

Защищённый режим.

Глава 1. Обзор режимов микропроцессоров IA-32.

Процессоры IA-32 могут работать в четырёх режимах: режиме реальных адресов, защищённом режиме, режиме виртуального процессора 8086 и режиме системного управления. Теперь более подробно об этих режимах.

1. Режим реальных адресов (real address mode). Также вы могли встречать термин "реальный режим", что должно обозначать то же самое, хотя, это немного не правильно и ещё одно название этого режима, встречающееся в документации Intel: R-Mode. Режим реальных адресов - это режим, в котором процессор работает как "эктишка" (8086), но позволяет пользоваться большинством своих технологий (MMX / SSE / SSE2, 32-разрядные регистры общего назначения, регистры управления и отладки и пр.). После аппаратного сброса процессор переходит в этот режим и начинает выполнять программную инициализацию из BIOS-а.

Вообще говоря, режим реальных адресов в современных процессорах предназначен для запуска компьютера и подразумевается, что операционная система будет работать в защищённом режиме (поэтому оптимизация по производительности для процессоров IA-32 производится для защищённого режима).

В режиме реальных адресов не доступны основные достоинства процессора - виртуальная память, мультизадачность, уровни привилегий, работа с кэшами, буферами TLB, буфером ветвлений и некоторыми другими технологиями, обеспечивающими высокую производительность.

2. Защищённый режим (protected mode или P-Mode). Как утверждает Intel, это "родной" (native) режим 32-разрядного процессора.

В защищённый режим процессор надо переводить специальными операциями над системными регистрами и войти в этот режим процессор может только из режима реальных адресов. При работе в защищённом режиме процессор контролирует практически все действия программ и позволяет разделить операционную систему, драйвера и прикладные программы разными уровнями привилегий. Благодаря этому ОС может ограничить области памяти, выделяемой программам, запретить или разрешить ввод/вывод по любым адресам, управлять прерываниями и многое другое. При попытке программы выйти за допустимый диапазон адресов памяти, выделенной ей, либо при обращении к "запрещённым" для неё портам процессор будет генерировать исключения - специальный тип прерываний. Грамотно оперируя исключениями, операционная система может контролировать действия программ, организовать систему виртуальной памяти, мультизадачность и другие программные технологии.

В защищённом режиме максимально доступны все ресурсы процессора. Например, R-Mode максимальный диапазон адресов памяти

ограничен одним мегабайтом, а в защищённом режиме он расширен до 4 Гб для процессоров 386 и 486 и 64 Гб для Pentium-ов.

3. Режим виртуального процессора 8086 (virtual-8086 mode) или V-Mode, или "виртуальный режим" (что, вообще говоря, не совсем правильно).

В V-Mode можно войти только из защищённого режима. В этом режиме процессор эмулирует работу 8086 процессора (1 Мб адресного пространства, "обычные" прерывания и пр.), но при этом процессор сохраняет все средства контроля, присущие защищённому режиму.

Обычно, V-Mode используется в операционных системах для запуска программ, рассчитанных на процессор 8086 (так называемая, "обратная совместимость ПО"). V-Mode реализуется как отдельная задача в мультизадачной среде и она может взаимодействовать с другими задачами, если, конечно же, ей позволит это операционная система.

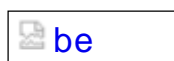
4. Режим системного управления (system management mode или S-Mode).

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Защищённый режим.

Глава 2. Уровни привилегий.

Защищённый режим (protected mode или P-Mode) обладает некоторыми особенностями, о которых нужно знать прежде, чем вы будете его использовать.

При работе в защищённом режиме процессор следит за правильным выполнением текущей программой ряда условий, например, программа не должна обращаться по определённым адресам портов ввода/вывода, запрещённых для неё. Если всё же происходит нарушение какого-либо условия, то процессор генерирует специальный тип прерывания, так называемое исключение, снабжает это прерывание информацией, описывающей где произошло нарушение и как оно произошло. Далее, специальная процедура обрабатывает это прерывание и решает, что дальше делать с "виноватой" программой (например, прекратить её выполнение).

Определением условий должна заниматься операционная система. Когда программа переводит процессор в защищённый режим, то ей, как говорится, "можно всё". Сразу после входа в защищённый режим процессор позволяет программе устанавливать свои условия для самой себя и для других программ. Для того, чтобы эти условия не могла переопределить другая программа, в процессоре введена система уровней привилегий. Благодаря этому, операционная система, например, может разрешить работу с дисковыми накопителями только для себя и тогда вирусы будут бессильны - они не смогут обратиться к дискам через порты ввода/вывода (разве что, только через "дыры" в самой операционной системе).

Основой защищённого режима являются уровни привилегий. Уровень привилегий - это степень использования ресурсов процессора. Всего таких уровней четыре и они имеют номера от 0 до 3. Уровень номер 0 - самый привилегированный. Когда программа работает на этом уровне привилегий, ей "можно всё". Уровень 1 - менее привилегированный и запреты, установленные на уровне 0 действуют для уровня 1. Уровень 2 - ещё менее привилегированный, а 3-й - имеет самый низкий приоритет. Таким образом, оптимальная схема работы программ по уровням привилегий будет следующая:

- уровень 0: ядро операционной системы,
- уровень 1: драйвера ОС,
- уровень 2: интерфейс ОС,
- уровень 3: прикладные программы.

Разумеется, это не единственный способ. Можно определить всю работу процессора в нулевом уровне - и ядро ОС, и драйвера и прикладные программы; можно, например, не разделять драйвера от ядра и т.п., но тогда

в такой системе не будет реализован встроенный в процессор механизм защиты программ и данных друг от друга и система будет неустойчивой.

Уровни привилегий 1, 2 и 3 подчиняются условиям, установленным на уровне 0, поэтому функционально эти четыре уровня можно разделить на 2 группы: уровень привилегий системы (0) и уровни пользователя (1, 2 и 3). На первый взгляд кажется, что проще было бы реализовать всего два уровня привилегий - системный и пользовательский, но со временем вы обнаружите, что четыре уровня привилегий - это очень удобно и гораздо лучше двух.

Программы и данные ограничены внутри своих уровней привилегий. Например, если программа работает на уровне привилегий 2, то она не сможет передать управление процедуре, работающей на любом другом уровне (0, 1 и 3), также, она не сможет обратиться к данным, определённым для использования на других уровнях привилегий. Процессор не допустит этого и в случае нарушений привилегий при доступе к данным и коду сгенерирует исключение общей защиты (general-protection exception) и передаст управление операционной системе, чтобы она приняла меры к нарушителю.

Сам по себе уровень привилегий ещё ничего не значит, его нельзя "установить в процессоре". Уровень привилегий применяется как одно из свойств при описании различных объектов, например, сегмента кода и действует при работе только с этим объектом.

Уровень привилегий обозначается как "PL" (от Privilege Level) и применяется в сочетаниях, например, IOPL - Input/Output Privilege Level - уровень привилегий ввода/вывода.

Когда процессор переходит в защищённый режим, то подразумевается, что программа будет работать в нулевом уровне привилегий.

[Следующая глава](#)


[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.

 [SpyLOG](#)

 [TopList](#)

 [be](#)

Защищённый режим:

Глава 3. Адресация памяти в защищённом режиме.

Первое, с чем сталкивается программа при переходе в защищённый режим - это совершенно другая система адресации памяти. Для начала, давайте вспомним, как это происходит в режиме реальных адресов - для обращения к памяти используется пара 16-разрядных регистров - сегментный регистр и смещение.

В сегментном регистре находится адрес сегмента. Сегмент - это область памяти размером в 64 Кб, которая должна начинаться на границе параграфа или, другими словами, на 16-байтной границе, то есть сегмент может начинаться только по адресу 0, 16, 32, 48 и т.д. Адресное пространство процессора 8086 равно одному мегабайту - это адреса в диапазоне от 00000h до FFFFFh. Т.к. сегмент начинается по адресу, кратному 16 (10h), то возможные адреса начала сегмента будут 00000h, 00010h, 00020h, ... , FFFF0h. Как видите, информация содержится в старших четырёх шестнадцатеричных разрядах - их и хранят в сегментном регистре. Другими словами, можно взять значение из сегментного регистра, умножить его на 16 (т.е. на 10h) и вы получите адрес начала сегмента.

Для указания конкретного адреса внутри сегмента используется второй 16-разрядный регистр, так называемое смещение. Использование пары регистров сегмент:смещение обеспечивает доступ ко всему мегабайту адресного пространства.

Каждый раз, перед тем как процессор обратится к памяти по адресу, указанному в паре регистров сегмент:смещение, он вычисляет адрес памяти по следующей схеме:

$$\text{физический_адрес} = \text{сегмент} * 10h + \text{смещение}.$$

Осталось заметить, что, вообще говоря, в режиме реальных адресов адресное пространство немного больше одного мегабайта, а именно:

$$1\text{Мб} + 64\text{Кб} - 16 \text{ байт}.$$

Такой адрес получается, если загрузить в пару регистров максимальные значения:

FFFF:FFFF

При этом адрес будет равен:

$$\text{FFFFh} * 10\text{H} + \text{FFFFh} = \text{FFFF0h} + \text{FFFFh} = 10\text{FFEFh} = 1114095$$

Теперь давайте рассмотрим схему образования адреса в защищённом режиме, как это происходит в процессоре i386 (80386 - это базовая модель для любого Pentium-a).

Адрес памяти в 32-разрядном процессоре является также 32-разрядным. Это значит, что адресное пространство для такого процессора равно 4 Гб (2^{32} байт).

Адресация памяти также производится через сегмент и смещение в сегменте, для чего используется пара регистров, но для описания сегмента используется больше информации, а именно:

- 32-разрядный адрес,
- 20-разрядный предел сегмента (предел = размер - 1),
- номер уровня привилегий сегмента,
- тип сегмента (код, стек, данные или системный объект)
- и дополнительные свойства, о которых подробнее будет сказано в других главах.

Как видите, информации об одном сегменте достаточно много. Для того, чтобы её хранить, используется не регистр, а специальная область памяти. Сегмент по-прежнему указывается в сегментном регистре, но теперь в нём хранится номер сегмента из списка определённых сегментов. Этот номер называется селектор.

В качестве смещения используется 16- или 32-разрядный регистр.

При обращении к памяти, процессор проверяет возможность доступа к сегменту по уровню привилегий, проверяет, не превысил ли адрес предел сегмента и можно ли обращаться к этому сегменту в данном случае (например, запрещена передача управления в сегмент, описывающий данные или стек). Если в результате проверки будет обнаружено нарушение какого-либо условия, то процессор сгенерирует исключение и тем самым обеспечит защиту.

Предел сегмента - это максимально допустимое смещение внутри него, таким образом, предел сегмента определяет его размер: $\text{размер_сегмента} = \text{предел_сегмента} + 1$

Обратите внимание, что значение предела - 20-разрядная величина, это значит, что максимальное значение предела равно $2^{20} - 1$.

Процессор измеряет размер сегмента двумя типами величин: либо байтами, либо страницами. Страница - это блок памяти размером в 4Кб.

В описании сегмента можно указать, в каких единицах измеряется сегмент и тогда можно получить два типа сегментов с максимальными размерами:

- в 1Мб (2^{20} байт) или

- в 4Гб (2^{20} страниц = $2^{20} * 4Кб = 2^{20} * 2^{12} = 2^{32}$ байт)

Эта способность измерять сегмент либо байтами, либо страницами, называется гранулярность.

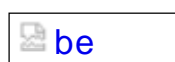
Значение предела сегмента может быть любым, от 0 до $2^{20} - 1$, гранулярность устанавливается по усмотрению программиста и может быть либо байтная, либо страничная. Всё это позволяет определять сегменты любого размера - от 0 байт до 4Гб.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Защищённый режим:

Глава 4. Дескриптор.

Прежде чем программа сможет обратиться по какому-либо адресу памяти, она должна определить набор сегментов, через которые она сможет получить доступ к памяти.

Сегмент определяется в виде структуры данных, которая называется дескриптор. Размер дескриптора - 8 байт, все дескрипторы хранятся последовательно в специально отведённой области памяти - глобальной дескрипторной таблице.

Далее приведен формат дескриптора:

биты:

0..15: предел, биты 0..15
16..31: адрес сегмента, биты 0..15
32..39: адрес сегмента, биты 16..23
40: бит A (Accessed)
41..43: Тип сегмента
44: бит S (System)
45,46: DPL (Descriptor Privilege Level)
47: бит P (Present)
48..51: предел, биты 16..19
52: бит U (User)
53: бит X (reserved)
54: бит D (Default size)
55: бит G (Granularity)
56..63: адрес сегмента, биты 24..31

Эта структура поясняется на рис. 4-1:

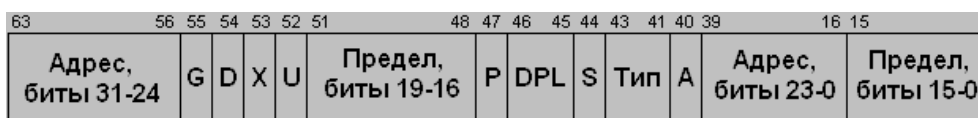


Рисунок 4-1. Структура дескриптора.

Как видите, значения предела и адреса сегмента "разбросаны" по всей структуре дескриптора. Это объясняется тем, что впервые защищённый режим появился в 16-разрядном процессоре 80286 и для совместимости с ним дескриптор не переделывали, а расширили дополнительными полями (биты с 49 по 63), благодаря чему программы, написанные для защищённого режима 286-го процессора работают и на 32-разрядных процессорах.

Практически, в программах формат дескриптора удобнее использовать в следующем виде:

descriptor:

dw	limit_low	; младшее слово предела
dw	address_low	; младшее слово адреса
db	address_hi	; 3-й (из четырёх) байт адреса
db	access_rights	; права доступа
db	limit_hi_and_flags	; старшая часть предела и флаги GDXX
db	address_hi	; 4-й байт адреса

Байт прав доступа имеет следующий формат:

биты:

0: бит A (Accessed)
1..3: тип сегмента
4: бит S (System)
5,6: поле DPL
7: бит P (Present)

Байт старшей части предела и флагов GDXX имеет формат:

биты:

0..3: старшая часть предела (биты 16..19)

4: бит U

5: бит X

6: бит D

7: бит G

Элементы дескриптора.

- **Адрес сегмента** - также называется базовым адресом, - 32-разрядный адрес области памяти, с которой начинается сегмент.
- **Предел сегмента** - предельное значение смещения в сегменте; также можно рассматривать предел как размер сегмента минус один элемент размера - байт или страницу, смотря в чём измеряется сегмент.
- **Бит А (Acessed)** - бит доступа в сегмент. Этот бит показывает, был ли произведен доступ к сегменту, описываемому этим дескриптором, или нет. Если процессор обращался к сегменту для чтения или записи данных или для выполнения кода, размещённых в нём, то бит А будет установлен (равен 1), иначе - сброшен (0).
С помощью бита А операционная система может определить, использовался ли за последнее время этот сегмент или нет и предпринять какие-либо действия.
Бит А процессором только устанавливается, сбрасывать его должна операционная система.
При создании нового дескриптора подразумевается, что бит А будет равен 0 (т.е. обращений к этому сегменту ещё не было).
- **Тип сегмента** - трёхбитовое поле, определяющее тип сегмента (см. таблицу 4-1). Каждый бит типа сегмента имеет следующие значения:
 - старший бит (3-й бит в байте прав доступа):
если 0, то это сегмент данных, если 1 - то кода
 - средний бит (2-й бит) - W (write-enable) бит разрешения записи:
если 0, то запись запрещена, 1 - разрешена
 - младший бит (1-й бит) - E (expansion-direction) направление расширения сегмента:
если 0, то вверх (в сторону старших адресов) - как обычно,
если 1, то вниз (в сторону младших адресов) - как в стеке.

Для сегмента кода значения битов W и E интерпретируются несколько иначе (см. таблицу 4-1)

Таблица 4-1. Тип сегмента

Бит #	11	10	9	Тип	
Название		E	W		
	0	0	0	Данные	Только чтение
	0	0	1	Данные	Чтение и запись
	0	1	0	Данные	Только чтение, расширяется вниз
	0	1	1	Данные	Чтение и запись, расширяется вниз
	1	0	0	Код	Только выполнение
	1	0	1	Код	Только выполнение
	1	1	0	Код	Только выполнение, согласованный
	1	1	1	Код	Выполнение и считывание, согласованный

Примечания:

1. В защищённом режиме процессор запрещает запись в сегмент кода и, как видно из таблицы, не всегда разрешает даже простое считывание (хотя, запись в сегмент кода, конечно же, можно сделать, но не явно)

2. Согласованный сегмент кода будет обсуждаться в следующих главах.

- Бит **S** (System) - определяет системный объект. Если этот бит установлен, то дескриптор определяет сегмент кода или данных, а если сброшен, то системный объект (например, сегмент состояния задачи, локальную дескрипторную таблицу, шлюз).
- Поле **DPL** (Descriptor Privilege Level) - Уровень привилегий, который имеет объект, описываемый данным дескриптором. Это двухбитовое поле, в него при создании дескриптора записывают значения от 0 до 3, определяющее уровень привилегий.

Например, Если вам нужно, чтобы процессор выполнял программу на нулевом уровне привилегий, то в DPL дескриптора сегмента кода, где размещается программа, должно быть значение 00B.

- Бит **P** (Present) - Присутствие сегмента в памяти. Если этот бит установлен, то сегмент есть в памяти, если сброшен, то его нет. Этот бит применяется при реализации механизма виртуальной памяти - если программе понадобится память, то она сохранит содержимое какого-либо сегмента на диск и сбросит бит P. Если любая программа в дальнейшем обратится к этому сегменту, то процессор сгенерирует исключение неприсутствующего сегмента и запустит обработчик этой ситуации, который должен будет подгрузить содержимое сегмента с диска и установить бит P. После этого управление снова передаётся команде, обратившейся к этому сегменту (производится повторное выполнение команды, вызвавшей сбой) и работа программы будет продолжена. Бит P устанавливается и сбрасывается программами, сам процессор его только считывает.
- Бит **U** (User) - Бит пользователя. Этот бит процессор не использует и позволяет программе использовать его в своих целях.
- Бит **X** (reserved) - Зарезервированный бит. Intel не рекомендует использовать этот бит, так как он может понадобиться в более поздних моделях процессоров.
- Бит **D** (Default size) - Размер операндов по умолчанию. Если бит сброшен, то процессор использует объект, описываемый данным дескриптором, как 16-разрядный, если бит установлен - то как 32-разрядный. Если ваша программа имеет 32-разрядный код, то он должен размещаться в 32-разрядном сегменте кода (т.е. в дескрипторе такого сегмента бит D должен быть равен 1). В защищённом режиме допускается использование одновременно 16- и 32-разрядных сегментов, но при написании новых программ подразумевается, что все сегменты будут 32-разрядные. Подробнее об этом см. в главе "Смешивание 16- и 32-разрядного кода".
- Бит **G** (Granularity) - Гранулярность сегмента, т.е. единицы измерения его размера. Если бит G=0, то сегмент имеет байтную гранулярность, иначе - страничную (одна страница - это 4Кб).

Например, сегмент, имеющий предел, равный 2, при G=0 будет иметь размер в три байта, а при G=1 - 12Кб (3 страницы)

Создавать дескрипторы достаточно легко. Все биты и битовые поля находятся в байте прав доступа `access_rights` (P, DPL, S, Тип, A) и в старших четырёх разрядах бита `limit_hi_and_flags` (G, D, X, U). Дескрипторы по типу отличаются значениями бита прав доступа, по свойствам - битами G и D, остальное - значения базового адреса и предела. Когда мы будем рассматривать пример перехода в защищённый режим, вы увидите один из вариантов конструирования дескрипторов.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр Семенко.



Защищённый режим:

Глава 5. Селектор.

При адресации памяти в защищённом режиме команды ссылаются на сегменты, указывая не их адреса (как в режиме реальных адресов), а описания сегментов (их дескрипторы). Указатель на описание сегмента называется селектор. Другими словами, селектор - это номер дескриптора из таблицы дескрипторов.

Адресация производится через пару регистров сегмент:смещение, причём, в качестве сегментного регистра используются обычные CS, SS, DS, ES, FS и GS (последние два появились в 386-м процессоре), но в них указывается не адрес сегмента, а селектор дескриптора.

Селекторы нужны, по крайней мере, по трём причинам:

1. Описание сегмента занимает 8 байт и использовать 8-байтные сегментные регистры было бы крайне неэффективно.
2. Селекторы имеют размер в 16 бит, благодаря чему их можно использовать в сегментных регистрах и обращаться к памяти можно по-прежнему через пару регистров.
3. Параметры всех сегментов хранятся в отдельной области памяти, доступ к которой имеет только операционная система. Программа, используя селектор, сможет получить о сегменте совсем немного информации и не сможет изменить параметры сегмента, благодаря чему очень удачно реализуется механизм защиты.

Адрес памяти можно указывать не только через пару регистров, но и в переменных, через пару значений селектор:смещение.

Можно было бы определить селектор просто, как номер сегмента, но в защищённом режиме кроме сегментов, дескриптор может определять целый ряд других системных объектов (например, задач), поэтому лучше не упрощайте понятия селектора и дескриптора, а постарайтесь привыкнуть к этой терминологии.

Селектор имеет следующий формат:

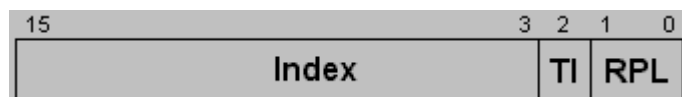


Рисунок 5-1. Формат селектора.

- Двухбитовое поле **RPL** (Requested Privilege Level) содержит номер уровня привилегий, которое имеет текущая программа. Значение этого поля процессор использует для защиты по привилегиям. К одному и тому же дескриптору можно обращаться, используя селекторы с

разными значениями RPL, но процессор позволит доступ только при определённых условиях (подробнее об этом см. в главе "Защита по привилегиям").

- Бит **TI** (Table Indicator) определяет таблицу, из которой выбирается нужный дескриптор. Если бит $TI = 0$, то обращение производится к глобальной дескрипторной таблице GDT (она одна на всю систему), если $TI = 1$ - то к текущей локальной дескрипторной таблице LDT (таких может быть много).

Подробнее дескрипторные таблицы обсуждаются в соответствующих главах.

- **Index** - это собственный номер дескриптора, от 0 до 8191. Т.к. поле индекса состоит из 13 бит, то максимальное число дескрипторов, одновременно существующих в системе равно 2^{13} , т.е. 8192. Как видите, это довольно-таки много и вполне удовлетворяет любым системным запросам. На самом деле, число дескрипторов можно значительно увеличить за счёт использования множества дополнительных локальных дескрипторных таблиц.

Использование селекторов достаточно простое. Для тех дескрипторов, которые будут определены заранее, например, сегментов кода, стека и данных, селекторы подготавливаются как константы и затем используются для загрузки в сегментные регистры. Для дескрипторов, которые программа будет динамически создавать, селекторы придётся определять в переменных и загружать в сегментные регистры из памяти либо конструировать "на ходу", или даже как константы - всё зависит от конкретных условий. Способы использования селекторов и дескрипторов вы можете найти в примерах, которые будут следовать в дальнейших главах.

Обращение к дескрипторной таблице процессор производит только в момент загрузки в сегментный регистр нового селектора. После этого содержимое дескриптора копируется в так называемый "теневого регистр", к которому имеет доступ только сам процессор и из которого оно в дальнейшем используется. Любое последующее обращение к сегменту будет происходить с помощью теневого регистра, без обращения к дескрипторной таблице и не потребует лишних тактов на циклы чтения памяти. Правда, эти такты тратятся каждый раз, когда вы загружаете новый селектор, но это не высокая плата за защиту дескрипторов от недозволенного доступа.

При загрузке недопустимого значения селектора процессор будет генерировать исключение, даже если вы не обращались через него к памяти.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Защищённый режим:

Глава 6. Использование регистров общего назначения для адресации.

Для обращения к памяти через регистры обычно используются четыре 16-разрядных регистра общего назначения (РОН): BX, SI, DI и BP. В защищённом режиме для адресации можно использовать все 8 регистров общего назначения.

В 32-разрядных процессорах обычные регистры расширены до 32-х разрядов. Вот так регистр AX расширен до регистра EAX:

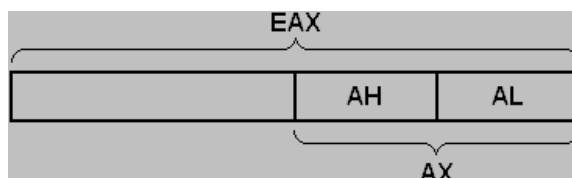


Рисунок 6-1. Формат регистра EAX.

Как видите, регистр AX является составной частью регистра EAX (так же, как регистры AL и AH являются составными частями регистра AX), т.е. если вы обращаетесь к регистру AX, то вы меняете содержимое регистра EAX.

Подобным образом расширены все 8 регистров общего назначения:

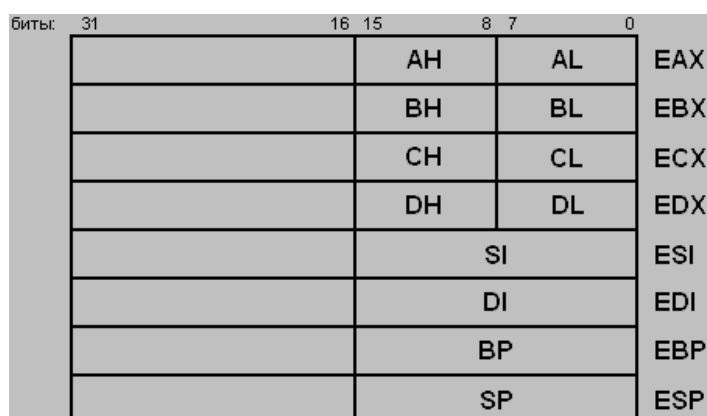


Рисунок 6-2. Регистры общего назначения.

Также, расширен 16-разрядный регистр FLAGS - теперь это 32-разрядный EFLAGS, младшая половина которого представляет собой FLAGS:

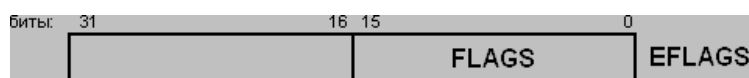


Рисунок 6-3. Формат регистра EFLAGS.

Регистр IP (Instruction Pointer) был расширен до 32-разрядного EIP:

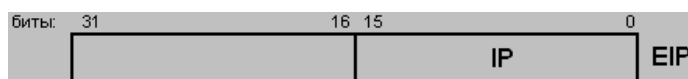


Рисунок 6-4. Формат регистра EIP.

Регистр EIP непосредственно использовать нельзя, но теперь следует учитывать, что при 32-разрядной адресации памяти в качестве адреса перехода можно указывать 32-разрядную величину.

Хотя указатель стека (регистр ESP) также относится к регистрам общего назначения и может использоваться в командах, настоятельно рекомендуется никогда не привлекать его к использованию вне стека. Особенно это важно при работе в защищённом режиме, когда процессор автоматически использует текущее значение стека, чтобы поместить в него значения, например, при обработке исключений.

В 32-разрядном процессоре вы по-прежнему можете адресовать память через четыре 16-разрядных регистра BX, SI, DI и BP, но дополнительно к этому можно использовать каждый из 32-разрядных регистров общего назначения, причём в любом режиме (не только защищённом). Например:

```
mov     ax,[ ebx ]      ; Поместить в AX значение из памяти
                        ; по адресу DS:EBX

mov     dx,[ ecx ]      ; Поместить в DX значение из памяти
                        ; по адресу DS:ECX

mov     cx,es:[ eax ]   ; Поместить в CX значение из памяти
                        ; по адресу ES:EAX
```

Дополнительно к этой возможности введены следующие:

- Использование константы и регистра:

```
mov     eax,[ ecx + 1 ]

mov     bl,[ edx + 12345678h ]
```

- Сумма двух регистров:

```
mov     ebp,[ ebx + edi ]

mov     eax,[ ecx + edx ]
```

- Сумма двух регистров и константы:

```
mov     bl,[ edx + eax + 12345678h ]
```

- Масштаб - автоматическое умножение на 2, 4 или 8 одного из регистров, участвующих в образовании адреса:

```
mov     ax,[ ebx * 2 ]
mov     cl,[ edx + ebp * 4 ]
mov     esi,[ edi + eax * 8 + 12345678h ]
```

Очевидно, что возможности, которые нам предоставляла XT, 32-разрядные процессоры значительно расширили. Следует, однако, учитывать, что в ранних процессорах (i386 и i486) на вычисление эффективного адреса процессор расходует дополнительное время, но благодаря универсальности этой системы адресации всё равно имеет место итоговый выигрыш в производительности.

При использовании 32-разрядных регистров для адресации в режиме реальных адресов, следует учитывать, что размер сегмента фиксирован и равен 64 Кб. Если процессор сформирует адрес, больший 64 Кб, то процессор зависнет, т.е. он не будет производить заворачивание адресов. Например:

```
mov     eax,1234h
mov     bl,[ eax ]      ; В регистр BL будет произведена загрузка
                        ; значения с адреса DS:EAX, равного DS:1234h.

mov     edx,ffffh
mov     bl,[ eax + edx ] ; Эффективный адрес будет равен
                        ; 1234h + ffffh = 11233h (это больше,
                        ; чем 64 Кб. Процессор зависнет.
```

Использование 32-разрядных регистров для адресации памяти в защищённом режиме очень распространено, в основном, из-за того, что размер сегментов может достигать 4 Гб, да и просто потому, что это удобно.

Copyright © Александр Семенко.



Защищённый режим:

Глава 7. Глобальная дескрипторная таблица.

Прежде, чем процессор перейдёт в защищённый режим, должна быть определена глобальная дескрипторная таблица GDT (Global Descriptor Table), так как все сегменты и прочие системные объекты должны быть описаны в дескрипторной таблице.

Глобальная дескрипторная таблица GDT - это область памяти, в которой находятся дескрипторы. Процессору всё равно, где именно вы расположили эту таблицу, но в любом случае она будет находиться в первом мегабайте адресного пространства, потому что только из режима реальных адресов можно перевести процессор в защищённый режим. Также подразумевается, что сама таблица GDT будет выравнена на границу 8 байт, так как дескрипторы, из которых она состоит, имеют 8-байтный размер. Такое выравнивание позволит процессору максимально быстро обращаться к дескрипторам, что, естественно, увеличивает производительность.

Число дескрипторов, определённых в GDT, может быть любым, от 0 до 8192. Нулевой дескриптор, т.е. определённый в самом начале GDT, процессор не использует, обращение к такому дескриптору могло бы быть, когда поле Index селектора равно 0. Если всё же в программе встречается обращение к нулевому дескриптору, то процессор генерирует исключение и не позволит доступ к такому дескриптору. В связи с этим, везде в литературе рекомендуется использовать нулевой дескриптор как шаблон, на основе которого программа может создавать новые дескрипторы, но на практике их удобнее создавать иными способами, о которых мы ещё будем говорить.

GDT используется процессором всё время, пока он находится в защищённом режиме. Параметры GDT хранятся в специальном 48-разрядном регистре GDTR:



Рисунок 7-1. Формат регистра GDTR.

Формат регистра GDTR следующий:

биты:

0..15: 16-разрядный предел GDT

15..47: 32-разрядный адрес начала GDT

Адрес начала GDT - это тот адрес, по которому вы разместили GDT.

Предел таблицы GDT - это максимальное смещение относительно её начала.

Например, вы создаёте GDT, состоящую из 3-х дескрипторов - для сегментов кода, стека и данных. Общее число дескрипторов будет равно четырём, потому что первым по счёту будет идти нулевой дескриптор, а за ним уже остальные три:

Смещение от начала GDT	Назначение дескриптора
0	Нулевой
8	Сегмент кода
16	Сегмент стека
24	Сегмент данных

Размер GDT в данном случае будет равен 32 байтам, следовательно, предельное смещение в таблице будет равно 31 - это и есть предел GDT.

Для загрузки значения в регистр GDTR используется команда LGDT. Операндом этой команды является 48-разрядное значение адреса в памяти, где размещается адрес и предел GDT. Вы также можете сохранить содержимое GDTR командой SGDT, указав в операнде адрес 48-разрядной переменной в памяти.

Далее приводится пример подготовки параметров GDT и их загрузки в регистр GDTR.

Пример 1. Подготовка параметров GDT и их загрузка в регистр GDTR.

Алгоритм:

1. Вычисляем 32-разрядный адрес GDT
2. Вычисляем размер GDT
3. Сохраняем параметры GDT в 48-разрядную переменную
4. Загружаем значение в регистр GDTR

Код:

; 1. Вычисляем 32-разрядный адрес GDT

```
xor  eax,eax    ; EAX = 0; адрес будем вычислять в
                ; регистре EAX.
mov  ax,ds      ; Подразумевается, что GDT находится в
                ; текущем сегменте данных.
shl  eax,4      ; EAX = адрес начала сегмента DS

xor  edx,edx    ; EDX = 0.
lea  dx,GDT     ; EDX = DX = смещение начала GDT
```

; относительно DS.

add eax,edx ; EAX = полный физический адрес GDT
; в памяти.

; 2. Вычисляем размер GDT

mov cx,4 ; CX = число дескрипторов +
; + нулевой дескриптор.
shl cx,3 ; CX = CX * 8 - столько байт будут
; занимать в GDT эти дескрипторы.

dec cx ; Предел меньше размера на 1

; 3. Сохраняем параметры GDT в 48-разрядную переменную

mov GDT_address, eax
mov GDT_limit, cx

; 4. Загружаем значение в регистр GDTR

lgdt GDT_params

;-----

GDT_params label dword
GDT_limit dw ?
GDT_address dd ?

GDT:

dd ? ; 0-й дескриптор
dd ?

dd ? ; 1-й дескриптор (код)
dd ?

dd ? ; 2-й дескриптор (стек)
dd ?

dd ? ; 3-й дескриптор (данные)
dd ?

Остаётся добавить, что размер GDT желательно не менять в процессе выполнения программ в защищённом режиме. Если ваша программа будет динамически создавать новые дескрипторы, то размер GDT лучше всего заранее задать достаточно большим, например, 64 Кб (максимальный

размер). Однако, следует учитывать, что при обращении процессора к несуществующим дескрипторам, его поведение непредсказуемо, хотя оно, скорее всего, закончится зависанием.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Защищённый режим:

Глава 8. Регистры управления.

Все 32-разрядные процессоры, начиная с i386, имеют набор системных регистров, предназначенных для использования в защищённом режиме. Среди них есть регистры управления (Control Registers) CR0, CR1, CR2, CR3 и CR4.

Регистры управления, в основном, состоят из флагов. Назначение и использование каждого флага достаточно сложно и требует отдельного рассмотрения. Для начала мы рассмотрим только один бит PE регистра CR0, отвечающего за переход процессора в защищённый режим и обратно. Полный список с описаниями регистров CRi приводится в приложении "Регистры управления CRi".

Регистры управления предназначены для считывания и записи информации. Они имеют размер в 32 бита и оперировать ими можно только целиком - считали значение целого регистра, изменили нужные биты и записали обратно. Единственная команда, которой позволен доступ к этим регистрам - это MOV, в качестве операнда которой используется 32-разрядный регистр общего назначения.

В дальнейших главах мы будем рассматривать переход в защищённый режим и возврат из него в режим реальных адресов, для чего будем использовать только один бит из регистра CR0 - это нулевой бит, который называется PE (Protection Enable). Если установить этот бит в 1, процессор перейдёт в защищённый режим, если сбросить - то в режим реальных адресов.

Процессор после аппаратного сброса переходит в режим реальных адресов и бит PE сброшен.

Вот примеры использования бита PE:

1. Переводим процессор в защищённый режим.

```
mov eax,cr0    ; Копируем в EAX содержимое регистра CR0.

or  al,1       ; Устанавливаем в копии 0-й бит, который
               ; соответствует 0-му биту CR0, т.е. биту PE.

mov cr0,eax    ; Записываем в CR0 обновлённое значение.
               ; Процессор перешёл в защищённый режим.

...           ; 1-я команда программы, которая выполнится
               ; в защищённом режиме.
```

2. Переводим процессор в режим реальных адресов.

```
mov eax,cr0
```

```
and al,0feh ; Сбрасываем бит PE.  
mov cr0,eax ; Процессор перешёл в режим реальных адресов.
```

На самом деле, если вы просто выполните переход в защищённый режим, как показано на приведенном выше примере, то процессор зависнет. Для работы в защищённом режиме процессор использует дескрипторы сегментов, вместо их адресов (которые находятся в сегментных регистрах), поэтому прежде, чем перевести процессор в защищённый режим, нужно провести некоторые подготовительные работы.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Защищённый режим:

Глава 9. Вход в защищённый режим.

Когда программа переводит процессор в защищённый режим, при всём том богатом потенциале, который предоставляют 32-разрядные процессоры, она оказывается практически беспомощной. В основном это связано с тем, что в защищённом режиме совершенно другая система прерываний и воспользоваться ресурсами, предоставляемыми операционной системой, в которой вы запускаете такую программу, невозможно. Более того, недоступными окажутся прерывания BIOS и IRQ. Подробно работа прерываний описана в разделе "Прерывания в защищённом режиме", там же вы найдёте примеры использования прерываний всех типов (программные, аппаратные и исключения), но пока наша программа не сможет ими воспользоваться.

В предыдущих главах не раз упоминалась фраза "операционная система", когда шла речь о программе, работающей в защищённом режиме. Дело в том, что после перевода процессора в P-Mode, программа должна определить действия и условия для всех ситуаций, вплоть до того, что определить драйвера (т.е. управляющие процедуры) для таких устройств, как клавиатура, мышь, видеоадаптер, дисковые накопители и даже таймер. Существует два способа реализации таких драйверов - либо написать их самому (что, вообще говоря, не очень сложно), либо обращаться к BIOS-у в режиме виртуального процессора 8086. Оба этих способа будут описаны в соответствующих разделах, где будет подразумеваться, что вы разобрались с разделом "Защищённый режим".

Прежде, чем мы перейдём к примеру, давайте определимся с тем, что нам надо сделать:

1. Подготовиться к переходу в P-Mode.
2. Перейти в P-Mode.
3. Сообщить в программе о переходе в P-Mode.

В третьем пункте мы заставим программу вывести на экран строку "I am in protected mode!!!", после чего программа зациклит процессор (компьютер повиснет). Дело в том, что возврат в режим реальных адресов немного сложнее, чем переход в защищённый режим и это требует дополнительного объяснения, поэтому в этом примере приводится только переход в P-Mode.

Вывод на экран будет происходить в текстовом режиме посредством прямой записи в видеопамять, для чего будет описана отдельная процедура - вы увидите, что код, предназначенный для выполнения в защищённом режиме не требует специальных определений (это будет 16-разрядный код; 32-разрядный определяется немного сложнее).

Остаётся добавить, что пример должен выполняться из простой

операционной системы, например, MS-DOS, работающей в режиме реальных адресов. Если вы попытаетесь запустить программу из-под ОС, работающей в защищённом режиме (например, Windows), то программа не заработает, т.к. процессор уже будет работать в P-Mode и не допустит повторного входа в этот режим.

Что касается зависания процессора в конце выполнения нашего примера - так это нормальное явление при отладке программ в защищённом режиме. Этот режим тем и характерен, что допускает работу только одного "хозяина" одновременно, а наша программа, войдя в защищённый режим, как раз и станет таким "хозяином процессора".

Хочу обратить ваше внимание на то, что первой командой после перехода в защищённый режим должна быть команда дальнего перехода (far jump), в которой будет указан селектор дескриптора сегмента кода и смещение в этом сегменте. При работе в защищённом режиме процессор может использовать в сегментных регистрах только селекторы существующих дескрипторов, любые другие значения (например, сегментный адрес) использовать нельзя - процессор сгенерирует исключение общей защиты. Тем не менее, при переходе в защищённый режим регистр CS будет содержать сегментный адрес, который использовался в режиме реальных адресов, поэтому выполнение следующей команды, какой бы она ни была, должно было бы привести к генерации процессором исключения. На самом деле этого не происходит, так как эта команда не выбирается из памяти - она уже находится в конвейере процессора (даже в таком процессоре, как i386, есть конвейер) и поэтому вы можете выполнить эту команду.

Команда дальнего перехода обязательно очистит конвейер процессора и заставит его обратиться к таблице GDT, выбрать оттуда дескриптор, селектор которого указан в адресе команды и начать выборку команд со смещения, также указанного в этом адресе. Это критический момент в работе программы. Если в GDT, селекторе, смещении или самой команде будет обнаружена ошибка, то процессор сгенерирует исключение, а так как систему прерываний мы для него пока не определяли, то он попросту зависнет либо произойдёт сброс - это уже зависит от "железа".

Если вы не выполните первой команду дальнего перехода, а другую, которая не изменит содержимое регистра CS (а это - все остальные команды), то процессор произведёт выборку в конвейер новой команды, используя текущие значения CS:IP, а так как в CS содержится не селектор (процессор уже в защищённом режиме!), то произойдёт исключение и зависание.

Это - теория. Такие условия перехода в P-Mode рекомендует Intel и опыт показывает, что лучше придерживаться этих рекомендаций. На практике наблюдаются некоторые чудеса. Например, можно сделать переход не дальний, а короткий (без смены значения в CS) - и программа будет работать, мало того, можно даже не определять стек - всё равно программа работает - процедуры вызываются, можно оперировать стеком, вот только нет уверенности, что стек будет отображаться на ту же область памяти, что

была до перехода.

С регистрами данных ситуация обстоит хуже - DS можно не инициализировать, но при работе через него вы получите совсем не те данные, что должны были бы, а обращение к ES, отображённому на видеопамять подвешивает процессор.

Эта ситуация была обнаружена при тестировании примера 2 на процессоре 80386 DX-40 и причина, на мой взгляд, в ошибках архитектуры, допущенных при проектировании этого процессора. Сам Intel подобные ошибки называет errata и сообщает обо всех обнаруженных багах в процессорах, самую свежую информацию о них вы можете найти на www.intel.com и www.intel.ru.

Использование ошибок процессоров не представляется мне практичным, т.к. вариаций одной и той же модели процессоров много - десятки, хотя если вы сможете найти достойное применение ошибкам, то я буду рад опубликовать на сайте ваши статьи.

И всё же, возвращаясь к примеру, давайте определимся: мы изучаем "правильное" программирование 32-разрядных процессоров, при этом не используя никаких ошибок в архитектуре и никаких недокументированных особенностей и команд. Это нам обеспечит уверенность в том, что наши программы будут надёжно работать на любых интеловских процессорах и их клонах.

Прежде, чем мы перейдём к примеру программы, давайте определим две функции, которые мы будем в дальнейшем использовать.

В программе нам понадобится создавать 4 дескриптора - для кода, данных (где будет храниться выводимая строка), стека (он будет использоваться для вызова функции вывода текста) и видеопамати. Все дескрипторы сегментов подобны друг другу, поэтому удобно их будет создавать отдельной функцией. Исходный код для этой функции (и для второй) поместите как макросы в отдельный файл - мы их будем использовать в дальнейшем и просто подключать к исходникам.

Функция первая, "set_descriptor" предназначенная для создания дескриптора, приводится сразу в том виде, в каком она будет находиться в подключаемом файле "pmode.lib".

При вызове этой функции подразумевается, что в паре регистров DS:BX находится указатель на текущую позицию в GDT. Функция после создания дескриптора переведёт указатель на позицию для следующего дескриптора. Все необходимые параметры передаются через регистры и функция всего лишь "перетасовывает" их в нужном порядке.

Файл "pmode.lib":

```
;-----
init_set_descriptor      macro

set_descriptor proc near
; Создаёт дескриптор.
; DS:BX = дескриптор в GDT
; EAX = адрес сегмента
; EDX = предел сегмента
; CL = байт прав доступа (access_rights)

    push eax
    push ecx      ; Регистры EAX и ECX мы будем использовать.

    push cx       ; Временно сохраняем значение access_rights.

    mov cx,ax      ; Копируем младшую часть адреса в CX,
    shl  ecx,16    ; и сдвигаем её в старшую часть ECX.

    mov cx,dx      ; Копируем младшую часть предела в CX.
                    ; Теперь ECX содержит младшую часть
                    ; дескриптора (т.е. первые 4 байта -
                    ; см. рис. 4-1).

    mov [ bx ],ecx ; Записываем младшую половину дескриптора в GDT.

    shr  eax,16    ; EAX хранит адрес сегмента, младшую часть
                    ; которого мы уже использовали, теперь будем
                    ; работать со старшей, для чего сдвигаем её в
                    ; младшую часть EAX, т.е. в AX.

    mov cl,ah      ; Биты адреса с 24 по 31
    shl  ecx,24    ; сдвигаем в старший байт ECX,
    mov cl,al      ; а биты адреса с 16 по 23 - в младший байт.

    pop  ax        ; Возвращаем из стека в AX значение
                    ; access_rights
    mov ch,al      ; и помещаем его во второй (из четырёх)
                    ; байт ECX.
                    ; Всё, дескриптор готов. Старшую часть
                    ; предела и биты GDXX мы не устанавливаем и
                    ; они будут иметь нулевые значения.

    mov [ bx + 4 ],ecx ; Дописываем в GDT вторую половину
                        ; дескриптора.

    add  bx,8      ; Переводим указатель в GDT на следующий
                        ; дескриптор.

    pop  ecx
```

```

    pop     eax

    ret

endp

endm
;-----

```

Вторую функцию "putzs" добавьте в файл "pmode.lib" после описания первой. Эта функция предназначена для вывода на экран строки, оканчивающейся нулевым байтом (т.е. байтом, равным 00h). Такая строка везде на этом сайте называется ZS-строка или просто ZS (от Zero-String). Также вам будут встречаться строки другого типа - LS (от Lenght-String), длина которых будет задана первым байтом.

Функция "putzs" получила своё название от комбинации слов "Put" и "ZS", по аналогии с похожими Си-функциями. В этой функции сохраняются используемые регистры - в приводимом далее примере это никакого эффекта не вызовет - процессор сразу после вывода зависнет, но для других примеров это будет полезно.

```

putzs    proc near
; DS:BX = ZS      ; ZS = Zero-String - строка, оканчивающаяся
                ; нулевым (00h) байтом.
; ES:DI = позиция вывода  ; ES описывает сегмент видеопамати,
                ; DI - смещение в нём.

    push ax
    push bx
    push es
    push di

    mov ah,1bh      ; В AH будет атрибут вывода - светло-циановые
                    ; символы на синем фоне.

putzs_1:
    mov al,[ bx ]   ; Читаем байт из ZS-строки.
    inc  bx         ; Переводим указатель на следующий байт.
    cmp al,0        ; Если байт равен 0,
    je   putzs_end   ; то переходим в конец процедуры.

    mov es:[ di ],ax ; Иначе - записываем символ вместе с
                    ; атрибутом в видеопамать по заданному
                    ; смещению - цветной символ появится на
                    ; экране.

```

```

    add di,2      ; Переводим указатель в видеопамяти на
                  ; позицию следующего символа.

    jmp putzs_1   ; Повторяем процедуру для следующего байта
                  ; из ZS-строки.

putzs_end:

    pop di
    pop es
    pop bx
    pop ax

    ret

endp
;-----

```

Теперь сам пример. Прежде чем приступить к его изучению, хочу сделать следующие замечания:

1. Программа протестирована и работает, поэтому, если она у вас не заработает, проверьте, Вы не ошиблись? Если нет - пишите, разберёмся.
2. Программа протестирована как отдельный .com-файл. В принципе, вы её можете без изменений перенести в образ .exe-файла или встроить в программу языка высокого уровня - всё должно работать.
3. Обратите внимание, что таблица GDT и сегменты не выровнены в памяти и программа всё равно работает. Это сделано специально, для демонстрации возможностей P-Mode. Для повышения производительности программы, конечно же следует выравнивать все структуры данных, используемые процессором непосредственно (у нас пока это только GDT) и сегменты.

Далее полностью приводится файл "example2.asm", который содержит пример перехода в защищённый режим, включая необходимые для tasm-а атрибуты.

Пример 2. Переход в защищённый режим.

Файл "example2.asm":

```

include pmode.lib; Подразумевается, что файл "pmode.lib" находится
                  ; в том же каталоге, что и "example2.asm".
;-----
.386p
pmode segment use16

```

```

assume  cs:pmode, ds:pmode, es:pmode

org 100h

main    proc far
start:
;-----
;-----
; Определяем селекторы как константы. У всех у них биты TI = 0 (выборка
; дескрипторов производится из GDT), RPL = 00B - уровень привилегий -
; нулевой.

Code_selector = 8
Stack_selector = 16
Data_selector = 24
Screen_selector = 32
;-----

    mov bx, offset GDT + 8 ; Нулевой дескриптор устанавливать
                           ; не будем - всё равно он не
                           ; используется.

    xor  eax, eax          ; EAX = 0
    mov  edx, eax          ; EDX = 0

    push cs
    pop  ax                ; AX = CS = сегментный адрес текущего
                           ; сегмента кода.

    shl  eax, 4            ; EAX = физический адрес начала сегмента кода.
                           ; Эта программа, работая в среде операционной системы
                           ; режима реальных адресов (подразумевается, что это -
                           ; MS-DOS) уже имеет в IP смещение относительно
                           ; текущего сегмента кода. Мы определим дескриптор
                           ; кода для защищённого режима с таким же адресом
                           ; сегмента кода, чтобы при переходе через команду
                           ; дальнего перехода фактически переход произошёл
                           ; на следующую команду.

    mov  dx, 1024          ; Предел сегмента кода может быть любым,
                           ; лишь бы он покрывал весь реально
                           ; существующий код.

    mov  cl, 10011000b     ; Права доступа сегмента кода (P = 1,
                           ; DPL = 00b, S = 1, тип = 100b, A = 0)

    call set_descriptor    ; Конструируем дескриптор кода.

    lea  dx, Stack_seg_start ; EDX = DX = начало стека (см. саму
                           ; метку).

    add  eax, edx          ; EAX уже содержит адрес начала сегмента

```

```

; кода, сегмент стека начнётся с последней
; метки программы Stack_seg_start.

mov dx,1024 ; Предел стека. Также любой (в данном
; примере), лишь бы его было достаточно.

mov cl,10010110b ; Права доступа дескриптора сегмента
; стека (P = 1, DPL = 00b, S = 1,
; тип = 011b, A = 0).

call set_descriptor ; Конструируем дескриптор стека.

xor eax,eax ; EAX = 0
mov ax,ds
shl eax,4 ; EAX = физический адрес начала сегмента данных.
xor ecx,ecx ; ECX = 0
lea cx,PMode_data_start ; ECX = CX
add eax,ecx ; ECX = физический адрес начала сегмента
; данных.

lea dx,PMode_data_end
sub dx,cx ; DX = PMode_data_end - PMode_data_start (это
; размер сегмента данных, в данном примере
; он равен 26 байтам). Этот размер мы и
; будем использовать как предел.

mov cl,10010010b ; Права доступа сегмента данных (P = 1,
; DPL = 00b, S = 1, тип = 001, A = 0).

call set_descriptor ; Конструируем дескриптор данных.

mov eax,0b8000h ; Физический адрес начала сегмента
; видеопамати для цветного текстового
; режима 80 символов, 25 строк
; (используется по умолчанию в MS-DOS).

mov edx,4000 ; Размер сегмента видеопамати (80*25*2 = 4000).
mov cl,10010010b ; Права доступа - как сегмент данных
call set_descriptor ; Конструируем дескриптор сегмента
; видеопамати.

; Устанавливаем GDTR:

xor eax,eax ; EAX = 0
mov edx,eax ; EDX = 0

mov ax,ds
shl eax,4 ; EAX = физический адрес начала сегмента данных.
lea dx,GDT
add eax,edx ; EAX = физический адрес GDT
mov GDT_adr,eax ; Записываем его в поле адреса образа GDTR.

```

```
mov dx,39          ; Предел GDT = 8 * (1 + 4) - 1
mov GDT_lim,dx     ; Записываем его в поле предела образа GDTR.
```

```
cli               ; Запрещаем прерывания. Для того, чтобы прерывания
                  ; работали в защищённом режиме их нужно специально
                  ; определять, что в данном примере не делается.
```

```
lgdt GDTR         ; Загружаем образ GDTR в сам регистр GDTR.
```

; Переходим в защищённый режим:

```
mov eax,cr0
or  al,1
mov cr0,eax
```

; Процессор в защищённом режиме!

```
db  0eah          ; Этими пятью байтами кодируется команда
dw  P_Mode_entry   ; jmp far Code_selector:P_Mode_entry
dw  Code_selector
```

;-----
P_Mode_entry:

; В CS находится уже не сегментный адрес сегмента кода, а селектор его
; дескриптора.

; Загружаем сегментные регистры. Это обеспечит правильную работу программы
; на любом 32-разрядном процессоре.

```
mov ax,Screen_selector
mov es,ax
```

```
mov ax,Data_selector
mov ds,ax
```

```
mov ax,Stack_selector
mov ss,ax
mov sp,0
```

; Выводим ZS-строку:

```
mov bx,0 ; DS:BX = указатель на начало ZS-строки. Адрес
          ; сегмента данных определён по метке
          ; PMode_data_start, а строка начинается сразу после
          ; этой метки, её смещение от метки равно 0,
          ; следовательно, это и будет смещение от начала
          ; сегмента данных.
```

```
mov di,480      ; Выводим ZS-строку со смещения 480 в
                  ; видеопамати (оно соответствует началу
                  ; 3-й строки на экране в текстовом режиме).
call putzs
```

; Зацикливаем программу:

```
loop_1:
    jmp loop_1
```

```
;-----
init_set_descriptor
init_putzs
;-----
```

; Образ регистра GDTR:

```
GDTR    labelword
GDT_lim    dw    ?
GDT_adr    dd    ?
```

```
;-----
GDT:
```

```
    dd    ?,? ; 0-й дескриптор
    dd    ?,? ; 1-й дескриптор (кода)
    dd    ?,? ; 2-й дескриптор (стека)
    dd    ?,? ; 3-й дескриптор (данных)
    dd    ?,? ; 4-й дескриптор (видеопамяти)
```

```
;-----
```

PMode_data_start: ; Начало сегмента данных для защищённого режима.

```
;-----
```

db "I am in protected mode!!!",0 ; ZS-строка для вывода в P-Mode.

```
;-----
```

PMode_data_end: ; Конец сегмента данных.

```
;-----
```

db 1024 dup (?) ; Зарезервировано для стека.

Stack_seg_start: ; Последняя метка программы - отсюда будет расти стек.

```
;-----
```

```
main    endp
pmode   ends
end start
```

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Защищённый режим:

Глава 10. Возврат в режим реальных адресов.

Практически защищённый режим процессора можно использовать двумя способами:

1. Операционная система. Процессор входит в P-Mode при загрузке ОС и не возвращается в R-Mode. Примером таких ОС являются Windows 95 и старше либо специализированные ОС, управляющие контроллерами на базе 32-разрядных процессоров.
2. DOS-программа. Программа запускается в ОС, которая работает в R-Mode (например, MS-DOS), переходит в P-Mode, выполняет свою работу, возвращается обратно в R-Mode и завершает свою работу. Примеров множество: это `hymem.sys`, обеспечивающий через P-Mode работу с памятью выше 1-го мегабайта (XMS-память); также это игрушки, работающие через Dos4GW - предзагружаемую ОС защищённого режима; это Windows 3.xx.

Второй способ, с возвратом в режим реальных адресов, предоставляет программисту больше возможностей - ведь не все пишут полнофункциональные операционные системы...

Фактически, переход в режим реальных адресов может быть только из защищённого режима и осуществляется сбросом бита PE в CR0:

```
mov    eax,cr0
and    al,0feh
mov    cr0,eax
```

или так

```
mov    eax,cr0
btr    eax,0
mov    cr0,eax
```

Вы наверное уже догадались, что на самом деле не всё так просто. Действительно, выполнение трёх вышеприведенных команд переведёт процессор в R-Mode, но дальше он повиснет либо произойдёт аппаратный сброс, потому что программная среда не будет соответствовать режиму реальных адресов.

Для корректного перехода из P-Mode в R-Mode необходимо подготовить процессор следующим образом:

1. Запретить прерывания (CLI).
2. Передать управление в читаемый сегмент кода, имеющий предел в 64Кб (FFFFh).
3. Загрузить в SS, DS, ES, FS и GS селекторы дескрипторов, имеющих следующие параметры:
 - Предел = 64 Кб (FFFFh)
 - Байтная гранулярность (G = 0)
 - Расширяется вверх (E = 0)
 - Записываемый (W = 1)
 - Присутствующий (P = 1)
 - Базовый адрес = любое значение

Сегментные регистры должны быть загружены ненулевыми селекторами. Те сегментные регистры, в которые не будут загружены описанные выше значения, будут использоваться с атрибутами, установленными в защищённом режиме.

4. Сбросить флаг **PE** в **CR0**.
5. Выполнить команду **far jmp** на программу режима реальных адресов.
6. Загрузить в регистры SS, DS, ES, FS и GS необходимые значения или 0.
7. Разрешить прерывания (STI).

В предыдущем примере (2) мы просто перевели процессор в защищённый режим, при этом не используя особенности прерываний в P-Mode и не подключали механизм виртуальной памяти.

Реализация каждой из этих технологий немного усложнит возврат в R-Mode и в разделах, описывающих эти технологии будут приведены соответствующие примеры перехода в P-mode и возврата из него.

Здесь подразумевается, что вы уже ознакомились со всеми предыдущими главами, поэтому предварительного подробного описания перехода в R-Mode не будет и мы перейдём сразу к примеру.

Этот пример является полноценной программой и в нём без комментариев повторяется переход в P-Mode. Процесс подготовки дескрипторов и GDTR зависит от предназначения каждого примера и по этой причине я не вынес его в отдельную функцию а полностью описываю каждый раз.

Пример ориентирован на использование как самостоятельная .com-программа. Это сделано по следующим причинам:

1. Пользоваться обычной программной средой - библиотеки языков высокого уровня, прерывания MS-DOS и прочими особенностями этой ОС из защищённого режима вы не сможете.
2. Размеры кода и данных в данном примере не большие и удобно использовать формат файла .com.
3. .com-программу можно просто загрузить в память как блок данных и, передав управление на смещение 100h, использовать как оверлей.
4. Вся мощь защищённого режима раскрывается при использовании 32-разрядного кода и данных. Инициализация защищённого режима должна происходить 16-разрядным кодом и 32-разрядные программы удобно использовать как отдельные модули.

Также хочу обратить ваше внимание на то, что в этом примере адреса сегментов для защищённого режима совпадают с адресами сегментов, используемых в R-Mode. В результате, обращение к памяти происходит непосредственно через метки, определённые в исходнике (в предыдущем примере для этого требовалось из адреса метки вычитать адрес начала сегмента данных).

Пример используется вместе с файлом "pmode.lib".

Пример 3. Вход в защищённый режим и возврат в режим реальных адресов.

```
include pmode.lib
;-----
.386p
pmode segment use16
    assume  cs:pmode, ds:pmode, es:pmode

    org     100h

main          proc      far
start:
;-----
;-----
; Определяем селекторы как константы. У всех у них биты TI = 0 (выборка
; дескрипторов производится из GDT), RPL = 00B - уровень привилегий -
; нулевой.

Code_selector    = 8
Stack_selector   = 16
Data_selector    = 24
Screen_selector  = 32
R_Mode_Code      = 40      ; Селектор дескриптора сегмента кода для возврата
                           ; в режим реальных адресов.
R_Mode_Data      = 48      ; Селектор дескриптора сегментов стека и данных
;-----

; Сохраняем сегментные регистры, используемые в R-Mode:

    mov     R_Mode_SS,ss
    mov     R_Mode_DS,ds
    mov     R_Mode_ES,es
    mov     R_Mode_FS,fs
```

```

        mov     R_Mode_GS,gs

; Подготавливаем адрес возврата в R-Mode:

        mov     R_Mode_segment,cs
        lea     ax,R_Mode_entry
        mov     R_Mode_offset,ax

; Подготовка к переходу в защищённый режим:

        mov     bx,offset GDT + 8
        xor     eax,eax
        mov     edx,eax

        push    cs
        pop     ax

        shl     eax,4
        mov     dx,1024
        mov     cl,10011000b
        call    set_descriptor          ; Code

        lea     dx,Stack_seg_start
        add     eax,edx
        mov     dx,1024
        mov     cl,10010110b
        call    set_descriptor          ; Stack

        xor     eax,eax
        mov     ax,ds
        shl     eax,4
        mov     dx,0ffffh
        mov     cl,10010010b
        call    set_descriptor          ; Data

        mov     eax,0b8000h
        mov     edx,4000
        mov     cl,10010010b
        call    set_descriptor          ; Screen

; Готовим дополнительные дескрипторы для возврата в R-Mode:

        xor     eax,eax

        push    cs
        pop     ax

        shl     eax,4          ; EAX = физический адрес сегмента кода
                                ; (и всех остальных сегментов, т.к.
                                ; это .com-программа)

        mov     edx,0ffffh
        mov     cl,10011010b    ; P=1, DPL=00b, S=1, Тип=101b, A=0
        call    set_descriptor  ; R_Mode_Code

        mov     cl,10010010b    ; P=1, DPL=00b, S=1, Тип=001b, A=0
        call    set_descriptor  ; R_Mode_Data

; Устанавливаем GDTR:

        xor     eax,eax
        mov     edx,eax

        mov     ax,ds

```

```

    shl     eax,4
    lea     dx,GDT
    add     eax,edx
    mov     GDT_adr,eax

    mov     dx,55          ; Предел GDT = 8 * (1 + 6) - 1
    mov     GDT_lim,dx

    cli

    lgdt    GDTR

    mov     R_Mode_SP,sp    ; Указатель на стек сохраняем в последн
                           ; момент.

; Переходим в защищённый режим:

    mov     eax,cr0
    or      al,1
    mov     cr0,eax

; Процессор в защищённом режиме

    db      0eah           ; Команда far jmp Code_selector:P_Mode_entry.
    dw      P_Mode_entry
    dw      Code_selector

;-----
P_Mode_entry:

    mov     ax,Screen_selector
    mov     es,ax

    mov     ax,Data_selector
    mov     ds,ax

    mov     ax,Stack_selector
    mov     ss,ax
    mov     sp,0

; Сообщаем о входе в P-Mode (выводим ZS-строку):

    lea     bx,Start_P_Mode_ZS
    mov     di,480
    call    putzs

; Работа программы в защищённом режиме (здесь - только вывод строки):

    lea     bx,P_Mode_ZS
    add     di,160
    call    putzs

; Возвращаемся в режим реальных адресов.
; 1. Запретить прерывания (CLI).
;    Прерывания уже запрещены при входе в P-Mode.

; 2. Передать управление в читаемый сегмент кода, имеющий предел в 64Кб.

    db      0eah           ; Команда far jmp R_Mode_Code:Pre_R_Mode_entry.
    dw      Pre_R_Mode_entry
    dw      R_Mode_Code

Pre_R_Mode_entry:

```

```

; 3. Загрузить в SS, DS, ES, FS и GS селекторы дескрипторов, имеющих
; следующие параметры:
; 1) Предел = 64 Кб (FFFFh)
; 2) Байтная гранулярность (G = 0)
; 3) Расширяется вверх (E = 0)
; 4) Записываемый (W = 1)
; 5) Присутствующий (P = 1)
; 6) Базовый адрес = любое значение

        mov     ax,R_Mode_Data    ; Селектор R_Mode_Data - "один на всех".
        mov     ss,ax
        mov     ds,ax
        mov     es,ax
        mov     fs,ax
        mov     gs,ax

; 4. Сбросить флаг PE в CR0.

        mov     eax,cr0
        and     al,0feh           ; FEh = 1111'1110b
        mov     cr0,eax

; 5. Выполнить команду far jump на программу режима реальных адресов.

        db      0eah
R_Mode_offset dw      ?          ; Значения R_Mode_offset и R_Mode_segment
R_Mode_segment dw      ?          ; сюда прописались перед входом в
                                   ; защищённый режим (в начале программы).
;-----
R_Mode_entry:
; 6. Загрузить в регистры SS, DS, ES, FS и GS необходимые значения или 0
; (восстанавливаем сохранённые значения):

        mov     ss,R_Mode_SS
        mov     ds,R_Mode_DS
        mov     es,R_Mode_ES
        mov     fs,R_Mode_FS
        mov     gs,R_Mode_GS

        mov     sp,R_Mode_SP      ; Восстанавливаем указатель стека
                                   ; непосредственно перед разрешением
                                   ; прерываний.

; 7. Разрешить прерывания (STI).

        sti

; Выводим ZS-строку "Back to real address mode..."

        lea     bx,R_Mode_ZS
        mov     ax,0b800h
        mov     es,ax
        mov     di,800
        call    putzs              ; Функция putzs универсальна и работает
                                   ; в обоих режимах.

        int     20h               ; Конец программы (здесь - выход в MS-DOS).

;-----
; Вставка макросами кода функций, определённых в текстовом файле
; "pmode.lib":

init_set_descriptor
init_putzs

```

```

;-----
; ZS-строка для вывода при входе в P-Mode:
Start_P_Mode_ZS:      db      "Entering to protected mode...",0

; ZS-строка для вывода при работе в P-Mode:
P_Mode_ZS:            db      "Working in P-mode...",0

; ZS-строка для вывода в R-Mode:
R_Mode_ZS:            db      "Back to real address mode...",0

;-----
; Значения регистров, которые программа имела до перехода в P-Mode:
R_Mode_SP             dw      ?
R_Mode_SS             dw      ?
R_Mode_DS             dw      ?
R_Mode_ES             dw      ?
R_Mode_FS             dw      ?
R_Mode_GS             dw      ?
;-----
; Образ регистра GDTR:

GDTR      label      fword
GDT_lim   dw          ?
GDT_adr   dd          ?
;-----
GDT:
      dd      ?,?      ; 0-й дескриптор
      dd      ?,?      ; 1-й дескриптор (кода)
      dd      ?,?      ; 2-й дескриптор (стека)
      dd      ?,?      ; 3-й дескриптор (данных)
      dd      ?,?      ; 4-й дескриптор (видеопамяти)
      dd      ?,?      ; 5-й дескриптор (код для перехода в R-Mode)
      dd      ?,?      ; 6-й дескриптор (стек и данные для перехода в R-Mode)
;-----
      db      1024 dup (?)      ; Зарезервировано для стека.
Stack_seg_start:      ; Последняя метка программы - отсюда будет расти стек.
;-----
main      endp
      pmode  ends
      end    start

```

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр Семенко.



Прерывания в защищённом режиме:

Глава 1. Основы работы прерываний.

Как вы уже знаете, в режиме реальных адресов может существовать 256 различных прерываний. В начале памяти, по адресу 0000:0000 расположена таблица прерываний, состоящая из 256 четырёхбайтовых дальних (far) адресов в виде dw сегмент:dw смещение. Каждый такой адрес указывает на процедуру, обрабатывающую прерывание. Такие процедуры называются обработчики прерываний, а адреса - векторами.

Режим реальных адресов позволяет использовать два типа прерываний:

1. Аппаратные прерывания - 16 прерываний от внешних устройств, отображённых на 16 векторов.
2. Программные прерывания - прерывания, генерируемые командой INT n.

В защищённом режиме работа прерываний происходит сложнее.

Во-первых, вводится новый класс прерываний, генерируемых самим процессором при нарушениях условий защиты - так называемые исключения (exceptions). Число возможных векторов прерываний по-прежнему равно 256, но 32 из них - от 00h до 1Fh используются исключениями.

Во-вторых, вместо дальних адресов в таблице прерываний используются дескрипторы специальных системных объектов, так называемых шлюзов.

В-третьих, сама таблица прерываний, которая называется IDT (Interrupt Descriptors Table), может находиться по любому адресу памяти.

Все эти особенности появились в процессоре 80386 и в полном объёме, с небольшими дополнениями, используются в всех 32-разрядных процессорах.

Прерывания с векторами от 00 до 1Fh, т.е. исключения - это основа защищённого режима. Благодаря исключениям процессор автоматически реагирует на любые попытки нарушить защиту системы и позволяет их корректно обработать. Благодаря разделению кода и данных по уровням привилегий, обработчики прерываний можно надёжно изолировать от других программ.

В грамотно построенной операционной системе никакая программа не сможет перехватить прерывание, изменить код или даже просто прочитать его(!), выйти за предел отведённых ей адресов и пр. Благодаря исключениям, операционная система может контролировать любые нарушения условий, поставленных ею.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Прерывания в защищённом режиме:

Глава 2. Дескрипторы прерываний.

Когда срабатывает прерывание, процессор должен передать управление соответствующей процедуре-обработчику. В режиме реальных адресов это происходит сразу - из памяти выбирается вектор и по `dw:dw` адресу происходит переход. В защищённом режиме ситуация обстоит сложнее - перед передачей управления процессор производит множество проверок возможности доступа к обработчику прерывания - обеспечивает защиту.

Адрес, по которому произойдёт переход на обработчик прерывания, находится в дескрипторе прерывания. Каждому вектору прерыванию соответствует свой дескриптор, все они (до 256) объединяются в специальную таблицу дескрипторов прерываний IDT и по формату похожи на дескрипторы сегментов, которые мы рассматривали в разделе "Защищённый режим".

Таблица дескрипторов прерываний (IDT) в любой системе - одна. Программ (задач, процедур, приложений и пр.) - много. IDT реализуется на нулевом уровне привилегий и, следовательно, непосредственно к ней обратиться могут только программы, работающие на том же уровне. Для того, чтобы программы с других уровней (1, 2 и 3) могли пользоваться прерываниями, предусмотрены специальные системные объекты - так называемые шлюзы (gates). При вызове прерывания, процессор, прежде, чем передать управление обработчику, "опускается" через шлюз на его уровень привилегий, а после завершения обработки - "поднимается" обратно.

IDT может содержать три типа дескрипторов шлюзов:

- Шлюз задачи
- Шлюз прерывания
- Шлюз ловушки

Шлюзы содержат указатели на обработчики прерываний и права доступа к ним. При переходе через шлюз задачи, процессор производит автоматическое переключение задач, а при переходе через шлюз прерывания или ловушки передаёт управление процедуре в контексте текущей программы. Единственное отличие прерывания от ловушки в том, что при переходе через шлюз прерывания процессор автоматически сбрасывает флаг `IF` в `EFLAGS` и тем самым не допускает генерации других прерываний и исключений на время работы обработчика, а для шлюза ловушки - не меняет состояние флага `IF`. Ловушки используются для отладки программ и поэтому обработка ловушки должна быть прозрачна для внешних прерываний.

Исключения и прерывания работают в основном через два типа шлюзов - задач и прерываний. Шлюз прерывания запускает обработчик в контексте

текущей программы, т.е. просто передаёт управление по адресу, указанному в дескрипторе. Такой подход хорош только в простых операционных системах, когда работают заранее определённые программы, от которых не нужно защищать ядро ОС.

Шлюз задачи является более удобным и универсальным, т.к. позволяет изолировать обработчик от других программ и его рекомендуется применять в системах, где программы потенциально могут нарушить целостность ОС. Шлюз задачи заставляет процессор автоматически переключаться на новую задачу при генерации исключения. Т.к. мультизадачность мы ещё не рассматривали, то обработчики исключений реализуем пока через шлюзы прерываний и ловушек.

Далее приводятся форматы дескрипторов шлюзов:

1. Шлюз задачи.

```
dw      0
dw      TSS_sel      ; Селектор TSS
db      0
db      access_rights ; Права доступа сегмента TSS
dw      0
```

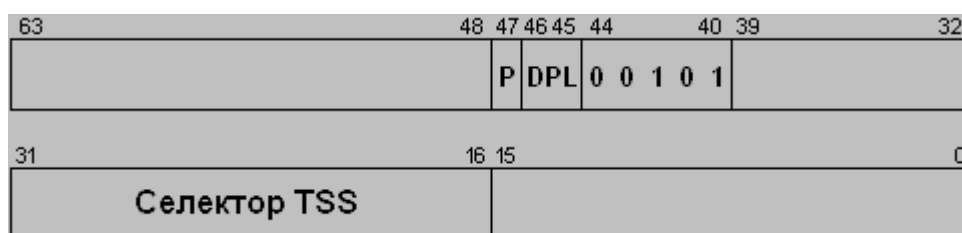


Рисунок 2-1. Схема шлюза задачи.

Обратите внимание на то, что бит 4 в `access_rights`, соответствующий биту `S` в формате дескриптора, равен 0. Это значит, что дескриптор описывает системный объект и биты 0..3 в `access_rights` определяют тип этого объекта.

Первое и последнее слова (`dw`) в формате дескриптора содержат 0, т.к. любая задача определяется своим дескриптором, на который и ссылается селектор TSS (подробно о задачах см. в разделе "Мультизадачность").

2. Шлюз прерывания.

```
dw      offset_low    ; Младшая часть смещения
dw      selector      ; Селектор сегмента кода
db      0
db      access_rights  ; Права доступа
dw      offset_hi     ; Старшая часть смещения
```

Шлюз прерывания через селектор и смещение задаёт адрес обработчика прерывания.

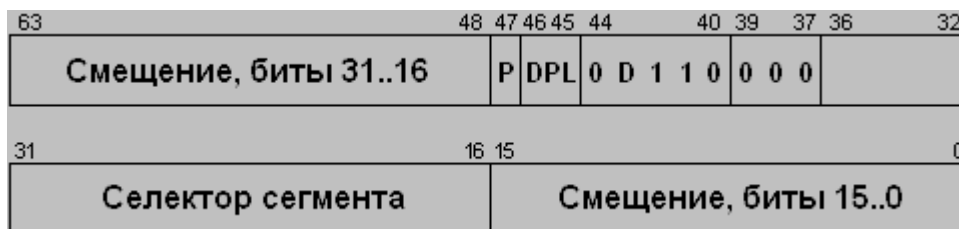


Рисунок 2-2. Схема шлюза прерывания.

3. Шлюз ловушки.

```

dw      offset_low      ; Младшая часть смещения
dw      selector        ; Селектор сегмента кода
db      0
db      access_rights    ; Права доступа
dw      offset_hi        ; Старшая часть смещения

```

Шлюз ловушки через селектор и смещение задаёт адрес обработчика прерывания.

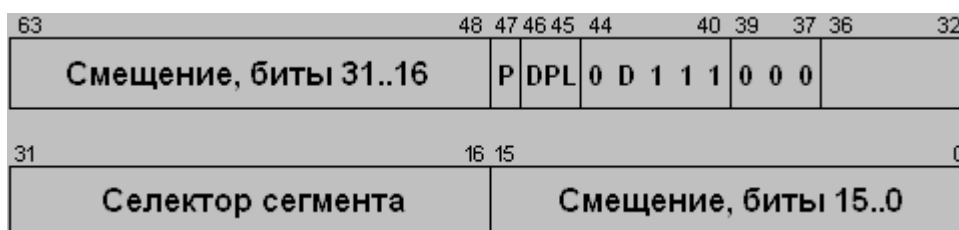


Рисунок 2-3. Схема шлюза ловушки.

Примечание.

D - это размер шлюза: 1 = 32 бита; 0 = 16 бит. Размер шлюза определяет размер стека, используемый процессором по умолчанию. Перед вызовом обработчика, процессор помещает в стек значения регистров CS, EIP, EFLAGS и иногда SS, ESP и dw-код ошибки. Если размер шлюза - 32 бита, то значения размером в 16 бит будут расширены нулями до 32-х.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Прерывания в защищённом режиме:

Глава 3. Исключения.

Исключениями называются прерывания, которые генерирует процессор в ответ на нарушения условий защиты. Повлиять на исключения прикладные программы (работающие на уровне привилегий, выше 0) не могут, замаскировать - тоже. Аппаратный контроль защиты - самый надёжный и 32-разрядные процессоры предоставляют этот сервис в полном объёме.

Исключения делятся на три типа, в зависимости от условий их возникновения:

1. Ошибка (fault)
2. Ловушка (trap)
3. Авария (abort)

Ошибка - это исключение, возникающая в ситуации ошибочных действий программы и подразумевается, что такую ошибку можно исправить. Такой тип исключения позволяет рестарт "виноватой" команды после исправления ситуации, для чего в стеке обработчика адрес возврата из прерывания указывает на команду, вызвавшую исключение. Примером такого исключения может быть исключение неprisутствующего сегмента (прерывание 0Bh), возникающее при попытке обратиться к сегменту, в дескрипторе которого бит P=0. Благодаря этому реализуется механизм виртуальной памяти, в частности, подкачка данных с диска.

Ловушка - это исключение, возникающее сразу после выполнения "отлавливаемой" команды. Это исключение позволяет продолжить выполнение программы со следующей команды (без рестарта "виноватой"). На ловушках строится механизм отладки программ.

Авария - это исключение, которое не позволяет продолжить выполнение прерванной программы и сигнализирует о серьёзных нарушениях целостности системы. Примером аварии служит исключение двойного нарушения (прерывание 8), когда сама попытка обработки одного исключения вызывает другое исключение.

Для некоторых исключений процессор помещает в стек обработчика двухбайтовый код ошибки, позволяющий конкретно определить ошибку.

Далее приводится полный список исключений и прерываний. В этой таблице применяются следующие обозначения:

Номер вектора - номер вектора прерывания, на которое отображено исключение.

Название - используется в документации Intel и состоит из заглавных букв английского названия исключения, например, #DE - Divide Error.

Error code - наличие dw-кода ошибки, который процессор добавляет в стек обработчика перед передачей ему управления.

Таблица 3-1. Исключения и прерывания защищённого режима.

Номер вектора	Название	Описание	Тип	Error Code	Источник исключения
0	#DE	Ошибка деления	Fault	Нет	Команды DIV и IDIV
1	#DB	Отладка	Fault/Trap	Нет	Любая команда или команда INT 1
2	-	Прерывание NMI	Прерывание	Нет	Немаскируемое внешнее прерывание
3	#BP	Breakpoint	Trap	Нет	Команда INT 3
4	#OF	Переполнение	Trap	Нет	Команда INTO
5	#BR	Превышение предела	Fault	Нет	Команда BOUND
6	#UD	Недопустимая команда (Invalid Opcode)	Fault	Нет	Недопустимая команда или команда UD2 ¹
7	#NM	Устройство не доступно (No Math Coprocessor)	Fault	Нет	Команды плавающей точки или команда WAIT/FWAIT
8	#DF	Двойная ошибка	Abort	Да (Нуль)	Любая команда
9	-	Превышение сегмента сопроцессора (зарезервировано)	Fault	Нет	Команды плавающей точки ²
0Ah	#TS	Недопустимый TSS	Fault	Да	Переключение задач или доступ к TSS
0Bh	#NP	Сегмент не присутствует	Fault	Fault	Загрузка сегментных регистров или доступ к сегментам
0Ch	#SS	Ошибка сегмента стека	Fault	Да	Операции над стеком и загрузка в SS
0Dh	#GP	Общая защита	Fault	Да	Любой доступ к памяти и прочие проверки защиты

0Eh	#PF	Страничное нарушение	Fault	Да	Доступ к памяти
0Fh	-	Зарезервировано Intel-ом. Не использовать.		Нет	
10h	#MF	Ошибка плавающей точки в x87 FPU (Ошибка математики)	Fault	Нет	Команда x87 FPU или команда WAIT/FWAIT
11h	#AC	Проверка выравнивания	Fault	Да	(Ноль) Обращение к памяти ³
12h	#MC	Проверка оборудования	Abort	Нет	Наличие кодов и их содержимое зависит от модели ⁴
13h	#XF	Исключение плавающей точки в SIMD	Fault	Нет	Команды SSE и SSE2 ⁵
14h-1Fh	-	Зарезервировано Intel-ом. Не использовать			
20h-FFh	-	Прерывания определяются пользователем	Прерывание		Внешнее прерывание или команда INT n

Примечания:

1. Команда UD2 появилась в процессоре Pentium Pro.
2. Процессоры IA-32 после Intel386 не генерируют это исключение.
3. Это исключение появилось в процессоре Intel486.
4. Это исключение появилось в процессоре Pentium и развивается в процессорах семейства P6.
5. Это исключение появилось в процессоре Pentium III.

Некоторое небольшое число исключений, являющиеся ошибками, не позволяют продолжить выполнение программы, т.к. при их генерации теряется часть данных программы, в которой произошла ошибка. Например, выполнение команды POPAD при недостаточном размере стека, вызывает такое исключение. В таком случае, обработчик исключения будет подразумевать, что команда POPAD не выполнена, хотя часть регистров общего назначения уже может измениться. При обнаружении таких ошибок рекомендуется прекращать выполнение программы, вызвавшей сбой.

Операционная система должна определить дескрипторы для всех исключений. Если процессор при генерации исключения не обнаружит в IDT соответствующего дескриптора, это приведёт к генерации ещё одного исключения. Процессор может обработать два исключения последовательно, но в некоторых случаях, когда это ему не удаётся, он генерирует [исключение двойной ошибки](#).

Если в IDT не будет дескриптора и для исключения двойной ошибки, либо не сможет корректно его обработать, то процессор переходит в режим отключения, похожий на режим, в который его переводит команда HLT (т.е. попросту, зависает) и будет реагировать только на сигналы типа аппаратного сброса.

Сами обработчики исключений не обязательно должны выполнять свои функции. На этапе создания операционной системы достаточно сделать "заглушки", которые будут выводить на экран номер исключения и параметры, переданные в стеке. Т.к. действия обработчиков сильно отличаются в зависимости от предназначения ОС, то здесь, в этом разделе, будут приведены лишь примеры "заглушек", а примеры обработчиков исключений и аппаратных прерываний вы сможете найти в соответствующих разделах.

В приложениях к теме "защищённый режим" вы можете найти [приведенную выше таблицу](#) с ссылками на подробные описания каждого исключения, а в этой главе она предназначена для ознакомления.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Прерывания в защищённом режиме:

Глава 4. Таблица дескрипторов прерываний.

Все дескрипторы прерываний и исключений объединяются в одну таблицу IDT (Interrupt Desriptor Table). Сама IDT может располагаться в памяти по любому адресу и состоять из любого числа дескрипторов в пределах от 0 до 256. В отличие от GDT, нулевой дескриптор в IDT используется нулевым вектором (исключение деления на 0).

Для неиспользуемых векторов бит Р дескрипторов должен быть равен 0, тогда при попытке обращения к нему процессор будет генерировать исключение неприсутствующего сегмента и ОС сможет корректно обработать неиспользуемое прерывание. В противном случае, скорее всего, возникнет другое исключение, тип которого заранее предусмотреть невозможно.

Для повышения производительности системы, рекомендуется размещать IDT по адресу, кратному 8. Размер IDT должен быть кратен 8, т.к. она состоит из 8-байтных дескрипторов, а предел, следовательно, на 1 меньше.

Если произойдёт обращение к вектору прерывания, дескриптор которого должен находиться за пределами IDT, то процессор сгенерирует исключение общей защиты.

Параметры IDT (адрес и предел) процессор хранит с специальным 48-разрядном регистре IDTR. Формат этого регистра следующий:

биты:

0..15: 16-разрядный предел IDT

16..48: 32-разрядный адрес начала IDT

Адрес начала IDT - это тот адрес, по которому вы разместили IDT.

Предел таблицы IDT - это максимальное смещение относительно её начала.

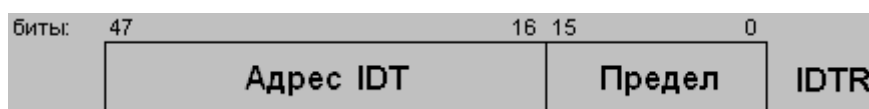


Рисунок 4-1. Формат регистра IDTR.

Подготовка и запись значения IDTR аналогична действиям для GDTR, поэтому соответствующий пример здесь не приводится.

Для загрузки содержимого IDTR из памяти в регистр используется команда LIDT, для сохранения из регистра в память - SIDT, причём, команда IDTR может выполняться только на нулевом уровне привилегий, а SIDT - на любом. Единственным операндом у обеих команд является адрес 48-разрядной переменной.

Программа, работающая не на 0-м уровне привилегий может получить

адрес и предел IDT и тут уже от самой операционной системы зависит, разрешит ли она доступ непривилегированной программе к IDT.

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр
Семенко.



Прерывания в защищённом режиме:

Глава 5. Аппаратные прерывания.

В режиме реальных адресов принято отображать аппаратные прерывания на фиксированные вектора: IRQ 0..7 - на вектора прерываний 8..0Fh, IRQ 8..15 - на 70h..7Fh. При работе в защищённом режиме такая схема работы IRQ нам не подходит, т.к. вектора 8..0Fh заняты исключениями. В связи с этим возникает необходимость при установке системы прерываний в защищённом режиме перенаправить аппаратные прерывания на другие вектора, лежащие за пределами 00..1Fh, а при возврате в режим реальных адресов - обратно, на 8..0F и 70h..7Fh.

Обработкой аппаратных прерываний в процессорах Intel386 и Intel486 занимается микросхема 8259A, а в Pentium и старше - продвинутый программируемый контроллер прерываний APIC (Advanced Programmable Interrupt Controller). Программирование APIC-а - это отдельная тема, а так как в этом разделе мы рассматриваем установку системы прерываний в защищённом режиме, то программировать APIC не будем. APIC обладает замечательным свойством - его можно отключить или, другими словами, запретить (disable) и тогда он будет эмулировать работу внешнего контроллера прерываний 8259A. Этим мы и воспользуемся.

APIC есть только в процессорах, начиная с Pentium, да и то не у всех, поэтому возникает необходимость обнаружения APIC в процессоре. В начале нашего примера будет сделан вызов на процедуру "what_cpu", которая выполнит следующее:

1. Не допустит выполнение программы, если процессор не 32-разрядный (i286 или XT).
2. Произведёт поиск локального APIC в процессоре и если обнаружит его, то установит переменную db APIC_presence в 1, иначе - в 0.

Текст этой процедуры здесь не приводится, т.к. обнаружение APIC затрагивает другие темы, которые здесь не обсуждаются - определение типа процессора и команда CPUID; по этой же причине, комментарии в самой процедуре минимальны. Исходный текст "what_cpu" заложен внутри исходника примера, который будет приведен в конце следующей главы.

Программирование контроллера прерываний также является отдельной темой, поэтому комментарии и здесь минимальны. Пока просто используйте эти "магические" действия:

1. Процедура P_Mode_redirect_IRQ отключает APIC, по мере необходимости и перенаправляет прерывания IRQ на вектора 20h..2Fh. Далее во всех наших примерах будет использоваться такая схема распределения векторов прерываний:

00..1Fh - исключения

20h..2Fh - IRQ

30h..FFh - программные прерывания

2. Процедура R_Mode_redirect_IRQ используется перед возвратом в R-Mode, она восстанавливает вектора прерываний IRQ.
3. Процедура redirect_IRQ используется обеими вышеприведенными процедурами, её прообраз взят из BIOS-а и слегка модифицирован для наших примеров.
4. Процедура disable_APIC запрещает APIC для правильной работы нашего примера, enable_APIC разрешает APIC только если он был включён.

```
;-----
init_P_Mode_redirect_IRQ      macro

P_Mode_redirect_IRQ          proc    near

    cmp     APIC_presence,1
    jne     pmrirq_1
```

```

        call    disable_APIC

pmrirq_1:
        mov     bx,2820h
        mov     dx,0FFFFh
        call    redirect_IRQ

        ret

endp

endm
;-----
init_R_Mode_redirect_IRQ      macro

R_Mode_redirect_IRQ      proc      near

        mov     bx,7008h
        xor     dx,dx
        call    redirect_IRQ

        cmp     APIC_presence,1
        jne     rmrirq_1
        call    enable_APIC

rmrirq_1:
        ret

endp

endm
;-----
init_redirect_IRQ      macro

redirect_IRQ      proc      near
; BX = { BL = Начало для IRQ 0..7, BH = начало для IRQ 8..15 }
; DX = Маска прерываний IRQ ( DL - для IRQ 0..7, DH - IRQ 8..15 )

        mov     al,11h
        out     0a0h,al
        jcxz    $+2
        jcxz    $+2
        out     20h,al
        jcxz    $+2
        jcxz    $+2

        mov     al,bh
        out     0a1h,al
        jcxz    $+2
        jcxz    $+2
        mov     al,bl
        out     21h,al
        jcxz    $+2
        jcxz    $+2

        mov     al,02
        out     0a1h,al
        jcxz    $+2
        jcxz    $+2
        mov     al,04
        out     21h,al
        jcxz    $+2
        jcxz    $+2

```

```

        mov     al,01
        out     0a1h,al
        jcxz    $+2
        jcxz    $+2
        out     21h,al
        jcxz    $+2
        jcxz    $+2

        mov     al,dh
        out     0a1h,al
        jcxz    $+2
        jcxz    $+2
        mov     al,dl
        out     21h,al
        jcxz    $+2
        jcxz    $+2

        ret

endp

endm
;-----
disable_APIC    proc    near
; Отключение локального APIC

        mov     bl,0

        mov     ecx,1bh
        db      0fh, 32h          ; Код команды RDMSR

        test    ah,1000b
        jz      dapic_end          ; Если APIC был уже отключён, то сброс
                                    ; переменной APIC_presence в 0 не допустит
                                    ; разрешения APIC в процедуре
                                    ; R_Mode_redirect_IRQ.

        and     ah,11110111b      ; Сбрасываем 11-й бит в MSR 1Bh
        db      0fh, 30h          ; Код команды WRMSR

        mov     bl,1

dapic_end:
        mov     APIC_presence,bl

        ret

endp
;-----
enable_APIC     proc    near
; Включение локального APIC

        mov     ecx,1bh
        db      0fh, 32h          ; Код команды RDMSR

        or      ah,1000b          ; Устанавливаем 11-й бит в MSR 1Bh
        db      0fh, 30h          ; Код команды WRMSR

        ret

endp
;-----

```

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр Семенко.



Прерывания в защищённом режиме:

Глава 6. Установка IDT в программе.

Как уже говорилось в разделе "защищённый режим", воспользоваться прерываниями BIOS и DOS программа, работающая в P-Mode, не может. Это связано с тем, что вектора от 00h до 1Fh заняты исключениями либо зарезервированы для них, следовательно, прерывания, отображённые на эти вектора, предоставляют доступ не к ресурсам BIOS-а, а к обработчикам прерывания. Например, команда INT 10h в режиме реальных адресов обеспечивает доступ к сервису управления видеоадаптером, а в защищённом режиме - к обработчику исключения плавающей точки x87 FPU.

Итак, для того, чтобы определить прерывания в защищённом режиме, нужно выполнить следующие действия:

1. Перенаправить аппаратные прерывания (IRQ)
2. Создать дескрипторы для всех используемых векторов (исключений, аппаратных и программных прерываний).
3. Подготовить образ IDTR и загрузить его в регистр IDTR.
4. Разрешить прерывания

Действие 1 были рассмотрены в предыдущей главе, действие 3 - см. в источнике - оно совмещено с построением GDT; в этой главе будет обсуждаться только 2-е.

Некоторые процедуры, из-за своей громоздкости, здесь будут приведены не полностью - будет показан только принцип их построения. В конце главы вы сможете найти ссылку на архив (7Кб) с файлами источника, библиотеки и самой программы.

Теперь для каждого примера будет своя библиотека, т.к. макросы, определяющие обработчики прерываний, можно будет менять и дополнять в пределах изучаемой темы и отдельная библиотека для каждого примера позволит избежать путаницы и лишней работы по редактированию макросов.

Создаём дескрипторы для всех используемых векторов прерываний.

Для всех исключений и прерываний создадим дескрипторы шлюзов прерываний и ловушек; шлюзы задач будут рассматриваться отдельно в разделе "Мультизадачность".

Шлюзы прерываний и ловушек содержат точку входа (сегмент:смещение) и права доступа обработчика. В следующем ниже примере функцию самих обработчиков будут выполнять заглушки, определённые соответствующими макросами; функциональная реализация обработчиков исключений и аппаратных прерываний будет описана в дальнейших главах.

Для установки дескрипторов прерываний давайте определим три следующих функции. Базовая функция `set_IDT_descriptor` устанавливает дескриптор с правами доступа в CX. Шлюзы прерывания и ловушки отличаются всего лишь одним битом в байте прав доступа (бит D в дескрипторе шлюза ловушки равен 0, т.к. стек, используемый обработчиком ловушек в нашем примере - 16-разрядный).

```
init_set_IDT_descriptor macro

set_IDT_descriptor      proc      near
; Установка IDT-дескриптора.
; DS:BX = дескриптор в IDT
; DX = селектор сегмента кода обработчика
; EAX = смещение в сегменте кода
; CX = access_rights (права доступа)

        push    eax
```

```

        push    ecx

        push    cx
        mov     cx,dx
        shl     ecx,16
        mov     cx,ax          ; ECX = dw Селектор & dw offset_low
        mov     [ bx ],ecx

        pop     ax
        mov     [ bx + 4 ],eax ; EAX = dw offset_hi & db access_rights & dk

        add     bx,8

        pop     ecx
        pop     eax

        ret

endp

endm

```

```

init_set_int_IDT_descriptor    macro

set_int_IDT_descriptor proc    near
; Установка IDT-дескриптора прерывания.
; DS:BX = дескриптор в IDT
; DX = селектор сегмента кода обработчика
; EAX = смещение в сегменте кода

        push    cx

        mov     cx,8600h          ; Права доступа шлюза прерывания
        call    set_IDT_descriptor

        pop     cx

        ret

endp

endm

```

```

init_set_trap_IDT_descriptor    macro

set_trap_IDT_descriptor proc    near
; Установка IDT-дескриптора ловушки.
; DS:BX = дескриптор в IDT
; DX = селектор сегмента кода обработчика
; EAX = смещение в сегменте кода

        push    cx

        mov     cx,8700h          ; Права доступа шлюза ловушки с D=0
        call    set_IDT_descriptor

        pop     cx

        ret

endp

```


endm

Обработчики прерываний определены следующим образом (здесь приводится неполный макрос, полностью - см. в конце главы, в архиве):

```
init_handlers    macro
; Обработчики исключений

ex_00_entry_point:
    exeption_00_handler
ex_01_entry_point:
    exeption_01_handler
...

ex_1f_entry_point:
    exeption_1f_handler

; Обработчики аппаратных прерываний

IRQ_0_entry_point:
    IRQ_0_handler
IRQ_1_entry_point:
    IRQ_1_handler
...

IRQ_f_entry_point:
    IRQ_f_handler

; Обработчики программных прерываний

Int_30_entry_point:
    Int_30_handler
Int_31_entry_point:
    Int_31_handler

endm
```

Сами макросы exeption_xx_handler, IRQ_x_handler и Int_3x_handler определяются следующим образом:

```
exeption_00_handler    macro

ex_00_start:
    jmp      ex_00_start

endm

exeption_01_handler    macro

ex_01_start:
    jmp      ex_01_start
```

```
endm
```

И т.д.

```
IRQ_1_handler    macro

IRQ_1_start:
    jmp          IRQ_1_start

endm
```

```
IRQ_2_handler    macro

IRQ_2_start:
    jmp          IRQ_2_start

endm
```

И т.д.

```
Int_30_handler    macro

    iret

endm
```

```
Int_31_handler    macro

    iret

endm
```

Такое определение макросов позволяет легко менять обработчики - просто заменить соответствующий макрос, не меняя при этом исходника.

Теперь определение IDT. Оно производится функцией `setup_IDT`, имеющей вид:

```
init_setup_IDT    macro

setup_IDT          proc    near

    lea    bx, IDT
    mov    dx, Code_selector    ; Обработчики все исключений и
                                ; прерываний в данном примере
                                ; находятся в одном сегменте кода.

    lea    eax, ex_00_entry_point
    call   set_int_IDT_descriptor

endm
```

```

lea    eax,ex_01_entry_point
call   set_trap_IDT_descriptor        ; Ловушка

lea    eax,ex_02_entry_point
call   set_int_IDT_descriptor

lea    eax,ex_03_entry_point
call   set_trap_IDT_descriptor        ; Ловушка

lea    eax,ex_04_entry_point
call   set_trap_IDT_descriptor        ; Ловушка

lea    eax,ex_05_entry_point
call   set_int_IDT_descriptor

...

lea    eax,ex_1f_entry_point
call   set_int_IDT_descriptor


lea    eax,IRQ_0_entry_point
call   set_int_IDT_descriptor

lea    eax,IRQ_1_entry_point
call   set_int_IDT_descriptor

...

lea    eax,IRQ_f_entry_point
call   set_int_IDT_descriptor


lea    eax,Int_30_entry_point
call   set_int_IDT_descriptor

lea    eax,Int_31_entry_point
call   set_int_IDT_descriptor

ret

endp

endm

```

Вот и все основные функции, необходимые для правильной работы системы прерываний в защищённом режиме. Для того, чтобы установить или обновить обработчик прерывания или исключения, нужно всего лишь изменить соответствующий макрос вида `..._handler`, все остальные функции при компиляции правильно построят IDT и обеспечат корректную работу системы.

Обратите внимание, что все обработчики исключений и аппаратных прерываний представляют собой простое заикливание - это и есть простейший пример реализации заглушек. Эти заглушки необходимы на начальной стадии построения операционной системы, когда вы ещё не решили, как будут обрабатываться исключения и прерывания, но работать они должны.

В главе 7 будут показаны примеры заглушек для исключений, которые будут выдавать

информацию о самих исключениях.

В этом примере определена IDT из 32h (т.е. 50) дескрипторов. Для примера - этого вполне хватает, для других программ - может не хватить, но главное, чтобы вы знали о возможности увеличить/уменьшить IDT простой заменой значения макроса `idt_descr_n`.

Для программных прерываний в примере определено всего 2 дескриптора - просто чтобы было понятно, как их определять. На самом деле, операционная система защищённого режима не нуждается в программных прерываниях - её сервис удобней предоставлять в виде вызовов соответствующих процедур или задач, к тому же, в защищённом режиме прерывание обрабатывается довольно-таки долго, по сравнению с дальним вызовом.

Сам пример рабочей программы и библиотеки вы можете скачать здесь: `pmode_4.lib`, `examp_4.asm` и `examp_4.com` в архиве [examp_4.zip](#) (7230 байт).

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр Семенко.



Прерывания в защищённом режиме:

Глава 7. Заглушки.

Зажушками называются процедуры, которые не допускают дальнейшее выполнение программы. При этом они либо явно зацикливают процессор (простая заглушка), либо производят возврат в режим реальных адресов и прекращение программы. На стадии разработки и отладки операционной системы защищённого режима заглушки являются нормальной частью программы.

При обработке простой заглушки процессор, естественно, "виснет", зато есть гарантия, что он остался в нужной точке программы, а не выполняет не предусмотренные действия. Пример такой заглушки - команда JMP с адресом перехода на саму себя, реализованный в примере 4 в предыдущей главе.

Однако, для нас всегда важно знать, откуда был переход на заглушку и причину этого перехода, поэтому в этой главе мы рассмотрим использование зажушек вместо обработчиков исключений, которые будут сообщать об исключении, переводить процессор в R-Mode и прекращать программу.

При вызове обработчика исключения процессор помещает в стек информацию, благодаря которой можно узнать причину возникшего исключения. Указателем на стек является пара регистров SS:SP, но т.к. мы рассматриваем программирование 32-разрядных процессоров, то во всех указателях будем использовать 32-разрядное смещение; для стека это будет SS:ESP. На рис. 7-1 представлен формат стека обработчика исключения после передачи ему управления:

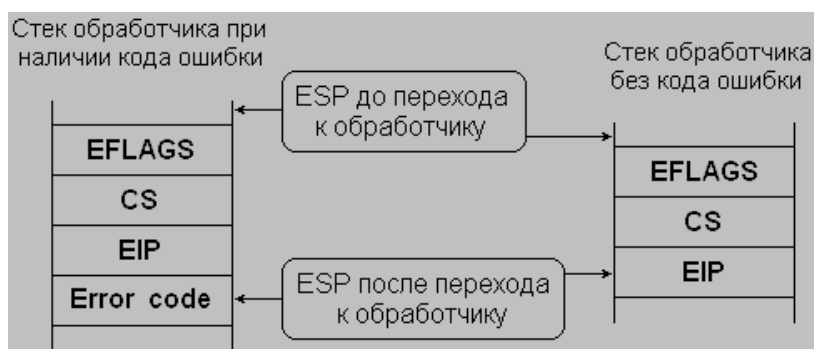


Рисунок 7-1. Использование стека при передаче управления обработчику прерывания.

Не все исключения снабжаются кодом ошибки, поэтому для каждого обработчика нужно учитывать его наличие (см. в Приложениях ["Исключения и прерывания защищённого режима"](#)).

Для вывода информации об исключениях в примере к этой главе введены следующие процедуры:

- **put_zs** - вывод ZS-строки (ZS = Zero-String - строка, заканчивающаяся нулём)
- **put_db_num**, **put_dw_num** и **put_dd_num** - вывод db-, dw- и dd-числа соответственно.

Эти процедуры производят вывод на экран в координаты X,Y, переданные в регистре DX и с атрибутом цвета, определённым в переменной text_color. Для преобразования координат в адрес используется процедура "get_adr".

Далее, во всех наших примерах эти процедуры сохраняются и для текстового режима на них будет построена система вывода.

Процедура putzs, определённая в предыдущих примерах, остаётся. Она является универсальной, т.к. для вывода использует адрес видеопамати в паре регистров ES:DI и

может использоваться как в R-Mode, так и в P-Mode.

Исходный текст этих процедур здесь не приводится, т.к. целью данной главы является определение заглушек, а не вывод на экран. В конце главы есть ссылка на архив с исходником, где вы сможете посмотреть реализацию этих процедур.

Принцип работы заглушек для всех исключений у нас одинаков - вывести на экран надписи и значения, переданные в стеке, поэтому все эти действия будет выполнять одна процедура "stopper".

Stopper имеет два входных параметра:

- номер исключения в регистре AL
- значение флага CF в EFLAGS - если CF = 1, значит в стеке есть кода ошибки.

Все обработчики исключений теперь будут заменены на следующие последовательности команд:

- Для исключений без кода ошибки (например, исключение деления на 00h):

```
mov    al,00h           ; Номер исключения.
clc                    ; Кода ошибки нет.
jmp     stopper
```

- Для исключений с кодом кода ошибки (например, исключение общей защиты 0Dh):

```
mov    al,0dh           ; Номер исключения.
stc                    ; Код ошибки есть.
jmp     stopper
```

Для возврата в режим реальных адресов теперь используется переход на метку "Return_to_R_Mode".

Далее следует текст заглушки "stopper":

```
stopper:
; AL = номер исключения
; CF = 1/0 - код ошибки есть / нет

mov     text_color,lah
mov     cl,al           ; Сохраняем в CL номер исключения

mov     al,0            ; AL = признак наличия кода ошибки
                        ; (1/0 - есть / нет).
jnc     stop_1

mov     al,1

stop_1:
xor     ebx,ebx
mov     bx,ss           ; EBX = BX = селектор стека
lar     edx,ebx ; EDX = старшая половина дескриптора
                        ; сегмента стека.
shr     edx,22
test    dl,1           ; Проверяем бит D (размерность сегмента
                        ; 16 или 32 бит).
jnz     stop_3

; стек - 16-разрядный
```

```

        cmp     al,1           ; Код ошибки есть?
        jne     stop_2

        pop     ax             ; Error code
        mov     dx,0747h
        call    put_dw_num

stop_2:
        pop     ax             ; IP
        mov     dx,0647h
        call    put_dw_num

        pop     ax             ; CS
        mov     dx,0547h
        call    put_dw_num

        pop     ax             ; FLAGS
        mov     dx,0447h
        call    put_dw_num

        jmp     stop_5

stop_3:
        ; стек - 32-разрядный

        cmp     al,1           ; Код ошибки есть?
        jne     stop_4

        pop     eax            ; Error Code
        mov     dx,0747h
        call    put_dd_num

stop_4:
        pop     eax            ; EIP
        mov     dx,0647h
        call    put_dd_num

        pop     eax            ; CS
        mov     dx,0547h
        call    put_dd_num

        pop     eax            ; EFLAGS
        mov     dx,0447h
        call    put_dd_num

stop_5:
        mov     text_color,1fh

        mov     dx,0347h
        mov     al,cl
        call    put_db_num      ; Выводим номер исключения, переданный в
                                ; процедуру в регистре AL.

        mov     text_color,1bh

        lea     bx,exept_mess_1
        mov     dx,033bh
        call    put_zs          ; "Exeption:"

```

```

mov     dx,043bh
call    put_zs           ; "EFLAGS:"

mov     dx,053bh
call    put_zs           ; "CS:"

mov     dx,063bh
call    put_zs           ; "EIP:"

mov     dx,073bh
call    put_zs           ; "Error Code:"

jmp     Return_to_R_Mode ; Выход в R-mode

```

Для демонстрации работы заглушек используется команда UD2, предназначенная для генерации исключения недопустимой команды.

Исходник примера можно скачать здесь: [examp_5.asm](#), [pmode_5.lib](#) и [examp_5.com](#) в архиве [examp_5.zip](#) (9177 байт).

[Следующая глава](#)

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр Семенко.



Прерывания в защищённом режиме:

Глава 8. Обработчики аппаратных прерываний.

Обработка аппаратных прерываний значительно отличается в различных ОС, поэтому имеет смысл давать лишь общие рекомендации. Более серьёзно к этому вопросу мы подойдём после того, как изучим мультитзадачность и виртуальную память, а пока при реализации обработчиков аппаратных прерываний придерживайтесь следующего:

1. Не используйте в IDT шлюзы ловушек, а только прерываний, т.к. при переходе через шлюз прерывания процессор автоматически запрещает маскируемые прерывания (сбрасывая флаг IF в EFLAGS), но не делает этого для шлюза ловушки.
2. В начале обработки прерывания посылайте в контроллер 8259A команду конца прерывания (EOI). Контроллер состоит из двух контроллеров master и slave. Master обслуживает первые 8 IRQ, slave - вторые и для них посылка EOI будет выглядеть так:

- для master (IRQ 0..7)

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

- для slave (IRQ 8..15)

```
mov     al, 20h
out     0a0h, al
```

3. Постарайтесь сделать обработку прерывания как можно быстрее, т.к. процессор не допустит генерации нового прерывания, пока не будет завершён обработчик.
4. При перенаправлении прерываний процедура "redirect_IRQ" запрещает контроллеру генерацию аппаратных прерываний. Значения в портах 21h и A1h содержат флаги маскировки прерываний для master- и slave-контроллера соответственно.

Для того, чтобы разрешить какое-либо прерывание, нужно сбросить соответствующий бит, а для запрещения - установить.

Прерывания master:		
Бит	IRQ	Устройство
0	0	Таймер
1	1	Клавиатура
2	2	Каскад (подключён ко второму контроллеру)
3	3	COM 2/4
4	4	COM 1/3
5	5	LPT 2
6	6	Контроллер дисководов FDC (Floppy Drive Controller)
7	7	LPT 1
Прерывания slave:		
Бит	IRQ	Устройство
0	8	Часы реального времени RTC (Real Time Clock)
1	9	Редирект с IRQ 2
2	10	Резерв (т.е. не имеет устройства по умолчанию)
3	11	Резерв (т.е. не имеет устройства по умолчанию)
4	12	Резерв (т.е. не имеет устройства по умолчанию)
5	13	Исключение сопроцессора
6	14	Контроллер винчестера HDC (Hard Drive Controller)
7	15	Резерв (т.е. не имеет устройства по умолчанию)

Например, для разрешения прерывания таймера нужно выполнить следующее:

```

in      al,21h          ; Читаем маску master-a
and     al,0feh ; FEh = 11111110b - сбрасываем 0-й бит.
out     21h,al          ; Записываем маску в контроллер. Таймер разрешё

```

Как правило, операционная система защищённого режима подразумевает возврат в режим реальных адресов и выход в ту ОС, из которой её запускали (например, в MS-DOS). В таком случае необходимо предусмотреть правильное маскирование прерываний IRQ перед возвратом в такую ОС, так как обычно не все прерывания разрешены.

Начиная со следующего примера в начале будет использоваться процедура, сохраняющая маску прерываний IRQ:

```

store_R_Mode_IRQ_Mask  proc    near
; Сохраняет значение маски IRQ в переменную R_Mode_IRQ_Mask для корректного
; восстановления IRQ при возврате в R-Mode.

    in     al,0a1h
    mov    ah,al
    in     al,21h
    mov    R_Mode_IRQ_Mask,ax

    ret

endp

```

Для корректного возврата в режим реальных адресов нужно изменить одну команду в процедуре перенаправления векторов IRQ для R-Mode:

```

init_R_Mode_redirect_IRQ      macro

R_Mode_redirect_IRQ          proc    near

    mov     bx,7008h
    mov     dx,R_Mode_IRQ_Mask      ; Вот эту команду мы используем,
                                   ; вместо MOV DX,0.
    call    redirect_IRQ

    cmp     APIC_presence,1
    jne     rmrirq_1
    call    enable_APIC

rmrirq_1:
    ret

endp

endm

```

Теперь, казалось бы, наш пример должен правильно работать, но MS-DOS приготовил один неприятный "подводный камень". Дело в том, что при повторном запуске нашего примера, при условии, что в нём выполняются какие-либо процессы, длительностью более, чем примерно 2 секунды, контроллер клавиатуры генерирует символ. Если не обработать его должным образом, то клавиатура будет заблокирована, поэтому во всех наших примерах предлагается следующее:

1. Обязательно размаскировать прерывание клавиатуры (IRQ 1).
2. Обязательно разрешать прерывания на время выполнения части программы, работающей в защищённом режиме.
3. Установить обработчик IRQ клавиатуры или хотя бы следующую заглушку:

```

IRQ_1_handler    macro

    push        ax

    in          al,60h           ; AL содержит скан-код клавиатуры, но в
                                ; этом примере он не сохраняется -
                                ; обработчик IRQ 1 работает как заглушка.

    in          al,61H
    mov         ah,al
    or          al,80h
    out         61H,al
    xchg        ah,al
    out         61H,al

    mov         al,20h
    out         20h,al

    pop         ax

    iret

endm

```

Как видите, установка обработчика IRQ клавиатуры свелась к простой замене определяющего его макроса "IRQ_1_handler".

А теперь вашему вниманию предлагается демонстрация обработки прерываний по таймеру. В приведенном ниже примере внесены следующие изменения (по сравнению с предыдущим и с учётом всего, сказанного выше):

1. Введена переменная "timer_count", в которой накапливаются "тики" таймера и ещё одна переменная - "timer_sec" - счётчик секунд. После каждого 18-го "тика" счётчик секунд увеличивается на 1. В качестве часов данный пример не совсем годится, т.к. за одну секунду таймер выдаёт около 18.2 "тиков" (если его дополнительно не программировать), а данный пример предназначен в качестве иллюстрации обработки IRQ и поэтому подсчёт времени здесь упрощённый.
2. Макрос "IRQ_0_handler" изменён - он считает "тики" таймера. Теперь это не заглушка, а Обработчик Прерывания.
3. Перед тем, как в программе будут разрешены прерывания (командой STI), размаскировывается IRQ 0 (а так же и IRQ 1, для корректной обработки контроллера клавиатуры).
4. В программе приводится простой алгоритм, в котором на экран выводится dd-число, которое в бесконечном цикле увеличивается на 1. При это постоянно проверяется содержимое переменной "timer_count" и:
 - сбрасывается в 0, как только она превышает 18,
 - при этом увеличивается на 1 переменная "timer_sec"
 - и как только она превысит значение 4, производится возврат в R-Mode.

Вот так теперь выглядит обработчик IRQ 0:

```

IRQ_0_handler    macro

    push        ax

    mov         al,20h
    out         20h,al

    pop         ax

    inc         timer_count

```

```
    iret  
  
endm
```

Как видите, всё что он делает - это посылает контроллеру прерываний команду конца прерывания (EOI) и увеличивает значение "timer_count" на 1. И всё! Так просто!

На самом деле, когда вы будете писать **свою** ОС, то, скорее всего, добавите в обработчик IRQ 0 функции подсчёта времени, даты и ещё что-нибудь важное, но даже в таком виде он будет корректно работать.

А вот так в примере разрешены прерывания и реализован алгоритм подсчёта и вывода времени:

; Демонстрация работы прерывания по таймеру

```
    in      al,21h  
    and     al,11111100b      ; Размаскируем прерывания таймера  
                                ; и клавиатуры.  
    out     21h,al  
  
    mov     al,0  
    mov     timer_count,al    ; Сбрасываем наши счётчики  
    mov     timer_sec,al  
  
    xor     eax,eax           ; Число в EAX будет выводиться на экран  
                                ; в бесконечном цикле.  
  
    sti                                ; Разрешаем аппаратные прерывания. Теперь наш  
                                ; бесконечный цикл будет "рваться" таймером и  
                                ; клавиатурой и остаётся только следить за  
                                ; счётчиками.  
timer_demo_start:  
  
    mov     dx,1400h          ; В 20-ю строку, нулевую позицию ...  
    call    put_dd_num        ; ... будет выводиться dd-число.  
    inc     eax               ; А здесь оно увеличивается.  
  
    cmp     timer_count,18  
    jnb     timer_demo_start  ; Выводим число, пока timer_count < 18  
  
    mov     timer_count,0     ; timer_count достиг 18.  
                                ; Сбрасываем его.  
  
    push    ax  
    mov     al,timer_sec  
    inc     al  
    mov     timer_sec,al  
    mov     dx,1410h  
    call    put_db_num        ; Выводим число секунд.  
  
    pop     ax  
  
    cmp     timer_sec,4  
    jbe     timer_demo_start  ; Продолжаем цикл, если timer_sec < 4  
  
    jmp     Return_to_R_Mode
```

Осталось добавить, что этот пример, как и предыдущий, правильно реагирует на исключения - выводит его номер, параметры и возвращается в R-Mode, так что можете смело экспериментировать

- компьютер не зависнет.

Исходный текст примера вы можете скачать здесь: [examp_6.asm](#), [pmode_6.lib](#) и [examp_6.com](#) в архиве [examp_6.zip](#) (9594 байт).

[Оглавление](#)

Вопросы? Замечания? Пишите: sasm@narod.ru

Copyright © Александр Семенко.

