ctrl-S		Перемещает влево на одну позицию
Ctrl-X		Перемещает вниз на одну строку

Shift-F1	Help / Index	Переход к оглавлению оперативного справочника
Shift-Tab		Перевод курсора в предыдущую область окна или элемент
Shift-→		Перемещение курсора между областями в окне в соответствии с направлением стрелок (область в направлении стрелки становится текущей областью)
Shift-←		
Shift-↓		
Shift-↑		
Esc		Закрытие окна проверки (Inspector), выход из меню
Ins		Начало выбки блока текста (подсветка)
Tab	Window / Next Pane (Окно / Следующая область)	Перемещение курсора к следующей области окна или к следующему элементу диалогового окна

Для того, чтобы вызвать основное меню (строку меню) следует нажать клавишу F10. Для перехода в один из пунктов основного меню следует:

- переместив курсор на заголовок меню, нажать клавишу Enter,
- нажать первую букву подсвеченного элемента (пункта) меню.

Кроме того, можно непосредственно открыть меню (не перемещаясь сначала к заголовку меню), нажав клавишу Alt в сочетании с первой буквой имени нужного меню.

На рисунке 4.2 показано основное (системное) меню.

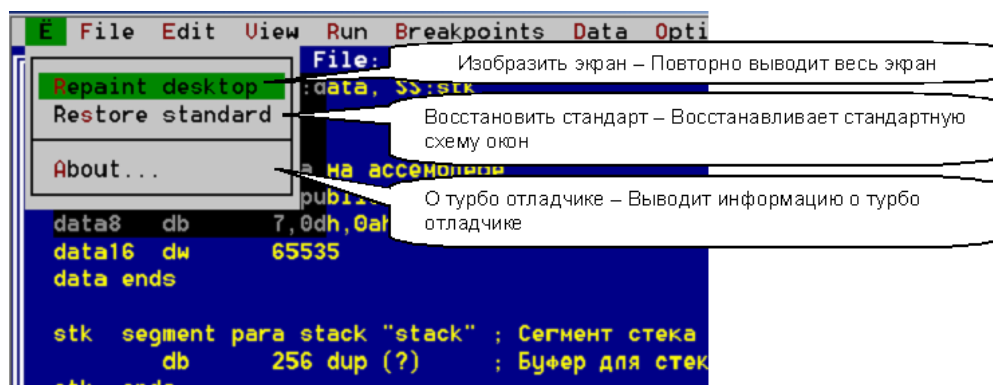


Рисунок 4.2 – Основное (системное) меню

На рисунке 4.3 показано меню File (Файл).

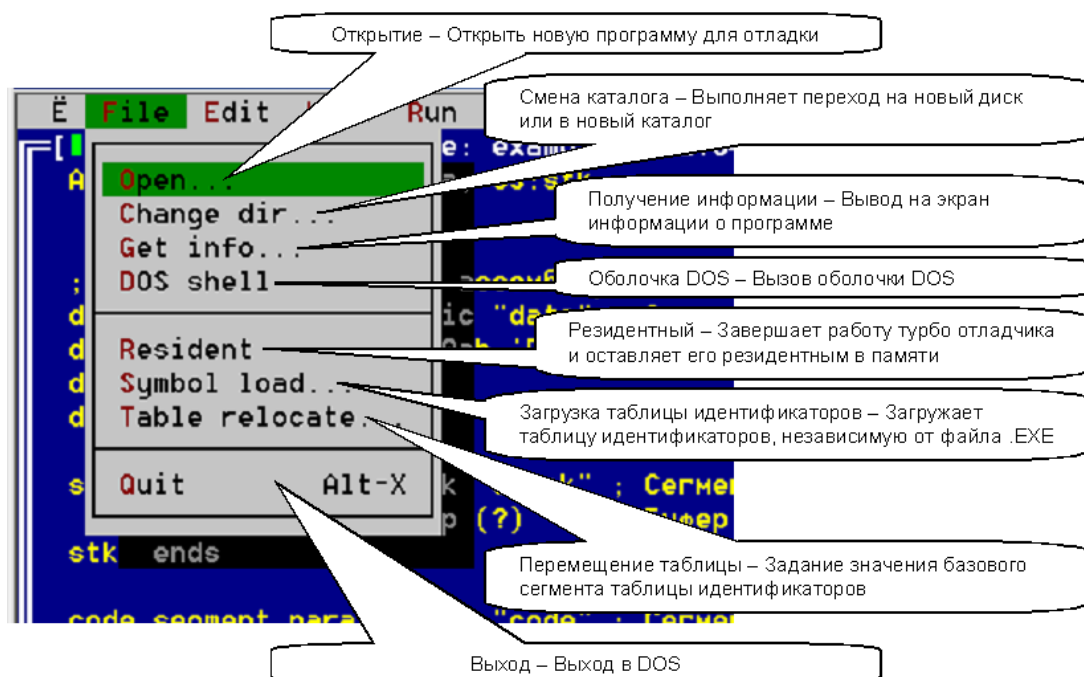


Рисунок 4.3 – Меню File (Файл)

На рисунке 4.4 показано меню Edit (Редактирование). На рисунках 4.5 – 4.11 – меню View (Обзор), Run (Выполнение), Breakpoints (Точки останова), Data (Данные), Options (Параметры), Window (Окно) и Help (Справка), соответственно.

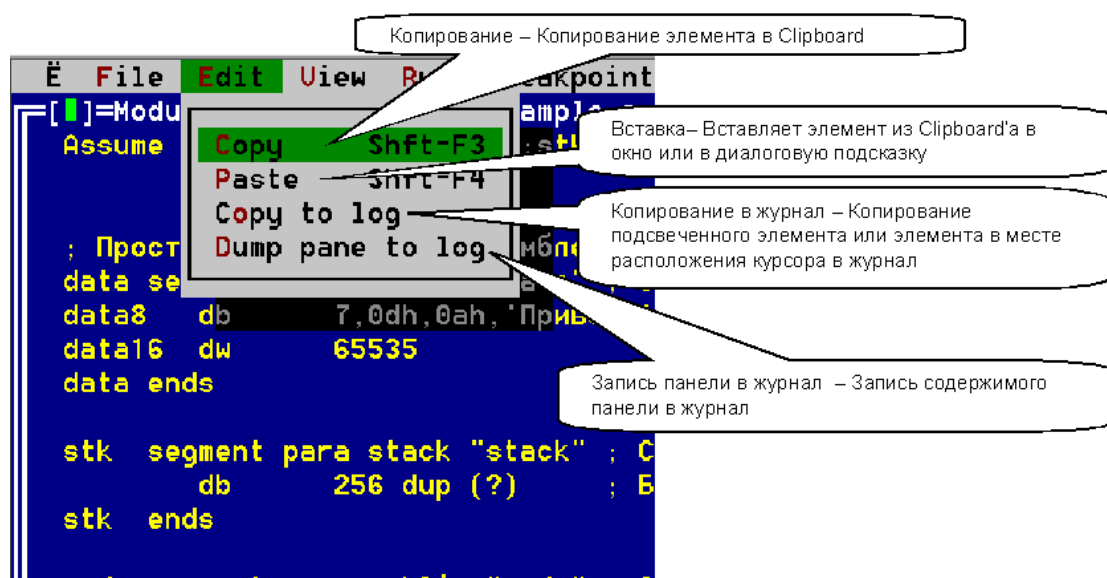


Рисунок 4.4 – Меню Edit (Редактирование)

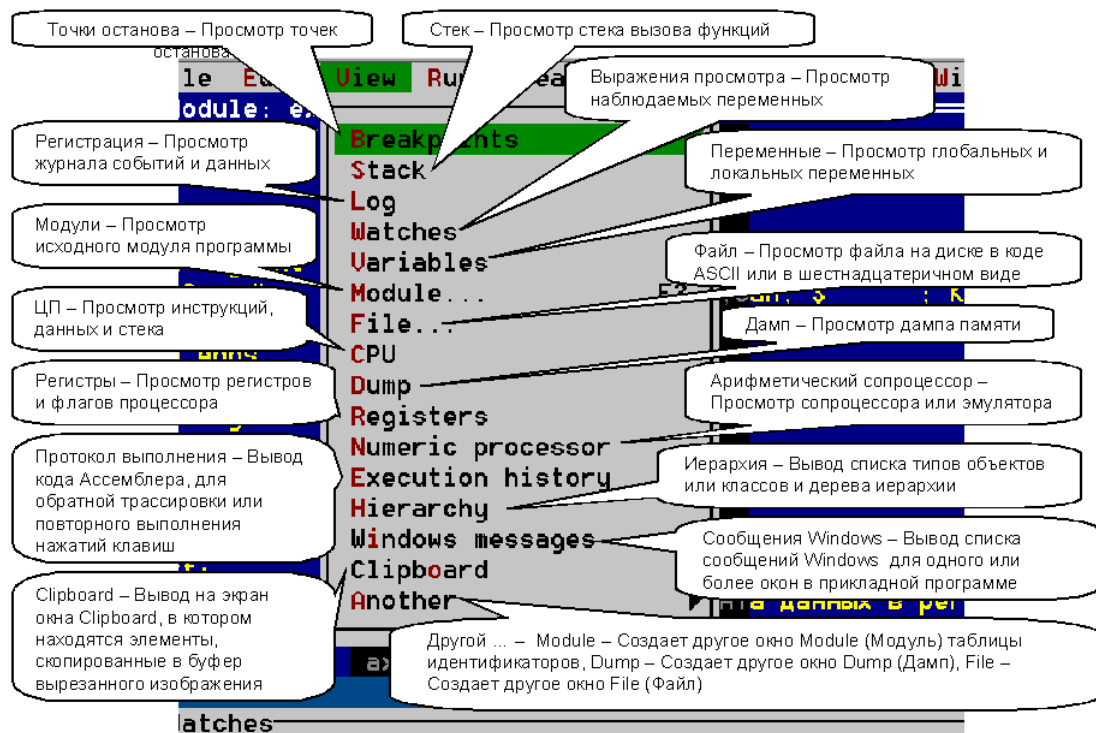


Рисунок 4.5 – Меню View (Обзор)

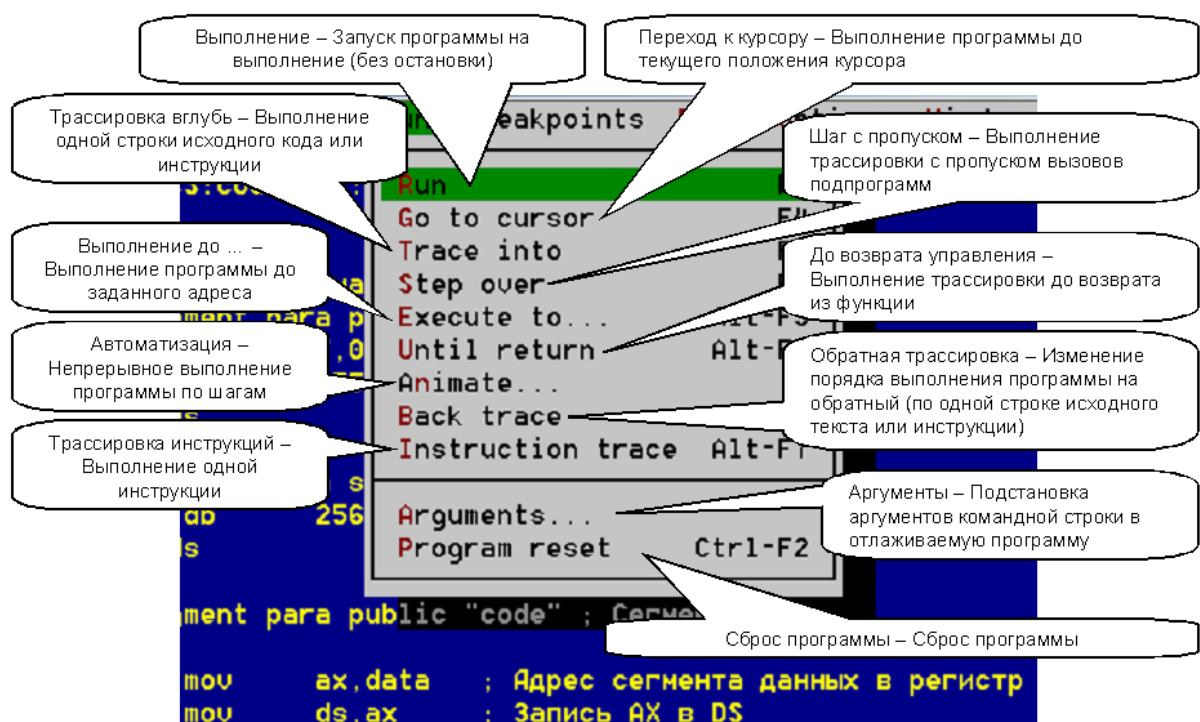


Рисунок 4.6 – Меню Run (Выполнение)

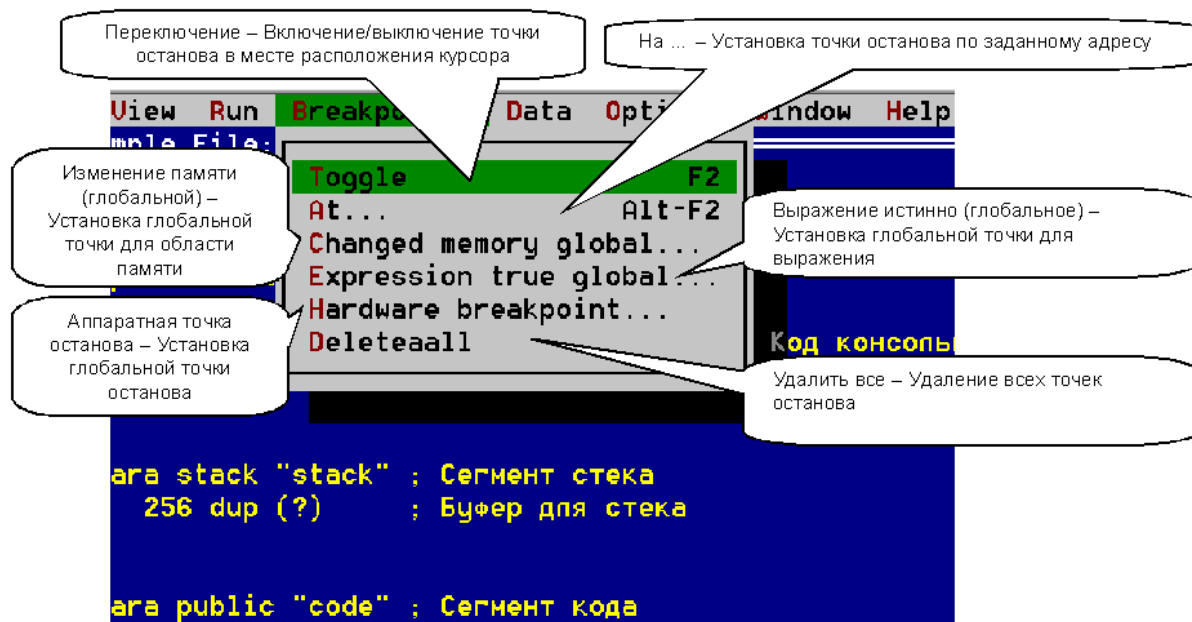


Рисунок 4.7 – Меню Breakpoints (Точки останова)

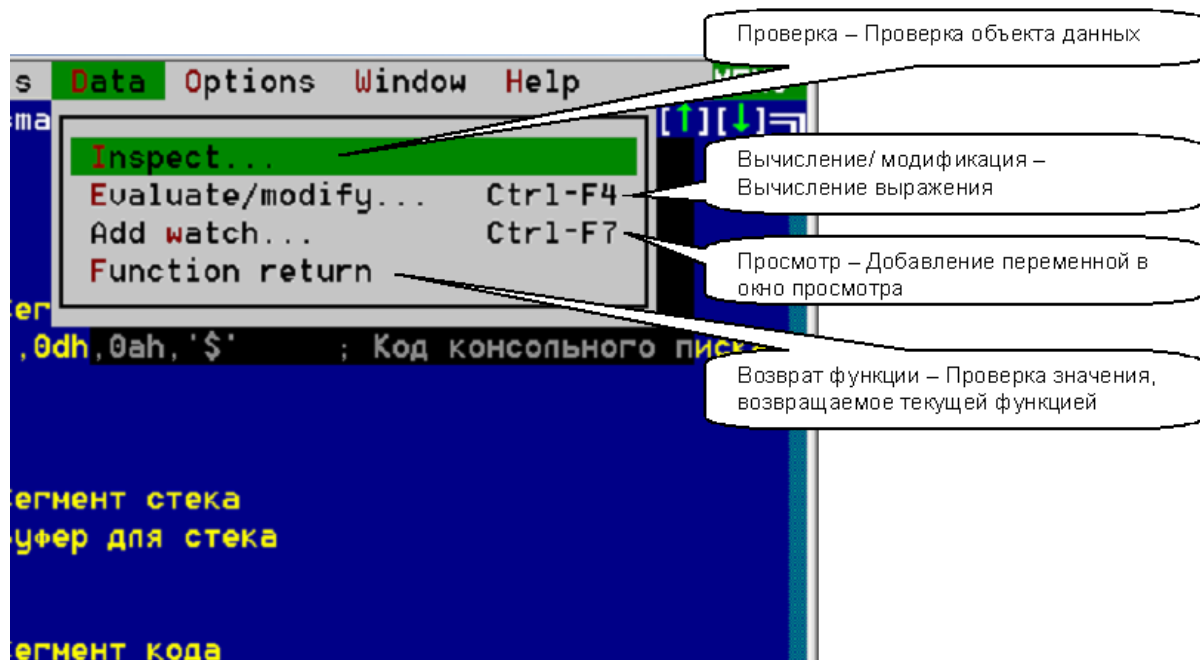


Рисунок 4.8 – Меню Data (Данные)

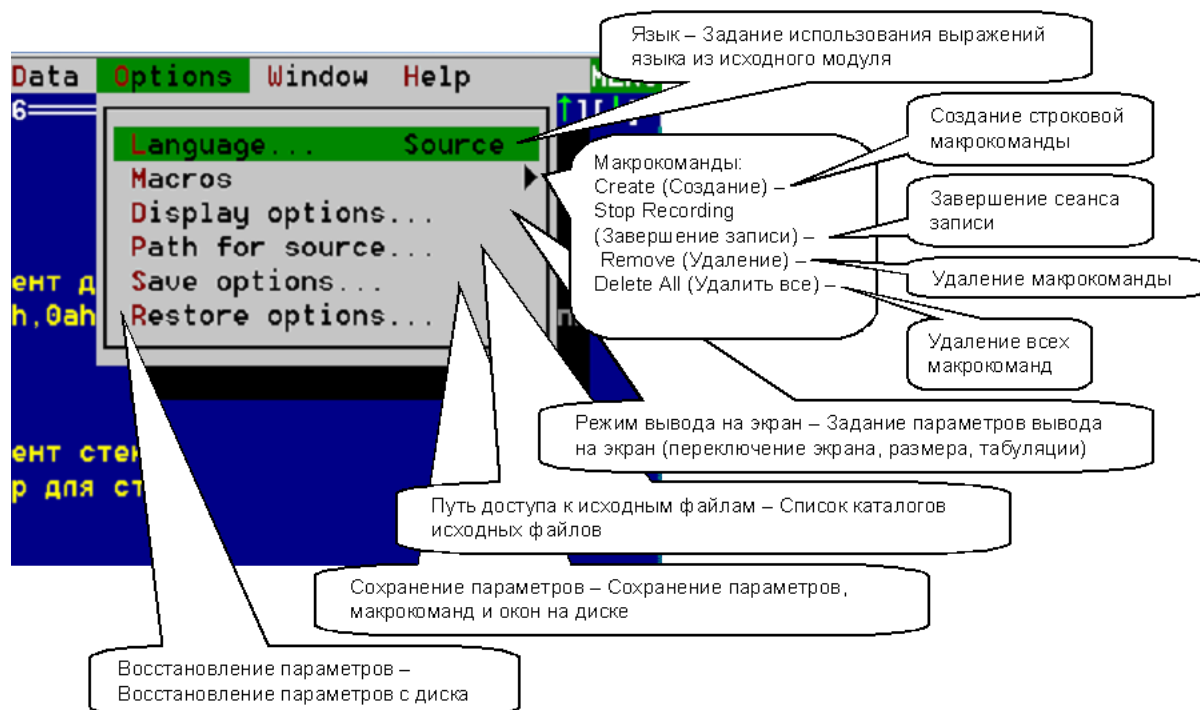


Рисунок 4.9 – Меню Options (Параметры)

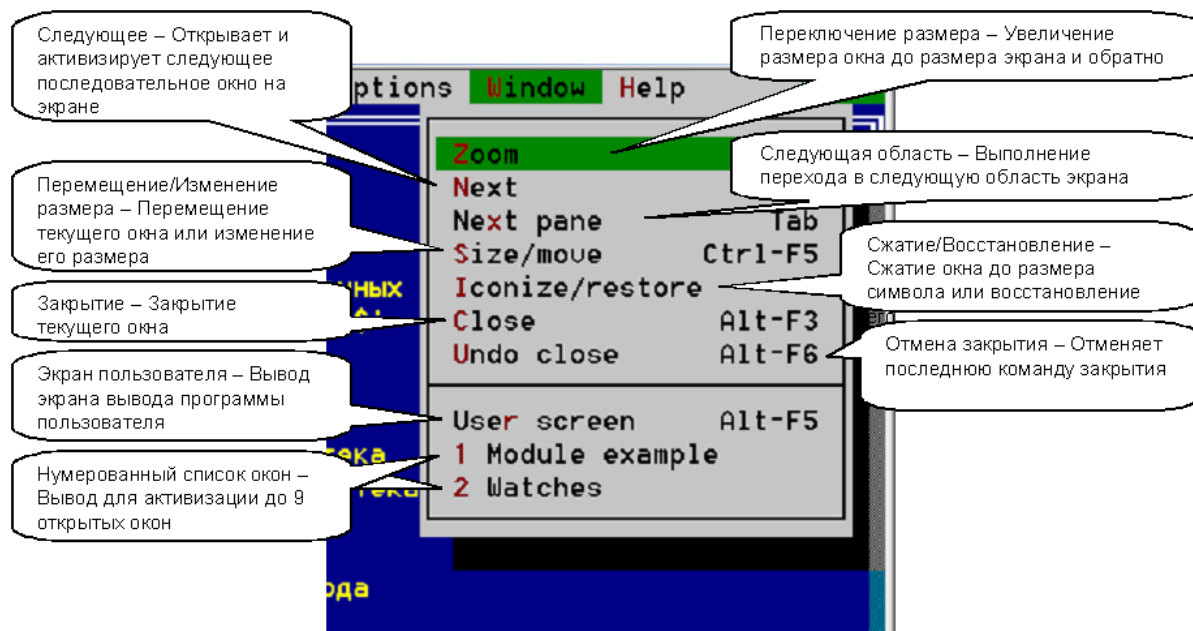


Рисунок 4.10 – Меню Window (Окно)

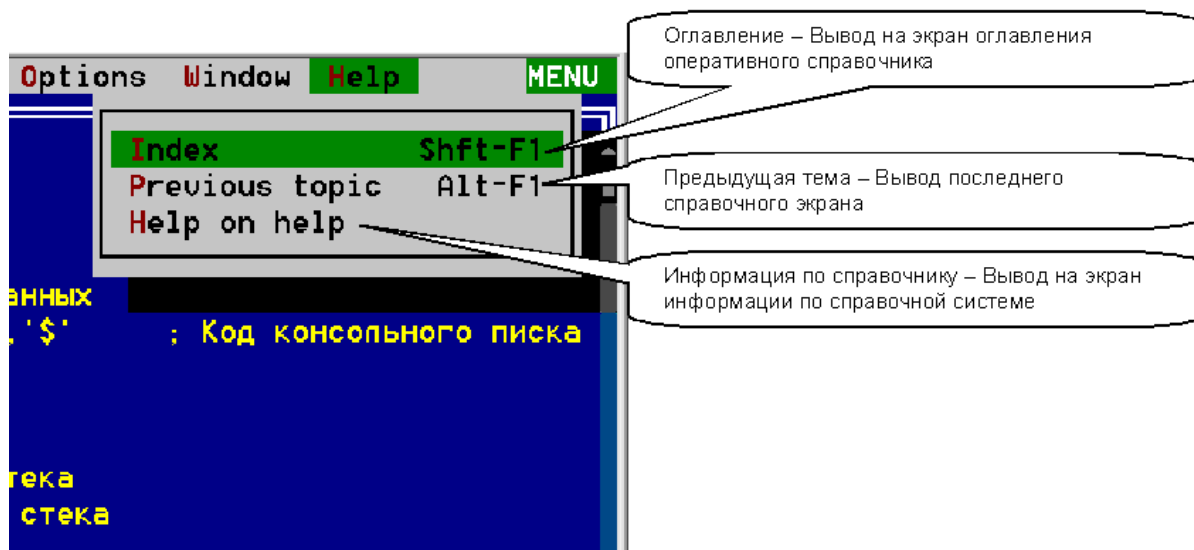


Рисунок 4.11 – Меню Help (Справка)

Для понимания этих возможностей следует запомнить, что турбо дебаггер является контекстно-зависимой программой. Он запоминает, какое окно открыто, какой текст выбран, и в какой части окна находится курсор. Другими словами, он точно знает, на что вы смотрите, и где находится курсор при выборе команды. Он использует эту информацию при реагировании на вашу команду. Рассмотрим пример для иллюстрации этого фундаментального вопроса.

Предположим в пользовательской программе на Ассемблере есть инструкция:

```
data16 dw 65535
```

При работе с турбо дебаггером получить информацию о переменной можно, нажав клавиши Ctrl-I (проверка). Если курсор находится под именем переменной, то будет выведено окно просмотра данных с указанием адреса переменной, ее формата и значения (рисунок 4.12).

Локальные меню

Другим аспектом контекстной зависимости турбо дебаггера является использование локальных меню (зависящих от обстоятельств). Локальные меню в турбо дебаггере настроены на конкретное активные окно или

область. Очень важно не путать локальные меню с глобальными (однако при реальной работе в отладчике оба типа меню никогда одновременно не выводятся). На рисунке 4.13 показана работа с локальным меню окна CPU.

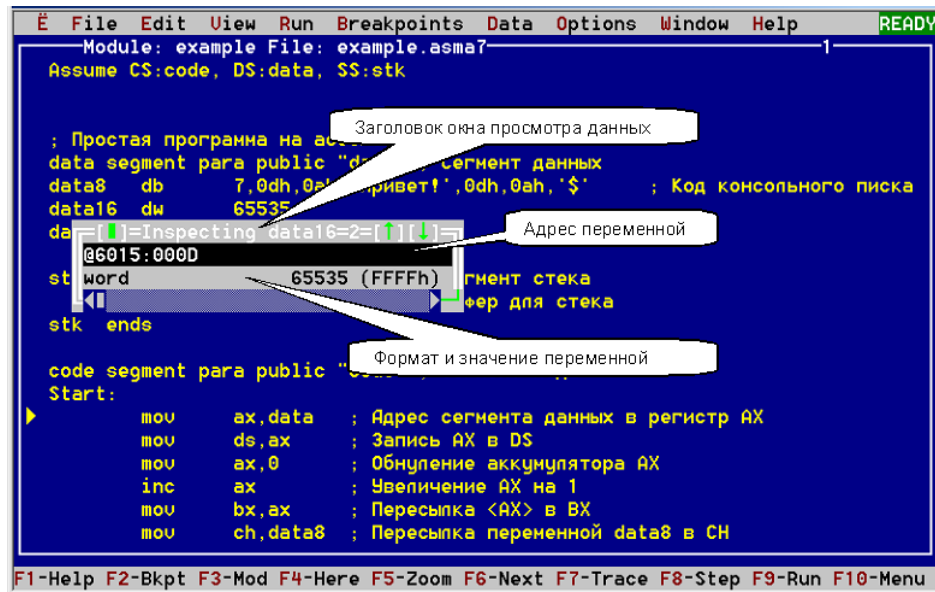


Рисунок 4.12 – Просмотр переменной с помощью контекстно-зависимой команды Ctrl-I

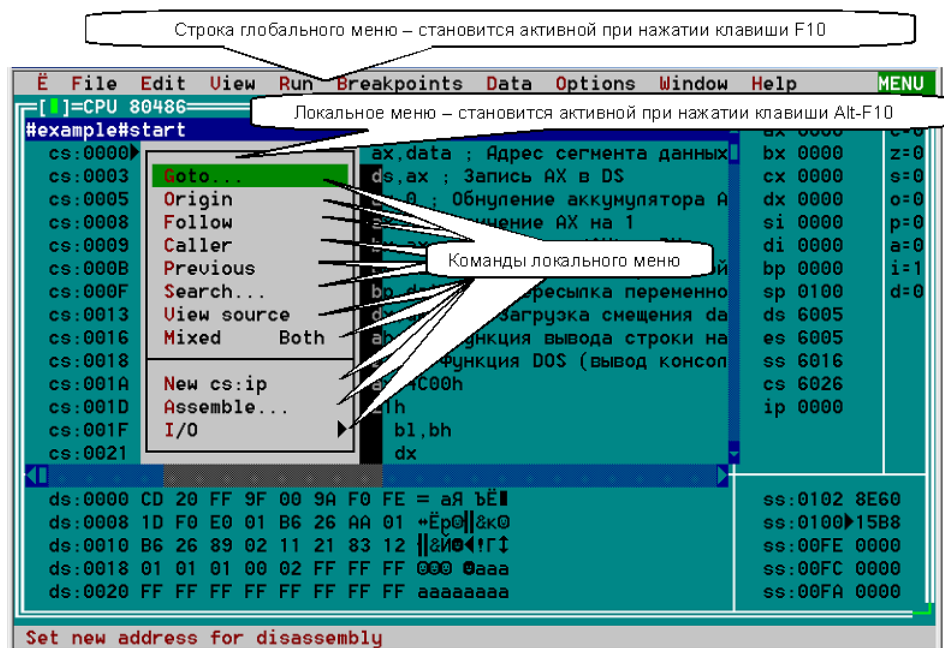


Рисунок 4.13 – Глобальное и локальные меню

Содержимое локального меню может изменяться (важно знать, что локальные команды появляются почти во всех локальных меню, так что множества команд каждого меню предсказуемо), но даже в этих случаях результат команд с одинаковыми именами может быть различным в зависимости от контекста.

Каждая команда локального меню имеет функциональный эквивалент, состоящий из нажатия клавиши Ctrl, плюс первой буквы команды. Вследствие этого функциональный эквивалент, например, Ctrl-S, может означать в одном контексте одно, а в другом контексте – совершенно другое, однако существует соответствие множества команд в последовательности локальных всплывающих меню. Например, команды Goto (Переход) или Search (Поиск) всегда делают одно и то же, даже вызванные из разных мест.

4.3 Команды локальных меню

Для текущего окна вызвать всплывающее или "локальное" меню можно с помощью клавиш Alt-F10. Если разрешено использовать сокращения с клавишей Ctrl (разрешить это можно с помощью программы установки TDINST), то к отдельным элементам этого меню можно обратиться непосредственно с помощью клавиши Ctrl и первой буквы нужного элемента (команды) меню (нажав их одновременно). Следует учитывать, что каждый тип окна и каждая область окна содержат разные локальные меню.

Ниже описаны локальные меню для каждого окна и области. Для удобства поиска меню в данном разделе упорядочены по алфавиту.

Некоторые области в своих локальных меню могут содержать общие команды (их сокращения для оперативных клавиш). В следующих разделах эти специальные клавиши описываются перед командами меню для той области, к которой они относятся. Во многих областях окон

клавиша Enter представляет собой сокращение для проверки или изменения текущего (подсвеченного) элемента. Клавиша Del часто вызывает команду локального меню, которая удаляет подсвеченный элемент. Некоторые области позволяют вам начать ввод букв или цифр без предварительного вызова команды локального меню. В этом случае выводится рамка (окно) подсказки, куда можно вводить данные.

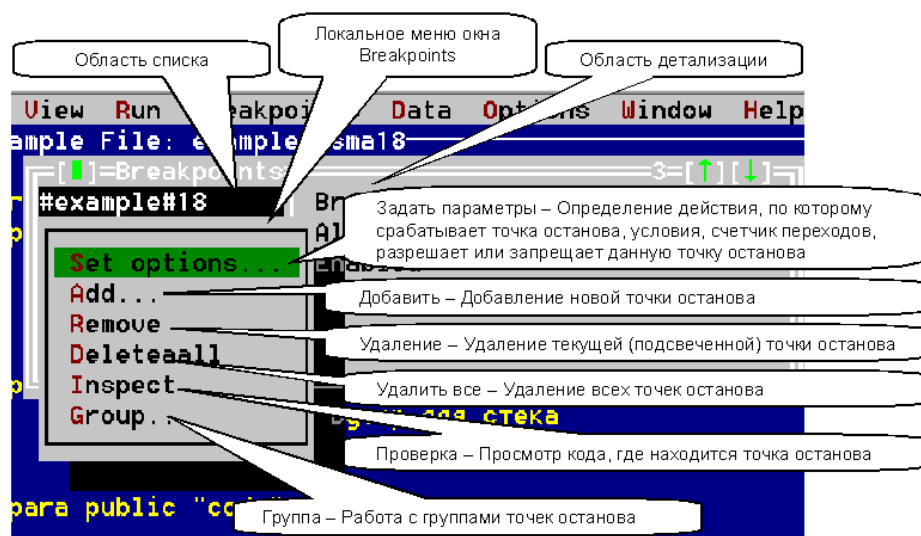


Рисунок 4.14 – Локальное меню окна Breakpoints

Локальное меню окна Breakpoints (Точки останова)

Окно Breakpoints содержит две области (рисунок 4.14): область списка (слева) и область детализации (справа). Локальное меню имеет только область списка.

В данном окне в качестве сокращения команды Remove (Удаление) используется клавиша Del.

Меню окна CPU (ЦП)

Окно CPU (Центральный процессор) имеет пять областей (область кода, область данных, область стека, область регистров и область флагов), и в каждой области имеется локальное меню.

Область кода

Вместо команды локального меню Assemble (Ассемблирование) можно использовать сокращенный вариант: набор любого символа.

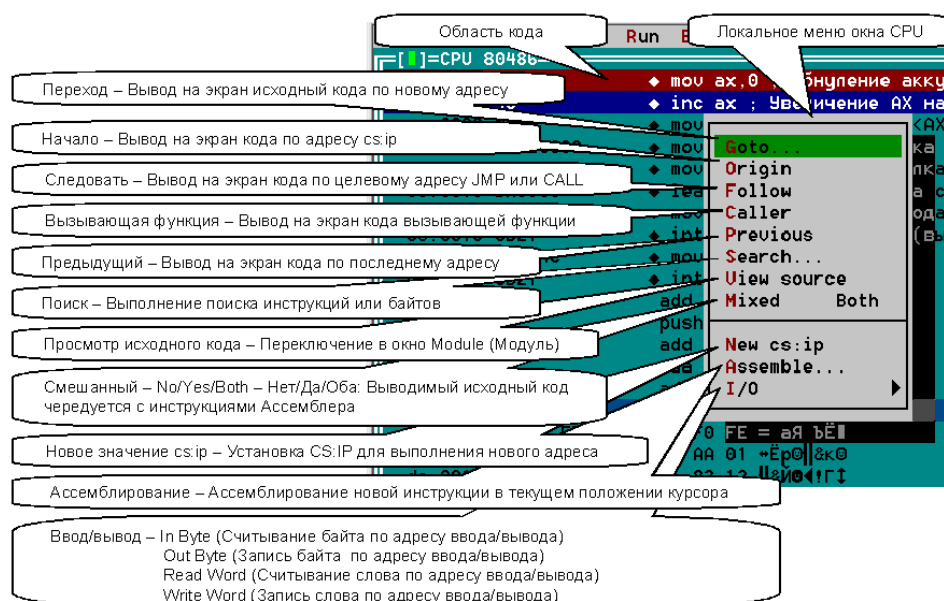


Рисунок 4.15 – Локальное меню области кода окна CPU

Область данных

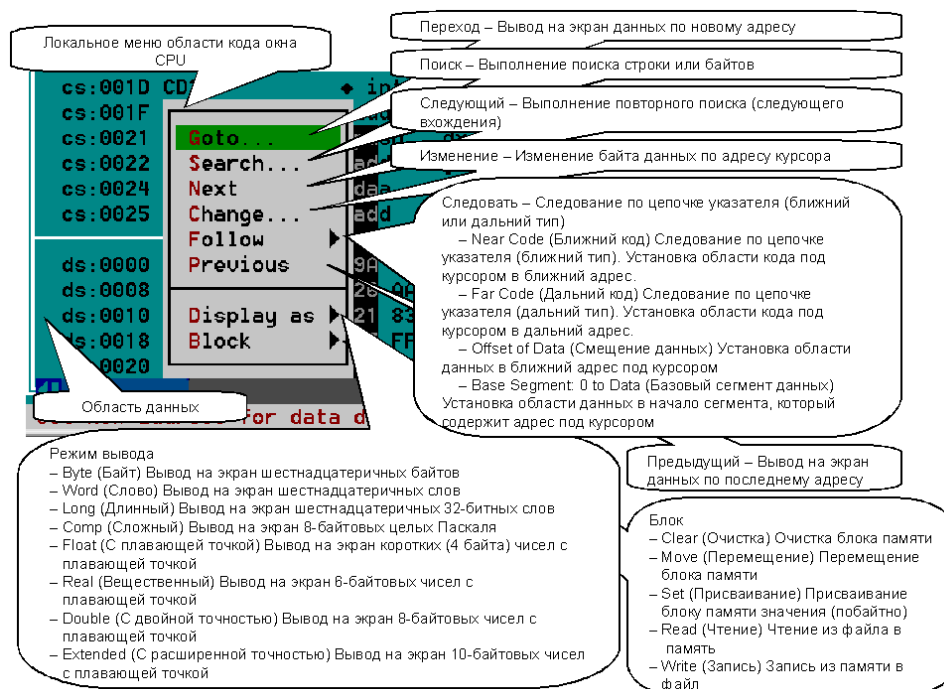


Рисунок 4.16 – Локальное меню области данных окна CPU

Вместо команды локального меню данной области Change (Изменение) можно использовать сокращенный вариант: набор любого символа.

Область флагов

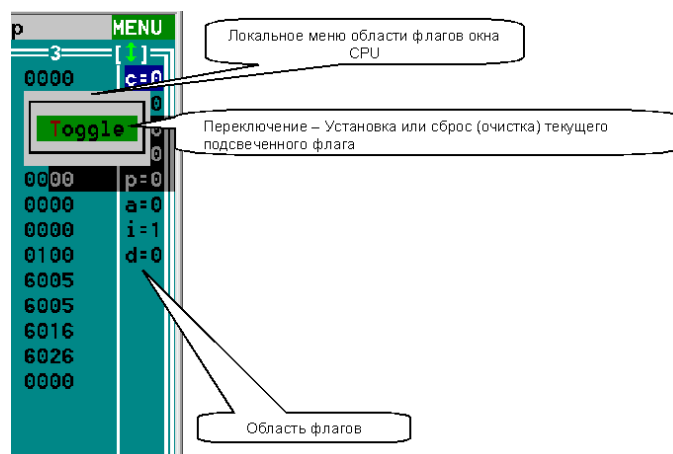


Рисунок 4.17 – Локальное меню области флагов окна CPU

В качестве сокращений данной команды можно использовать клавиши Enter или "пробел".

Область регистров

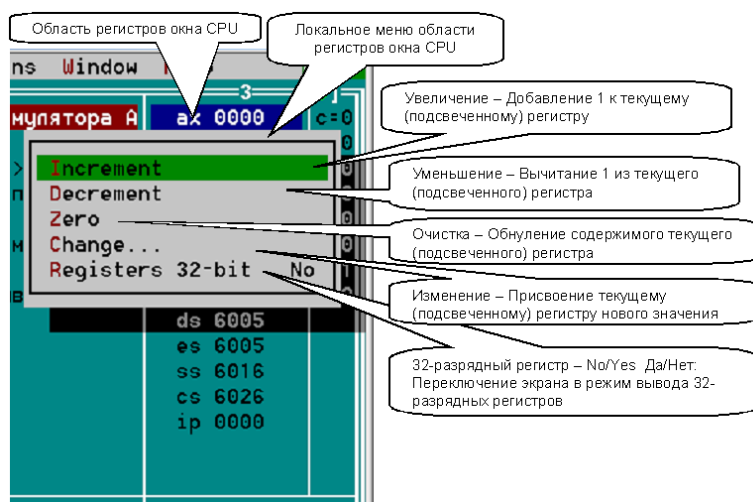


Рисунок 4.18 – Локальное меню области регистров окна CPU

Область стека

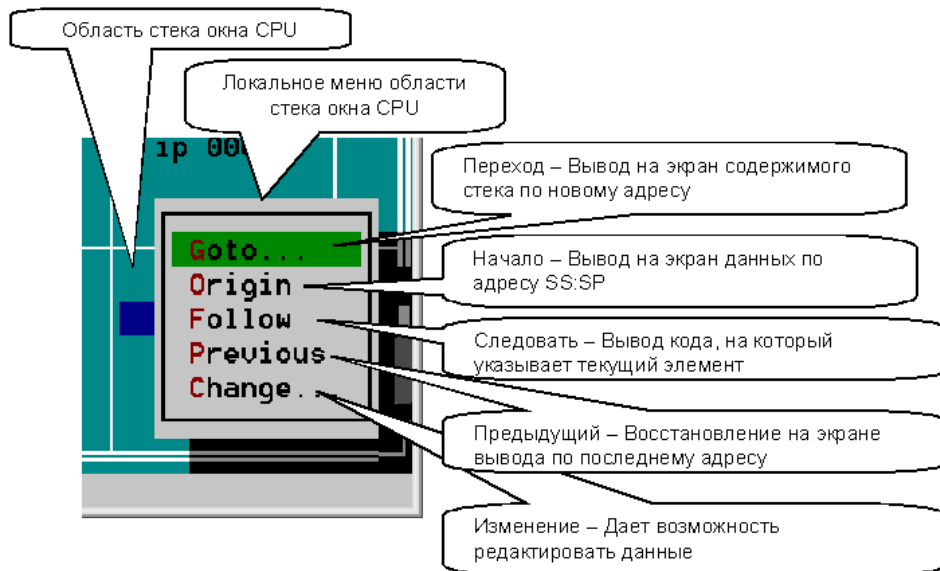


Рисунок 4.19 – Локальное меню области стека окна CPU

Окно Dump (Дамп)

Окно Dump идентично области данных окна CPU (ЦП). Их локальные меню также эквивалентны.

Окно File (Файл)

В окне File (Файл) в текстовом или шестнадцатеричном виде выводится содержимое файла на диске (рисунок 4.21).

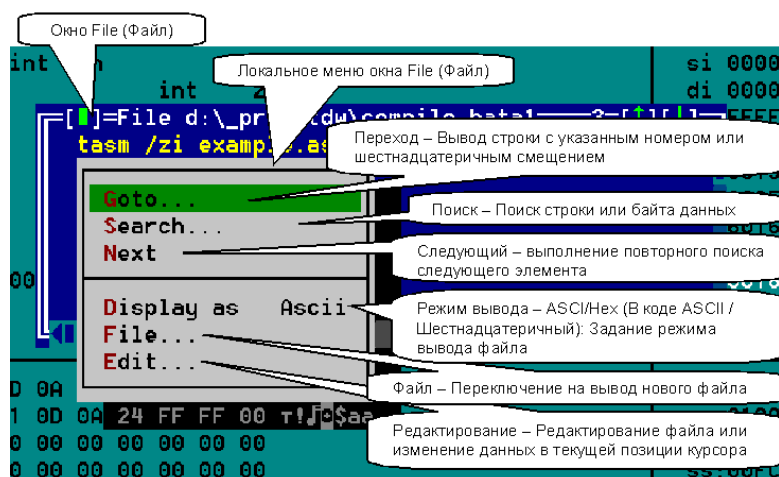


Рисунок 4.20 – Локальное меню окна File (Файл)

Вместо команды локального меню Search (Поиск) можно использовать сокращенный вариант: набор любого символа.

Локальное меню окна Log (Регистрация)

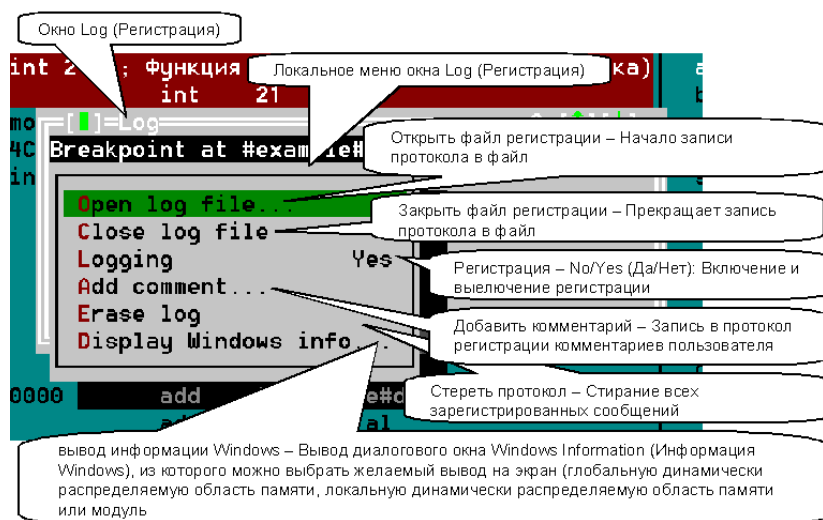


Рисунок 4.21 – Локальное меню окна Log (Регистрация)

В окне Log (Регистрация) выводятся сообщения, переданные для регистрации (протокол) (рисунок 4.21).

Вместо команды локального меню Add Comment (Добавить комментарий) можно использовать сокращенный вариант: набор любого символа.

Окно Module (Модуль)

В окне Module (Модуль) выводится содержимое исходного файла программного модуля.

Окно Clipboard

В окне Clipboard выводятся все элементы, которые вы скопировали в карман. Оно имеет единственную область со следующими командами локального меню (рисунок 4.23).

Окно Numeric Processor (Сопроцессор)

Окно Numeric Processor (Арифметический сопроцессор) содержит три области: область регистров, область состояния и область управления.

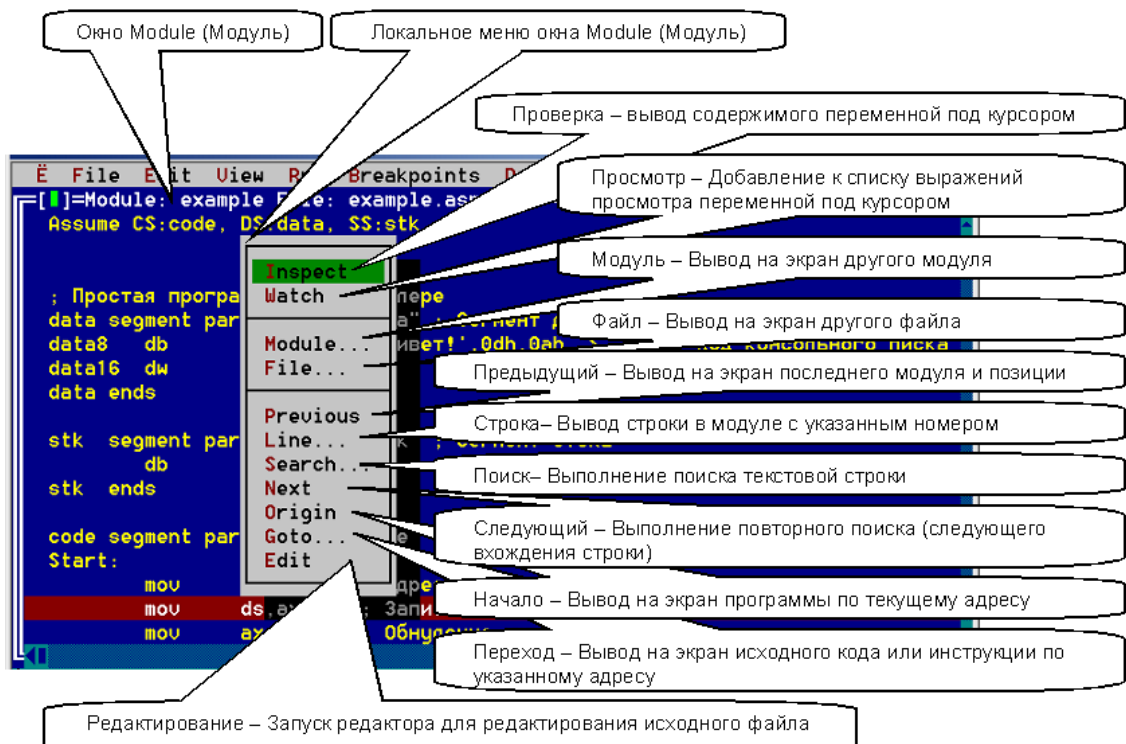


Рисунок 4.22 – Локальное меню окна Module (Модуль)

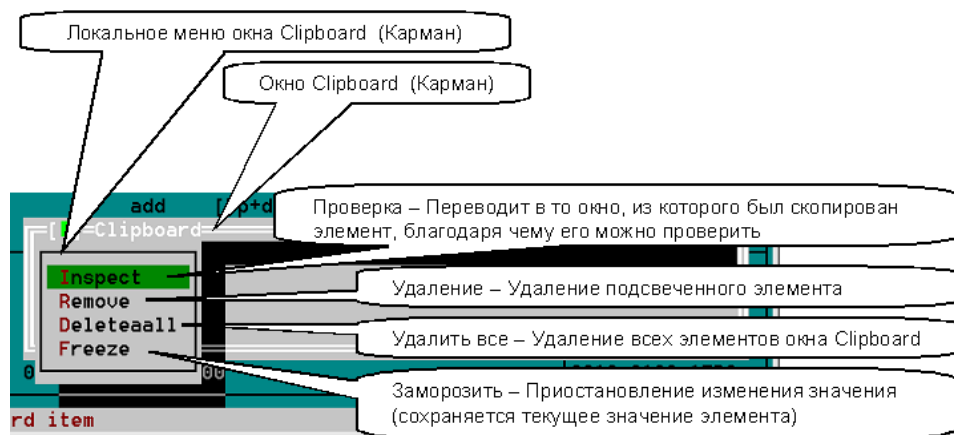


Рисунок 4.23 – Локальное меню окна Clipboard

Область регистров

В качестве сокращенных вариантов команд локального меню данной области можно использовать следующие клавиши (рисунок 4.24).

Вместо команды локального меню данной области Change (Изменение) можно использовать сокращенный вариант: набор любого символа.

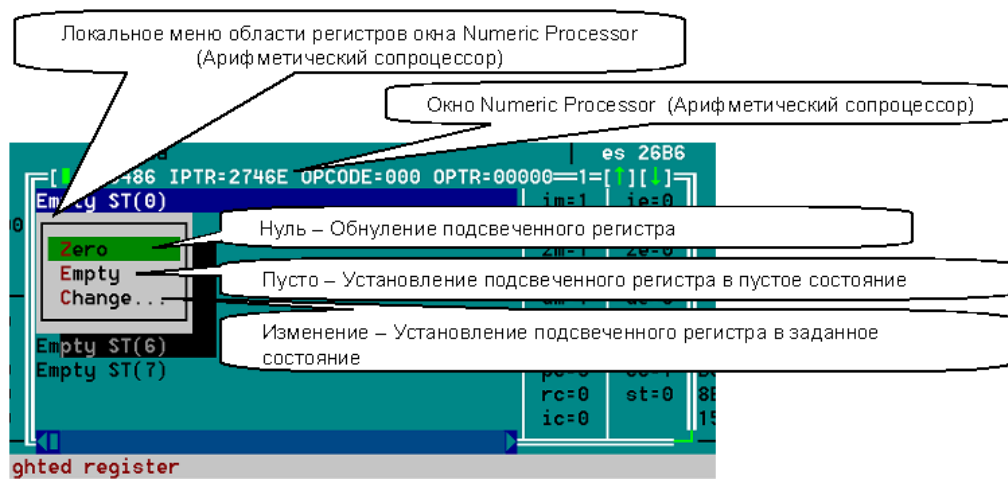


Рисунок 4.24 – Локальное меню области регистров окна Numeric Processor

Область состояния

В качестве сокращенного варианта данной команды локального меню можно использовать просто нажатие клавиши Enter или пробел.

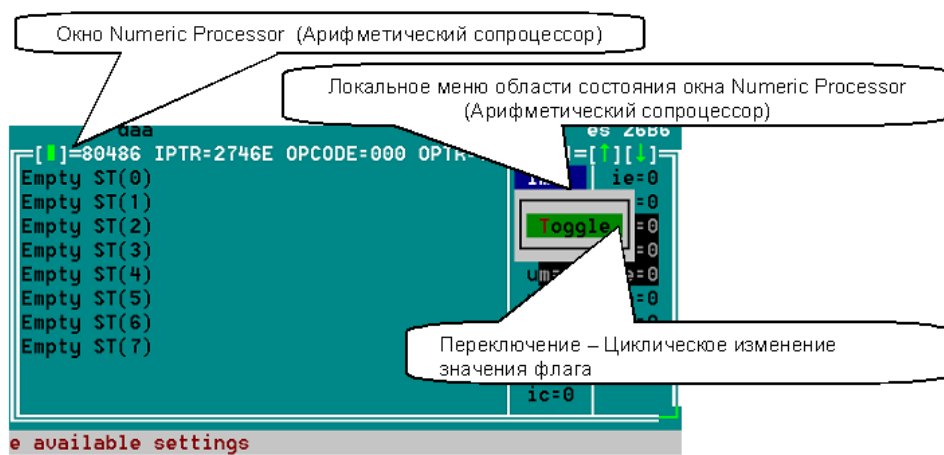


Рисунок 4.25 – Локальное меню области состояния окна Numeric Processor

Область управления

В качестве сокращенного варианта данной команды локального меню можно использовать просто нажатие клавиши Enter или пробел.

Меню окна Registers (Регистры)

Окно Registers (Регистры) идентично областям регистров и флагов окна CPU (ЦП). Его локальные меню идентичны локальным меню области регистров и области флагов.

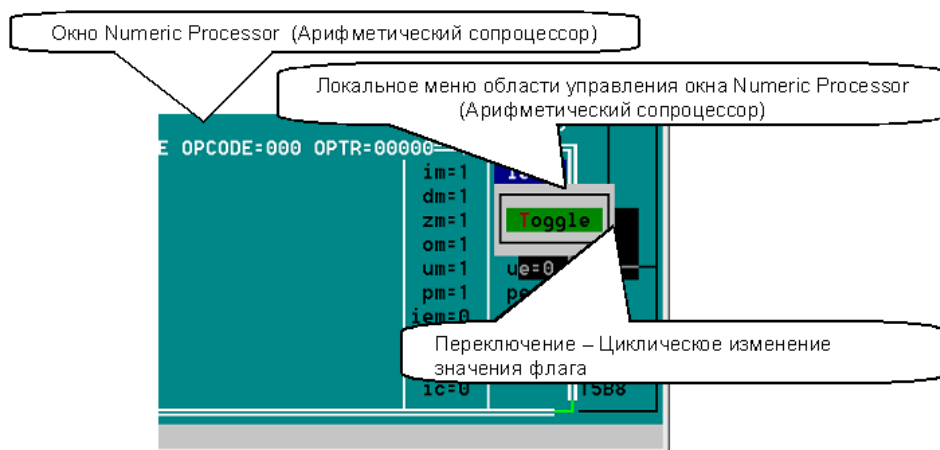


Рисунок 4.26 – Локальное меню области управления окна Numeric Processor

Окно Stack (Стек)

В области стека выводятся активные в данный момент процедуры (рисунок 4.27).

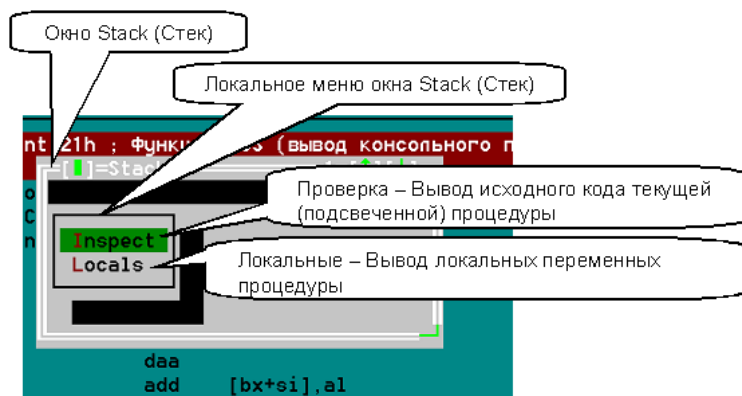


Рисунок 4.27 – Локальное меню окна Stack (Стек)

В качестве сокращенного варианта команды локального меню Inspect можно использовать просто нажатие клавиши Enter.

Окно Variables (Переменные)

Это окно разделено на две области, у каждой из которых имеется свое локальное меню: область глобальных идентификаторов и область локальных идентификаторов (рисунки 4.28 и 4.29).

Область глобальных идентификаторов

Нажатие клавиши Enter является сокращенной формой команды Inspect локального меню данной области.

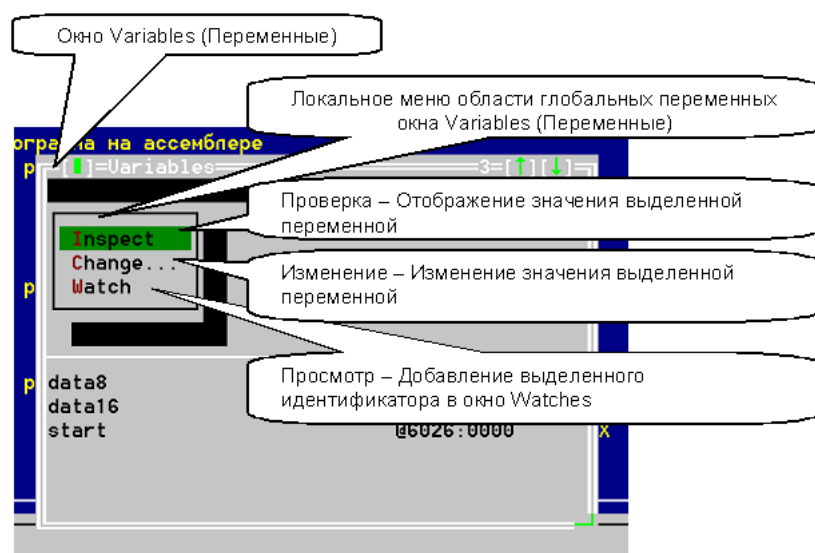


Рисунок 4.28 – Локальное меню области глобальных переменных окна Variables

Область локальных идентификаторов

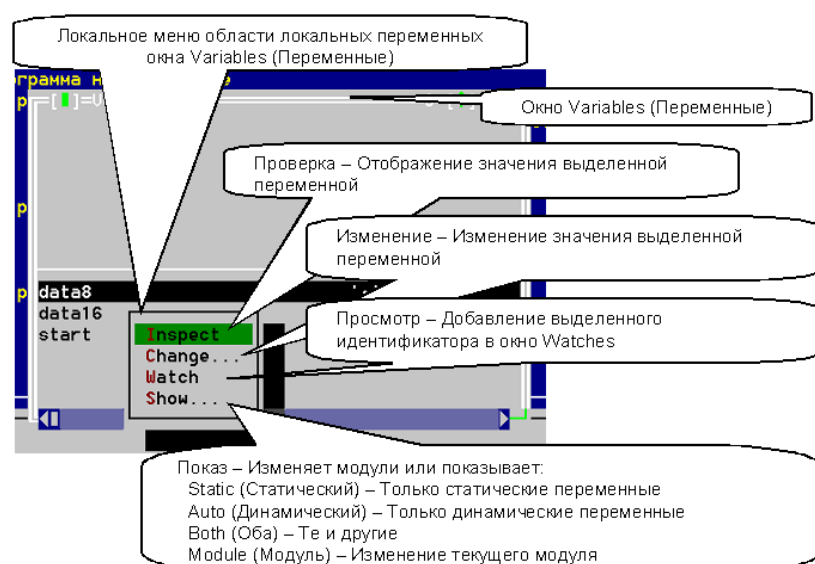


Рисунок 4.29 – Локальное меню области локальных переменных окна Variables

В качестве сокращенного варианта команды локального меню Inspect можно использовать просто нажатие клавиши Enter.

Окно *Watches* (Просмотр)



Рисунок 4.30 – Локальное меню окна *Watches* (Просмотр)

Окно *Watches* (Просмотр) содержит единственную область, в которой выводятся имена и значения просматриваемых переменных (рисунок 4.30).

4.4 Пример отладки простой программы

Для иллюстрации использования турбо дебаггера рассмотрим простой пример программы на ассемблере. В данном разделе рассмотрены основные средства турбо дебаггера. Пример программы называется *Example.asm*, и приведен ниже.

```

Example.asm
Assume CS:code, DS:data, SS:stk
; Простая программа на ассемблере
data segment    para public "data" ; Сегмент данных
data8    db          7,0dh,0ah,'Привет! ',0dh,0ah,'$'      ; Код
                                     ;консольного писка + сообщение
data16    dw          65535
data ends

stk segment    para stack "stack" ; Сегмент стека
    db          256 dup (?)      ; Буфер для стека

```

```
stk ends
```

```
code segment para public "code" ; Сегмент кода
Start:
    mov ax,data ; Адрес сегмента данных в регистр AX
    mov ds,ax ; Запись AX в DS
    mov ax,0 ; Обнуление аккумулятора AX
    inc ax ; Увеличение AX на 1
    mov bx,ax ; Пересылка <AX> в BX
    mov ch,data8 ; Пересылка переменной data8 в CH
    mov bp,data16 ; Пересылка переменной data16 в BP
    dec word ptr data16
    call mes
    mov ax,4C00h ; функция завершения программы
    int 21h ; функция Dos
mes proc near ; Процедуры вывода сообщения на экран
    lea dx,data8 ; Загрузка смещения data8 в DX
    mov ah,9 ; функция вывода строки на экран
    int 21h ; функция DOS
    ret
mes endp
code ends
END Start
```

В приведенном примере определены переменные – байтовая и словная, а также используются функции DOS – вывод сообщения на экран и функция завершения программы.

Для использования турбо дебаггера при отладке программы следует оттранслировать программу с включением отладочной информации. Для этого компиляцию программы следует производить с ключом **/zi (tasm /zi example.asm)**, а затем скомпоновать программу с ключом **/v (tlink /v example.obj)**. Следует отметить, что ключи должны задаваться строчными символами.

Запуск программы под турбо дебаггером

Для запуска программы в турбо дебаггере следует набрать **td.exe example.exe**. Окно запущенного турбо дебаггера с отлаживаемой программой показано на рисунке 4.31.

Турбо дебаггер позиционирует курсор на начале отлаживаемой программы (на первой выполняемой ее строке).

На рисунке 4.31 видны оперативные клавиши (нижняя строка экрана): F1=Help – справка; F2=Bkpt – точка останова; F3=Close – закрыть; F4=Here – здесь; F5=Zoom – переключение окон; F6=Next – дальше; F7=Trace – трассировка; F8=Step – шаг; F9=Run – выполнить; F10=Menu – меню.

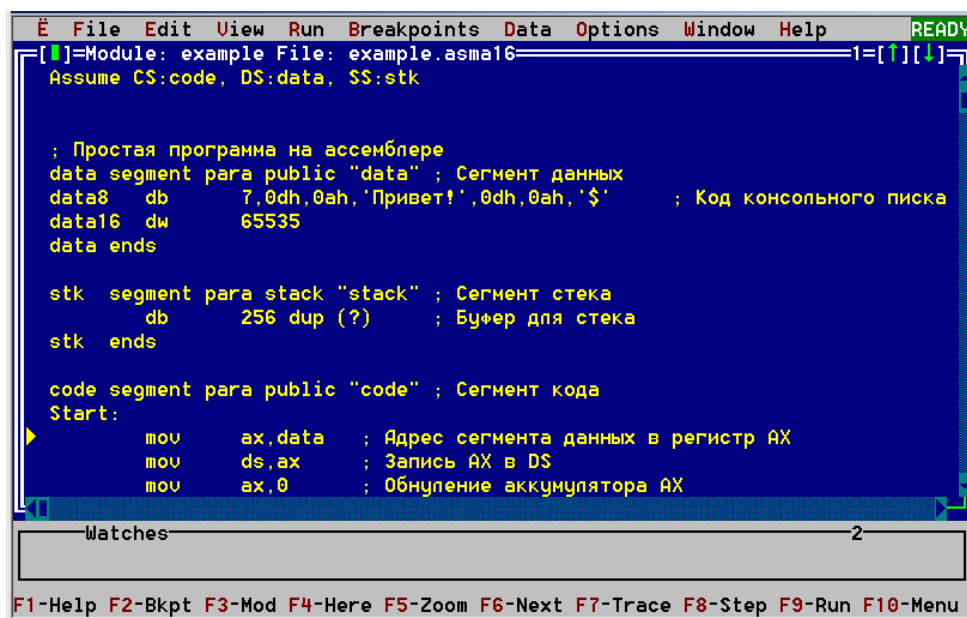


Рисунок 4.31 – Начальный экран при работе с отлаживаемой программой

На рисунке видно также основное меню турбо дебаггера (верхняя строка экрана): Ё – системное меню; File – файл; Edit – редактирование; View – обзор; Run – выполнение; Breakpoints – точки останова; Data – данные; Options – параметры; Window – окно; Help – справка.

Данный экран состоит из основной строки меню, окон Module (Модуль) и Watches (Просмотр) и справочной строки.

Завершение работы

Чтобы выйти из отладчика в любой момент и вернуться в операционную среду DOS, нажмите клавиши Alt-X. Если необходимо вернуться в начальное состояние программы, нажатие клавиш Ctrl-F2 в любой момент позволит вам перезагрузить программу и начать выполнение сначала.

Однако при нажатии данных клавиш не сбрасываются точки останова или просматриваемые величины. Чтобы сделать это, нужно воспользоваться клавишами Alt-F O (клавиши Alt-B D также позволяют удалить все точки останова, но иногда быстрее перезагрузиться с помощью Alt-F O).

Получение подсказки

Когда необходимо получить справочную информацию о текущем окне, для этого нажмите клавишу F1. Вы можете получить разнообразную информацию (на английском языке), проходя по системе меню и нажимая клавишу F1 (будет выводиться краткий перечень того, что делает каждая команда).

Отладка примера программы на языке Ассемблера

Закрашенная стрелка (?) в левом столбце окна Module показывает, где турбо дебаггер остановил выполнение вашей программы.

Если программа еще не была запущена, стрелка находится на первой выполняемой ее строке (на рисунке 4.31 – это 16-я строка).

Для трассировки одной исходной строки программы следует нажать клавишу F7. Стрелка и курсор переместятся при этом на следующую выполняемую строку.

Если посмотреть на правую границу заголовка окна Module (Модуль) на рисунке 4.31, можно увидеть, что там указан номер строки, в которой находится курсор. Если переместить курсор с помощью клавиш управления курсором (стрелки) вверх и вниз, номера строки в заголовке будет меняться.

Для позиционирования курсора на строку в окне Module, можно нажать клавиши Ctrl-L, ввести номер строки и нажать Enter.

Как можно видеть, войдя в меню Run (Выполнение), существует несколько способов запуска программы на выполнение. Предположим, к примеру, что вы хотите выполнить программу до строки 24.

Чтобы запустить программу на выполнение, пока она не достигнет строки 24, переместите курсор на эту строку, а затем нажмите клавишу F4. Теперь, когда курсор находится на строке 24, нажмите клавишу F7 для выполнения еще одной строки исходного кода. Поскольку выполняемая вами строка представляет собой вызов другой функции, то стрелка теперь позиционируется на первой строке функции **mes** (строка 28 исходного текста). Курсор немедленно переходит к строке 28, где находится начало функции **mes**.

Продолжайте нажимать клавишу F7, пока не будет выполнена процедура **mes** и не произойдет возврат на строку, следующую за вызовом (строка 25).

Если на строке 24 вместо клавиши F7 нажать клавишу F8, то вместо перехода в функцию произойдет переход сразу к строке 25.

Клавиша F8 аналогична клавише F7, которая выполняет процедуру, но она не выполняет по шагам код процедуры.

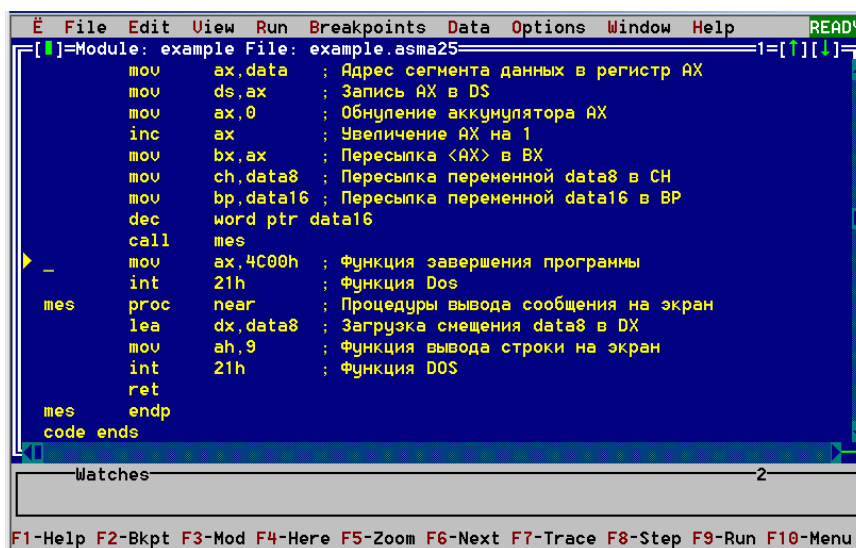


Рисунок 4.32 Программа остановилась после возврата из процедуры **mes**

Чтобы выполнить программу до тех пор, пока не будет достигнуто заданное место, вы можете непосредственно указать функцию или номер строки, не перемещая на данную строку курсор, а затем выполнить

программу до этой точки. Чтобы задать метку, до которой вы хотите выполнить программу, нажмите клавиши Alt-F9. Выведется диалоговое окно. Введите **mes** и нажмите клавишу Enter. Программа начнет выполнение и остановится в начале процедуры **mes** (строка 28).

Задание точек останова

Другой способ управлять остановкой программы состоит в использовании точек останова. Простейший способ задать точки останова заключается в использовании клавиши F2. Переместите курсор на строку 25 и нажмите клавишу F2. турбо дебаггер подсвечивает строку, показывая, что на ней установлена точка останова (рисунок 4.33).

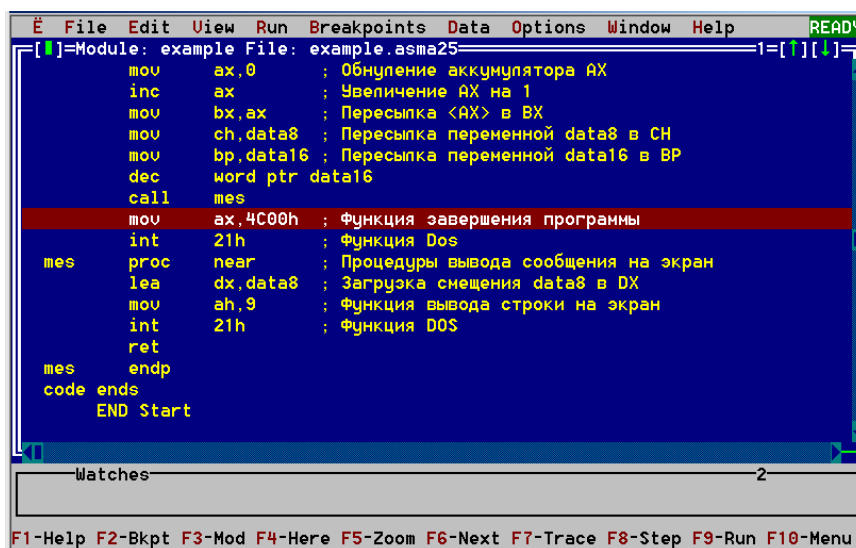


Рисунок 4.33 – Подсвеченная точка останова на строке 25

Для переключения (установки и отмены) точек останова в программе можно также использовать "мышь". Для этого надо щелкнуть кнопкой "мыши", находясь в первых двух позициях окна Module (Модуль).

Оперативные клавиши: F1=Help – справка; F2=Bkpt – точка останова; F3=Close – закрыть; F4=Here – здесь; F5=Zoom – переключение окон; F6=Next – дальше; F7=Trace – трассировка; F8=Step – шаг; F9=Run – выполнить; F10=Menu – меню.

Для иллюстрации процесса отладки программы следует сбросить программу в исходное состояние. Для этого надо (Главное меню→Run→Programm reset) или клавиши нажать Ctrl-F2.

Теперь надо запустить программу. Для этого следует выбрать (Главное меню→Run→Run) или нажать клавишу F9 для выполнения программы без прерывания (до первой точки останова, которая у нас установлена на строке 25). При выполнении процедуры **mes** на экран выводится «Привет!», однако это сообщение не видно, так как экран занят окном турбо дебаггера. Для того, чтобы посмотреть, что же получилось на экране пользовательской программы, необходимо переключиться в режим экрана пользователя (Главное меню→Windows→User Screen) (рисунок 4.34). Того же эффекта можно достичь, нажав клавиши Alt-F5. В окне турбо дебаггера появится окно программы пользователя с сообщением «Привет!». Для выхода из окна программы пользователя следует нажать Esc. На дисплее вновь появится экран турбо дебаггера, а стрелка будет позиционирована на строке 25 (рисунок 4.35), где установлена точка останова, и прекратила свое выполнение программа.

Если теперь опять нажать клавишу F9, программа будет выполнена до конца, так как больше точек останова нет. По завершении программы турбо дебаггер выводит сообщение о завершении программы и о том, что код возврата – (это означает, что программа завершилась без ошибок). Экран турбо дебаггера с сообщением о завершении программы пользователя показан на рисунке 4.36. Это сообщение следует квити́ровать нажатием клавиши Enter.

Использование окна Watches

В окне Watches (Просмотр) в нижней части экрана показываются значения выбранных переменных. Например, чтобы увидеть значение переменной data8, следует переместить курсор на имя переменной на

строке 6 и выбрать команду Watch (Просмотр) из локального меню окна Module. Можно использовать также сокращение этой команды.

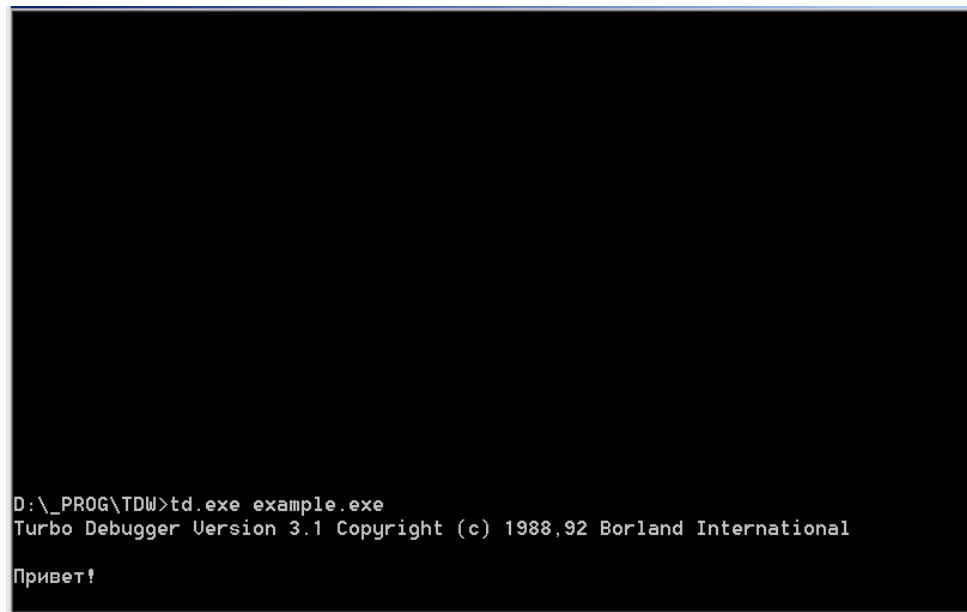


Рисунок 4.34 – Результат выполнения процедуры **mes** на экране пользовательской программы

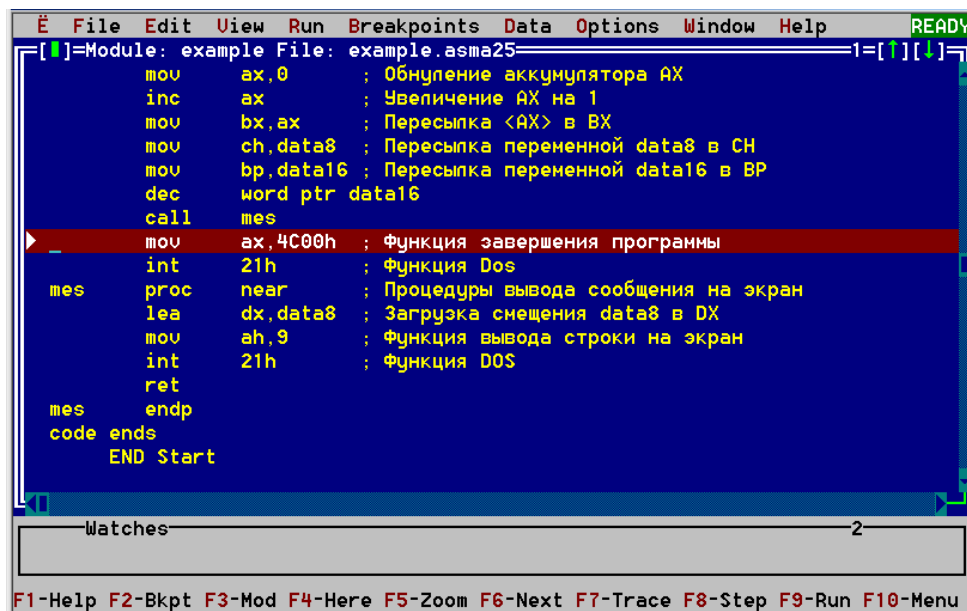


Рисунок 4.35 – Возврат в окно турбо дебаггера после просмотра окна пользовательской программы

Для этого надо нажать клавиши Alt-F10 для выбора локального меню, затем нажать клавишу W. Для выбора второй переменной – data16, следует выполнить те же манипуляции, поместив предварительно курсор на строку 7.

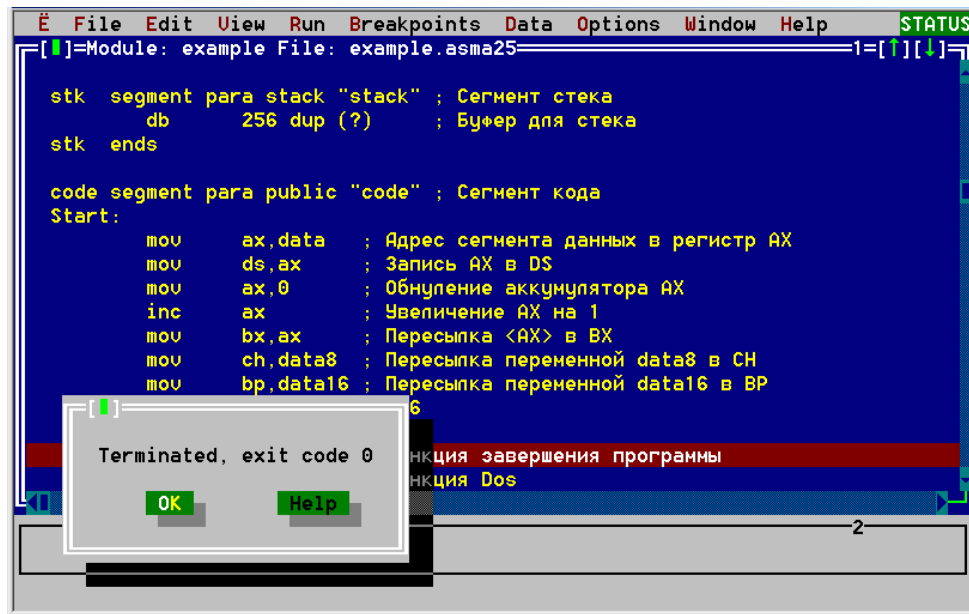


Рисунок 4.36 – Сообщение турбо дебаггера о завершении программы

Теперь переменные data8 и data16 в окне Watches (Просмотр) в нижней части экрана, где указаны также их тип (byte для data8 и word для data16) и значение (рисунок 4.37). По мере выполнения программы турбо дебаггер изменяет эти значения и отражает их текущие значения.

На рисунке 4.37 видно, что значение переменной data16 – 65535 (0FFFFh). Для того, чтобы посмотреть, как работает окно Watches, поставим точку останова на строке 23 (поместит курсор на строку 23 и нажать клавишу F2). После этого запустим программу (клавиша F9), при этом программа выполнится до точки останова (строки 23). Для того, чтобы выполнить команду на строке 23 (декремент переменной data16), нажмем клавишу F7. При этом будет выполнена команда на строке 23 (значение переменной data16 уменьшится на 1) и управление перейдет на следующую строку (рисунок 4.38).

На рисунке 4.38 видно, что значение переменной data16 в окне Watches уменьшилось на единицу – стало 65534 (0FFFEh).

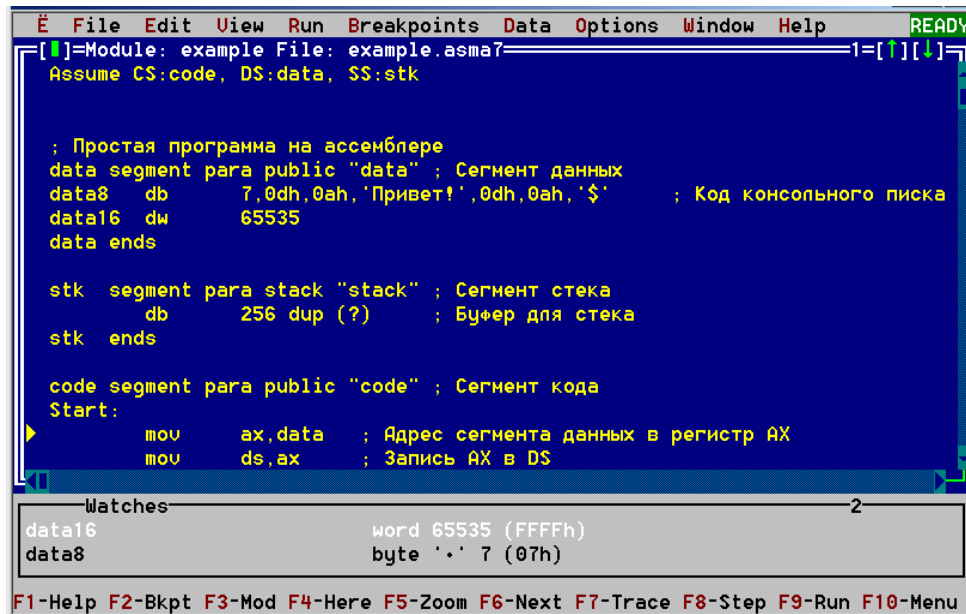


Рисунок 4.37 – Переменные в окне Watches

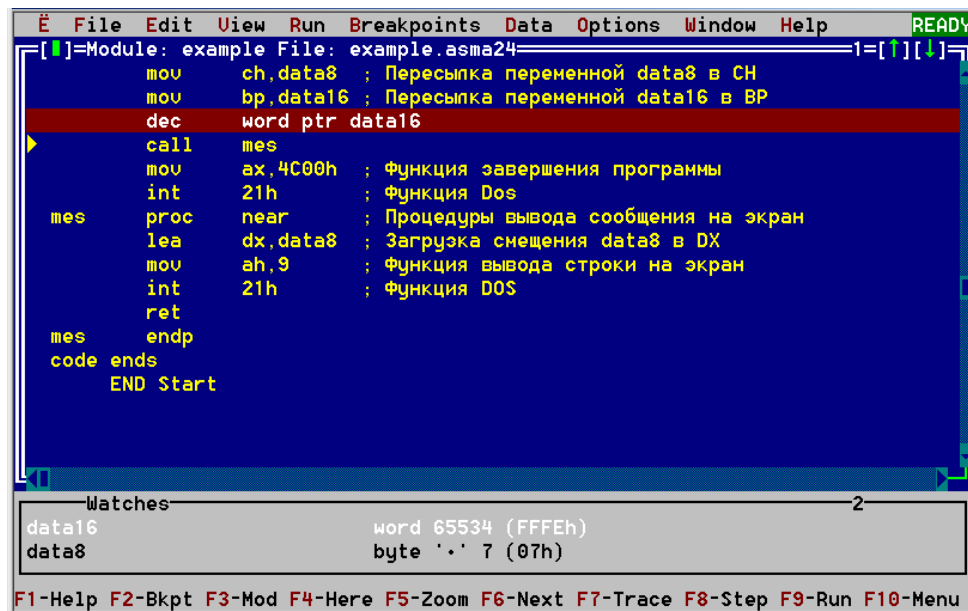


Рисунок 4.38 – Индикация выполнения команды *dec data16*

Анализ переменных можно осуществлять также при помощи окна Inspect (Проверка). Когда программа остановлена (как в нашем случае), существует много способов просмотра данных с помощью команды Inspect (Проверка). Это очень мощное средство позволяет вам анализировать

данные таким же образом, как если бы вы визуально наблюдали их при разработке программы.

Команды Inspect (в различных локальных меню и в меню Data) позволяют вам наблюдать за любой заданной переменной. Предположим, вы хотите взглянуть на значение переменной data16 после того, как программа была выполнена до команды в строке 16 включительно (то есть, был выполнен декремент этой переменной). Для этого следует поставить точку останова на строку 17 и выполнить программу (выбрать подпункт Run из основного меню Run, или просто нажать клавишу F9). При этом будут выполнены все команды до строки 16 включительно, и выполнение программы остановится на строке 17. Для того, чтобы посмотреть интересующую нас переменную data16, следует поместить курсор под этой переменной, вызвать локальное меню (Alt-F10) и выбрать такм первый пункт (Inspect). После этого на экране появится окно Inspector (Проверка) (рисунок 4.39).

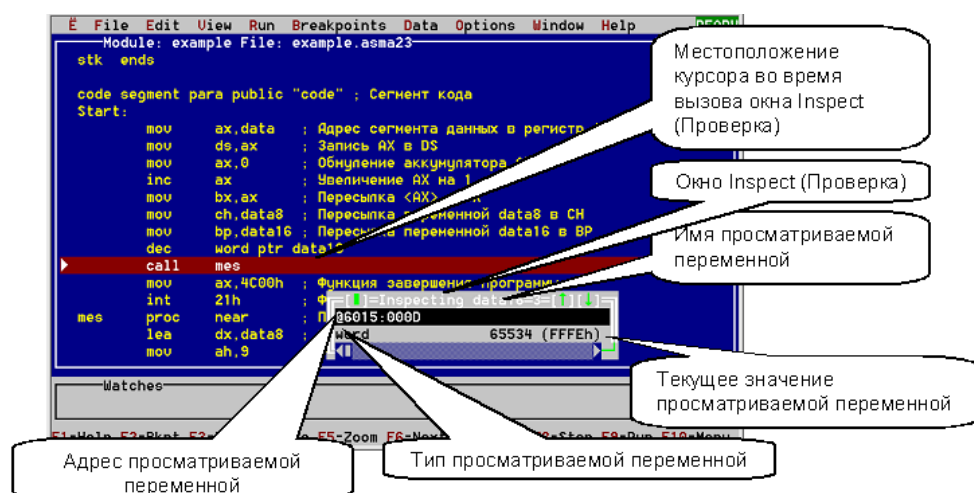


Рисунок 4.39 – Окно Inspector (Проверка)

В первой строке (заголовке) этого окна вам сообщается адрес данной переменной в памяти. Третья строка показывает, какой тип данных хранится в переменной data16 (это тип word). Справа указано текущее

значение переменной. Теперь, проверив значение этой переменной, нажмите клавишу Esc для того, чтобы закрыть окно Inspector. Для этого (как и во всех других окнах) можно также воспользоваться клавишами Alt-F3, либо закрыть окно с помощью "мыши".

Просмотреть переменную можно и более простым способом. Для этого надо нажать Ctrl-I. При этом откроется окно для ввода имени просматриваемой переменной. В имеющееся поле ввода следует ввести имя переменной – data16, и нажать кнопку ОК (рисунок 4.40).

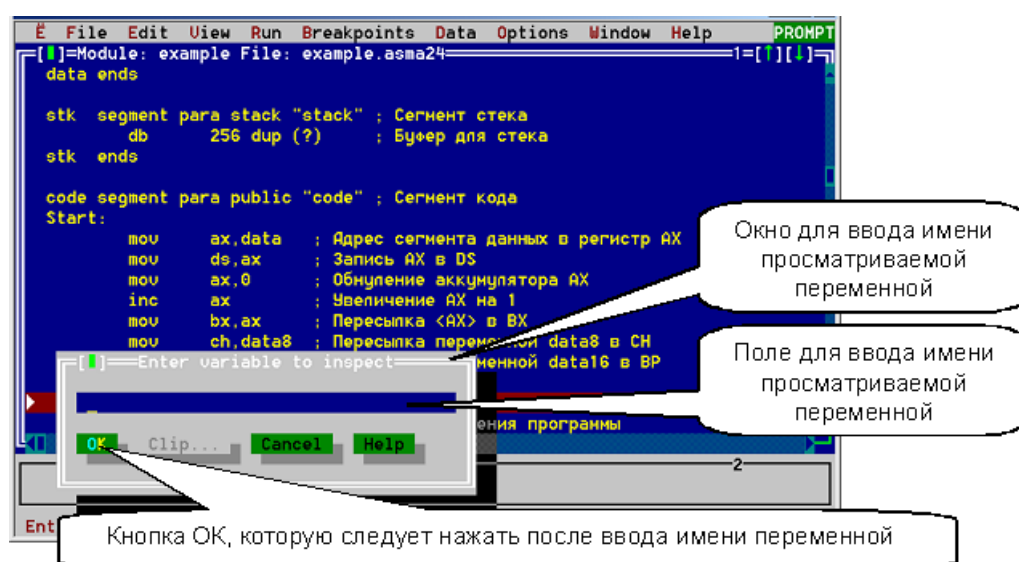


Рисунок 4.40 – Окно для ввода имени просматриваемой переменной

После нажатия кнопки ОК, на экране появится окно, изображенное на рисунке 4.39.

Таким образом можно просматривать значения любых переменных на любом этапе выполнения программы.

Изменение значений переменных в программе

Теперь попробуем изменить значение переменной. С помощью клавиш со стрелками перейдите на строку 22 исходного файла. Поместите курсор на переменную data16, и для проверки ее значения нажмите клавиши Ctrl-I. После того, как будет открыто окно Inspector, нажмите для вывода локального меню окна Inspector клавиши Alt-F10 и выберите

команду Change (Изменить) (рисунок 4.41). (Это можно сделать также непосредственно, нажав клавиши Ctrl-C.) Появляется подсказка (диалоговое окно), запрашивающая новое значение (рисунок 4.42).

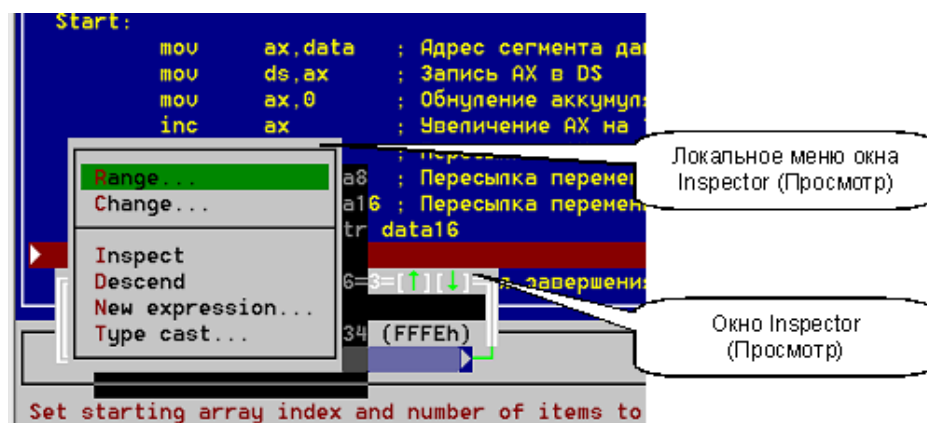


Рисунок 4.41 – Локальное меню окна Inspector (Проверка) для выбора команды Change (Изменить)

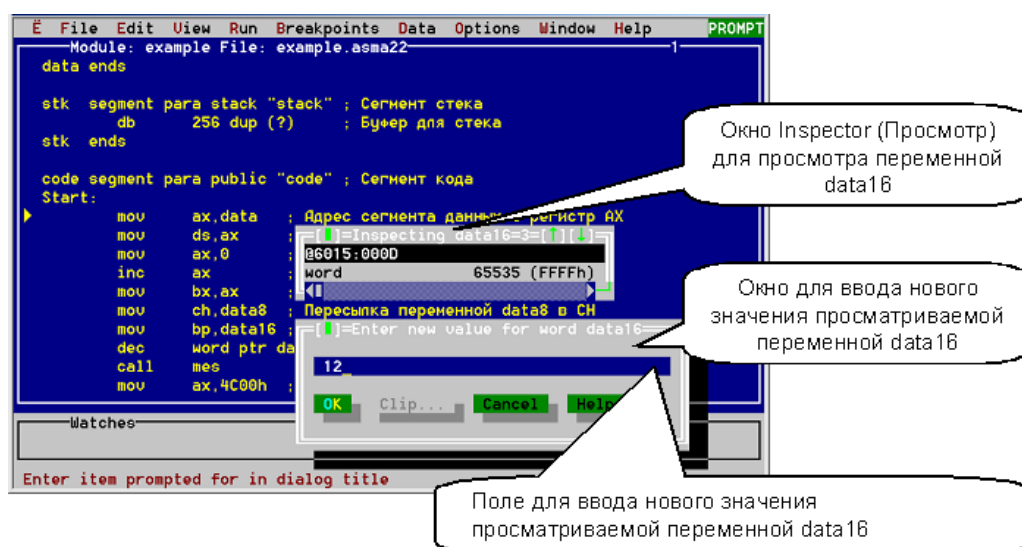


Рисунок 4.42 – Диалоговое окно для ввода нового значения переменной

В заголовке диалогового окна сообщается: *Enter new value for word data16* (введите новое значение для переменной *data16* типа *word*).

В это поле можно ввести любое вычисляемое выражение языка Ассемблер, при вычислении которого получается число. Если, например, набрать *data8+5* и нажмете клавишу Enter, в окне проверки теперь будет

показано новое значение – 65528 (FFF8h), как это показано на рисунке 4.43.

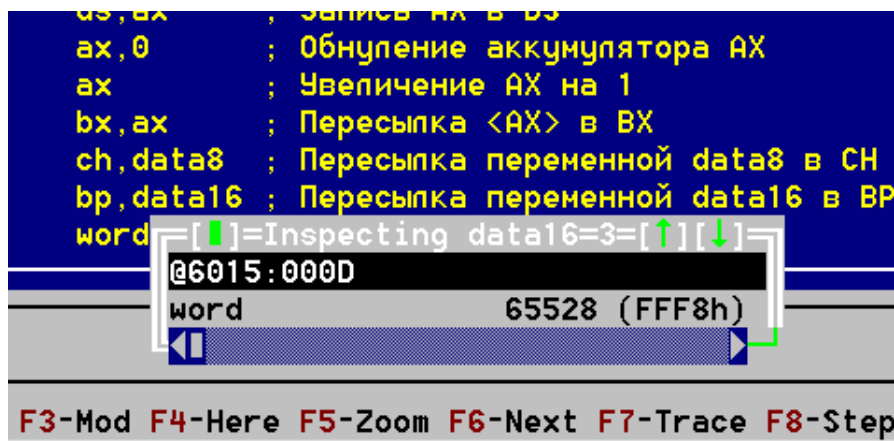


Рисунок 4.43 – Индикация нового значения переменной в окне Inspector
(Проверка)

Для изменения элемента данных, который не виден в текущем окне Module (Модуль), следует выбрать команду Data/Evaluate/Modify (Данные/Вычисление/Модификация). При этом будет выведено окно подсказки, в которой вы можете ввести имя изменяемой переменной, как это было показано на рисунке 4.40.

4.5 Процессорные средства тестирования программ

Программные средства отладки

Процессоры x86 имеют отладочные средства, предназначенные как для программного использования, так и для тестирования с помощью внешнего оборудования. Особенности и состав этих средств различается для разных моделей процессоров.

Внутренние средства отладки предназначены для облегчения выполнения отладочных процедур. Они подразделяются на 3 типа:

- однобайтная команда прерывания *INT3* (код *0CCh*);
- пошаговый режим работы (управляемый флагом ловушки *TF*);
- останова по командам и данным, задаваемые с помощью регистров отладки процессора.

Команда прерывания

Команда *INT3* используется программными отладчиками (в том числе и турбо дебаггером, описанным выше). Выполнение этой команды вызывает исключение 3 (прерывание отладки).

В отличие от других команд прерывания, имеющих двухбайтный формат, команда *INT3* однобайтная. Это делает ее удобной для использования в программных отладчиках при установке точек прерывания (например, путем подмены первого байта любой команды). Процессор, встречая в программе команду с кодом *0CCh*, вызывает программу обработки прерывания с вектором 3, которая и используется для связи с отладчиком.

Кроме того, данная команда нечувствительна к значению привилегии *IOPL* как в защищенном режиме, так и в режиме виртуального процессора *V86*.

Пошаговый режим

В пошаговом режиме выполнение программы осуществляется по одной команде. После выполнения каждой команды вызывается исключение 1 (исключение отладки).

Пошаговый режим задается установкой в единицу флага *TF* (Trap Flag – флаг ловушки).

Регистры отладки

Отладочные регистры появились в архитектуре микропроцессоров семейства x86, начиная с процессора Intel386. Эти регистры позволяют выставлять точки останова и перехватывать обращения процессора к памяти. В процессорах Pentium и выше можно останавливаться и по обращениям ввода-вывода.

(В свое время, в ЭВМ первого-второго и даже третьего поколений такие точки останова можно было задать на переключателях пульта оператора или системного инженера).

Отладочных регистров у процессора восемь: *DR0...DR7*.

Первые четыре регистра: *DR0...DR3* используются для задания до четырех 32-разрядных адресов точек останова. Заданный адрес указывает байт, слово или двойное слово, попадание в который(ое) вызывает срабатывание ловушки останова. Что именно указывает адрес определяется полем *LEN_i* ($i = 0...3$) в регистре *DB7*. В этом же регистре, но только в поле *RW_i*, задается тип перехватываемого обращения к памяти:

00 – выборка команды из памяти

01 – запись данных в память

10 – обращение к портам ввода-вывода (только для Pentium и выше при включении расширения отладки: бит *DE* регистра *CR4*)

11 – чтение или запись данных памяти.

Генерируемые при этом исключения различаются по типу. При выборке команды исключение классифицируется как отказ (fault) и обрабатывается до выполнения (этой выбираемой из памяти) команды. В

случае обращения к данным исключение рассматривается как ловушка (trap) и обрабатывается после передачи (читаемых или записываемых) данных.

31																													0			
Линейный адрес точки останова 0																													DR0			
Линейный адрес точки останова 1																													DR1			
Линейный адрес точки останова 2																													DR2			
Линейный адрес точки останова 3																													DR3			
Зарезервировано																													DR4			
Зарезервировано																													DR5			
01																BT	BS	BD	0	01	01	01	01	01	01	01	01	B3	B2	B1	B0	DR6
LEN3	R3	W3	LEN2	R2	W2	LEN1	R1	W1	LEN0	R0	W0	0	0	GD	0	0	01	GE	LE	G3	L3	G2	L2	G1	L1	G0	L0	DR7				
31																	16	15											0			
<div><div>01</div><div>0 – для i386, i486 1 – для Pentium</div></div>																																

Рисунок 4.44 – Регистры отладки

Для управления установкой отладочных точек используются два младших байта регистра *DR7*, биты которого имеют следующее назначение:

- бит *GD* (Global Debug Register Access Detect – обнаружение доступа к регистрам отладки), доступный только в реальном режиме или в защищенном режиме на уровне привилегии *CPL* = 0, позволяет отслеживать любые обращения к отладочным регистрам. При *GD* = 1 любая попытка обращения вызовет исключение 1 (отказ);
- биты *GE* и *LE* (Global и Local Exact data breakpoint match – глобальная и локальная точка немедленного останова по совпадению данных) определяют, будет ли исключение генерироваться сразу после завершения операции обмена при включенной ловушке на обращение к данным или оно произойдет несколько позже (возможно, никогда). Ловушка на обращения за командами всегда срабатывает немедленно. Бит *LE* автоматически сбрасывается при переключении задач, бит *GE* не изменяет своего состояния при таких переключениях;

- биты Gi и Li (Global и Local breakpoint enable – разрешение глобальной и локальной точек прерывания) разрешают срабатывание ловушек по отладочным точкам. Биты Li автоматически сбрасываются при переключении задач, биты Gi не изменяют своего состояния при таких переключениях. Автоматический сброс битов Li блокирует лишние срабатывания отладочных точек при переключениях задач.

Для упрощения определения отладчиком причины, вызвавшей срабатывание отладочной точки (исключение 1), могут использоваться биты регистра состояния отладки $DR6$ (Debug Status Register), идентифицирующие эти причины:

- биты Bi – срабатывание точки останова по регистру DRi ,
- бит BS – ловушка пошагового режима,
- бит BT – ловушка переключения задач (по биту T в TSS),
- бит BD – отказ при попытке доступа к регистрам отладки при $GD = 1$.

Значения флагов Bi действительны только для контрольных точек с установленными битами Li и/или Gi .

Генерация исключений по контрольным точкам может быть отключена флагом RF регистра флагов процессора.

Аппаратные средства отладки

Начиная с ряда моделей i486, в процессоры стали включать тестовый интерфейс JTAG.

Стандарт IEEE 1149.1 Boundary Scan Architecture (архитектура сканирования границ), или интерфейс JTAG, разработан для тестирования сложных логических схем, установленных в целевое устройство. В данном случае тестированию подлежит внутренняя логика процессора.

Интерфейс имеет всего 4 сигнала:

- *TMS* (Test Mode Select – Выбор режима тестирования) – сигнал выбора тестового режима,
- *TDI* (Test Data Input – Входные данные тестирования) – входные данные в последовательном двоичном коде,
- *TDO* (Test Data Output) – выходные данные в последовательном двоичном коде,
- *TCK* (Test Clock – Синхроимпульсы тестирования) – сигнал синхронизации последовательных данных.

Эти сигналы образуют тестовый порт *TAP* (Test Access Port), через который тестируемое устройство подключается к тестирующему оборудованию. В задачу этого оборудования входит формирование последовательностей тестовых сигналов по программе тестирования, определенной разработчиком тестируемого устройства, и сравнение полученных результатов с эталонными. (Такие последовательности, как входные, так и выходные, иногда называют сигнатурами).

Работой порта управляет *TAP*-контроллер, использующий несколько специальных регистров порта.

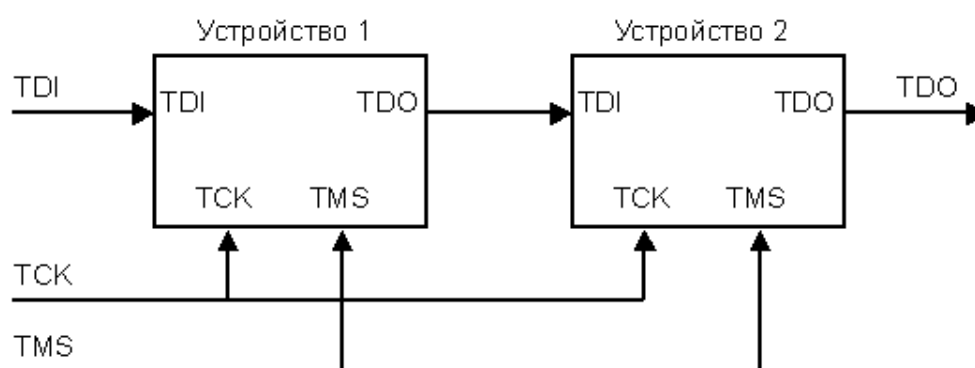


Рисунок 4.45 – Цепочка устройств с интерфейсом JTAG

Один и тот же контроллер и порт могут использоваться для тестирования любого числа устройств, поддерживающих интерфейс JTAG. Для этого они соединяются в цепочку по входам и выходам данных тестирования, как показано на рисунке 4.45. А стандартный логический

формат данных и сигналов управления позволяет контроллеру независимо общаться с каждым из устройств цепочки, при условии, конечно, исправности самих ячеек интерфейса JTAG.

Общая структурная схема организации поддержки интерфейса JTAG показана на рисунке 4.46. Тестируемая часть устройства (например, процессора) в этом случае имеет своих на входах и выходах (участвующих в тестировании) специальные ячейки BS Cell (Boundary Scan Cell – ячейки сканирования границ), содержащие в себе элементы памяти и переключатели.

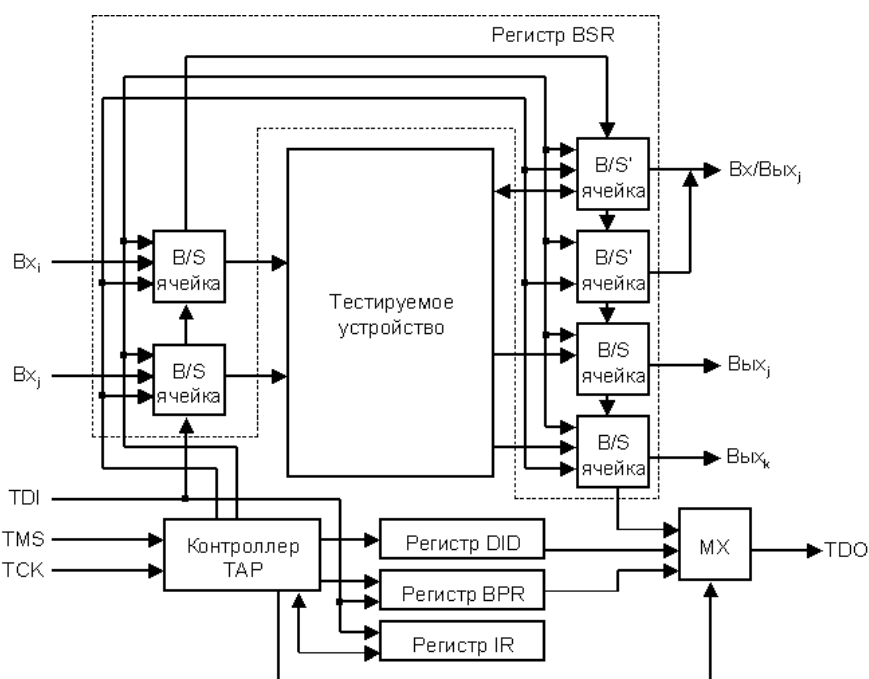


Рисунок 4.46 – Организация поддержки интерфейса JTAG

Это позволяет отключать входы тестируемой логики от внешних цепей и подавать на них заранее известные сигналы, установленные предварительно в элементах памяти ячеек, или фиксировать в этих элементах поступающие извне входные сигналы в требуемый момент времени.

Аналогично, можно управлять и выходами тестируемого устройства, отключая их от внешних цепей с одновременной подачей в них сигналов

из триггеров BS ячеек, или фиксировать в триггерах выходные сигналы тестируемого устройства в требуемый момент времени.

(В случае двунаправленных шин BS ячейка должна предварительно настраиваться на ввод или вывод, для чего для каждой такой шины имеется по дополнительной ячейке. На рисунке 4.46 такая ячейка обозначена как B/S'.)

Ячейки BS объединены в регистр *BSR*, обведенный на рисунке 4.46 пунктиром.

Кроме регистра *BSR* в схеме имеются регистры:

- *DID* – регистр идентификации устройства (Device Identification Register), имеющий 32-битную разрядность и содержащий в себе код производителя и номер версии, по которым контроллер может определить тип устройства,
- *BPR* – регистр транзита (Bypass Register), имеющий всего один бит и использующийся для транзитной передачи входных данных *TDI* через устройство, если оно не должно получать эти данные при объединении нескольких устройств в цепочку,
- *IR* – регистр команд (Instruction Register) – четырехразрядный регистр, использующийся для получения и хранения команд для контроллера, определяющих выполняемый тест.

Все эти регистры представляют собой независимые сдвигающие регистры, соединенные параллельно. На их входы поступают сигналы *TDI*, а с выходов – снимаются сигналы *TDO*. По каждому синхросигналу *TCK* данные сдвигаются на один бит (при соответствующем разрешающем сигнале контроллера). Данные на вход регистра *DID* не поступают.

Управляет работой регистров контроллер, представляющий собой синхронный конечный автомат, изменяющий свое состояние по сигналу *TCK*. Граф переходов этого автомата можно найти в [3] или в [5]. Основные выполняемые им команды:

- *EXTEST* – предназначенная для проверки внешних соединений тестируемого устройства,
- *SAMPLE* – позволяющая зафиксировать состояния всех внешних сигналов или загрузить данные в ячейки регистра *BSR*,
- *IDCODE* – подключает к интерфейсу регистр *DID* для считывания содержимого регистра идентификации,
- *BYPASS* – предназначенная для подключения однобитного обходного регистра *BPR* и блокировки работы тестируемого устройства в режиме контроля,
- *RUNBIST* – запускающая встроенный тест процессора *BIST* (Build In Self Test, отдельный тест процессора, не увязываемый жестко с интерфейсом JTAG).

Контроллер TAP переводится в исходное состояние при включении питания или при удержании высокого уровня сигнала *TMS* не менее пяти тактов *TCK*.

Как уже отмечалось, при включенном тестовом режиме контроллер может логически отсоединять входы и выходы устройства от внешних выводов, задавать входные воздействия и считывать результаты с выходов, что и необходимо для тестирования как комбинационных, так и последовательностных схем. Причем делается все это с помощью только четырех названных выше сигналов.

Для интерфейса JTAG существует специальный язык описания устройств BSDL (Boundary Scan Description Language). Состав и порядок следования информационных и управляющих ячеек в регистре *BSR* специфичен для каждого устройства (для его определения и используется код идентификации устройства из регистра *DID*).

В процессорах, начиная с Pentium, имеется также дополнительный сигнал прерывания *R/S#*, по которому процессор переходит в зондовый

режим отладки. В этом режиме с помощью дополнительных команд контроллера TAP возможен доступ к регистрам процессора.

4.6 Вопросы для самопроверки

1. Что такое Турбо дебаггер?
2. Для чего используется Турбо дебаггер?
3. Позволяет ли Турбо дебаггер вычислять выражения языков Си, Паскаль и Ассемблер?
4. Можно ли настраивать выводимую в Турбо дебаггере на экран информацию?
5. Что такое точка останова?
6. В чем заключается процесс отладки программы?
7. Что такое «обнаружение ошибки»?
8. Как ищется место возникновения ошибки?
9. Как определяется причина возникновения ошибки?
10. Как можно исправить ошибку?
11. Что такое трассировка программы?
12. Что такое обратная трассировка программы?
13. Что такое пошаговое выполнение программы?
14. Что такое просмотр переменных, точек останова и содержимого стека?
15. Чего турбо дебаггер не может сделать?
16. Из чего состоит главное меню Турбо дебаггера?
17. Что такое выпадающее меню?
18. Что можно сделать с помощью выпадающего меню?
19. Каким образом можно выбрать пункт главного меню?
20. Какими клавишами можно осуществить перемещение по меню Турбо дебаггера?
21. Можно ли использовать указатель мыши для работы с различными меню Турбо дебаггера?

22. Как можно выйти из меню?
23. Как можно вернуться в меню предыдущего уровня?
24. Как установить точку останова в программе?
25. Как можно выполнить одну исходную строку или команду программы?
26. Как можно выполнить одну исходную строку или инструкцию, пропуская вызовы процедур?
27. Как можно запустить программу на выполнение?
28. Что позволяет сделать локальное меню окна Breakpoints?
29. Что позволяет сделать локальное меню окна CPU?
30. Из каких областей состоит окно CPU?
31. Что отображается в области кода окна CPU?
32. Что отображается в области данных окна CPU?
33. Что отображается в области флагов окна CPU?
34. Что отображается в области регистров окна CPU?
35. Что отображается в области стека окна CPU?
36. Что позволяет сделать локальное меню окна File?
37. Что позволяет сделать локальное меню окна Log?
38. Что позволяет сделать локальное меню окна Module?
39. Что позволяет сделать локальное меню окна Clipboard?
40. Что позволяет сделать локальное меню окна Numeric Processor?
41. Что позволяет сделать локальное меню окна Variables?
42. Что позволяет сделать локальное меню окна Watches?
43. Как выйти из Турбо дебаггера?
44. Как получить подсказку по элементу окна?
45. Как индицируется место останова программы в Турбо дебаггере?
46. Как использовать окно Watches для наблюдения переменных?
47. Как в Турбо дебаггере посмотреть на экране результат выполнения отлаживаемой программы?
48. Как вернуться из окна просмотра результата выполнения программы?

- 49.Как узнать код возврата, с которым завершилась отлаживаемая программа?
- 50.Для чего можно использовать команду Inspect в Турбо дебаггере?
- 51.Как можно изменить значение переменной в отлаживаемой программе?
- 52.Какие имеются аппаратные средства отладки в микропроцессорах семейства x86?
- 53.Что такое интерфейс JTAG?
- 54.Какие сигналы используются в интерфейсе JTAG?
- 55.Как включается цепочка отлаживаемых устройств по интерфейсу JTAG?
- 56.Как используются двунаправленные шины в интерфейсе JTAG?
- 57.Что такое BS ячейка в интерфейсе JTAG?
- 58. Как описываются устройства для интерфейса JTAG?
- 59.Что такое тестовый порт TAP?
- 60.Какие команды может выполнять контроллер TAP?

Литература

1. Кулаков В. Программирование на аппаратном уровне. Специальный справочник. – СПб: Питер, 2001. – 496 с.: ил.
2. Thomas Roden. Four Gigabytes in Real Mode. – Programmer's Journal 7.6, 1989.
3. Intel Architecture Software Developer's Manual, Volume1: Basic Architecture. – Intel Corp., 1999.
4. Intel Architecture Software Developer's Manual, Volume1: Instruction Set Reference Architecture. – Intel Corp., 1999.
5. Intel Architecture Software Developer's Manual, Volume1: System Programming. – Intel Corp., 1999.
6. Гук М. Процессоры Intel: от 8086 до Pentium II. – СПб: Питер, 1997. – 224 с.: ил.
7. Гук М. Процессоры Pentium II, Pentium Pro и просто Pentium. – СПб: Питер Ком, 1999. – 288 с.: ил.
8. Рудаков П.И., Финогенов К.Г. Програмируем на языке ассемблера IBM PC. Изд. 3-е. – Обнинск: Изд-во «Принтер», 1999, – 495 с.: ил.

Учебное издание

Рощин Алексей Васильевич

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Особенности программирования 32-разрядных
процессоров

Учебное пособие

ЛР № 020418 от 08 октября 1997 г.

Подписано в печать

Формат 60x84 1/16

Объем 5,5 п.л. Тираж 100 экз. Заказ №

ГОУ ВПО “Московский государственный университет

приборостроения и информатики”

107996, Москва, ул. Стромывнка, 20