**Прерывания в защищённом режиме:**

Глава 1. Основы работы прерываний. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_int/main.htm;j=false;s=1280*1024;d=24;rand=0.093715411673299

        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 позоляет 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 вектором 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 находится 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.

**Прерывания в защищённом режиме:**

Глава 2. Дескрипторы прерываний. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_int/chap_1.htm;j=false;s=1280*1024;d=24;rand=0.36377537583460884

        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 проэтому 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.

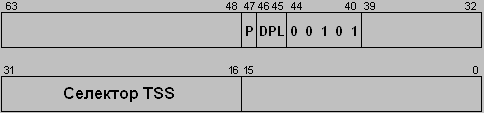
dw 0

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

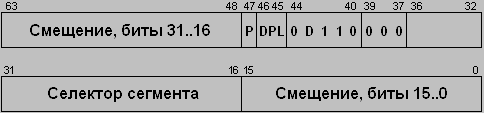
dw selector ; Селектор сегмента кода

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

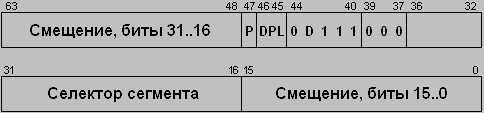
dw selector ; Селектор сегмента кода

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.

Глава 3. Исключения. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_int/chap_2.htm;j=false;s=1280*1024;d=24;rand=0.6685732329190974

        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 неприсутствующего 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 с 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 UD21 |
| 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 точки2 |
| 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 | (Нуль) Обращение к пямяти3 |
| 12h | #MC | Проверка оборудования | Abort | Нет | Наличие кодов и их содержимое зависит от модели4 |
| 13h | #XF | Исключение плавающей точки в SIMD | Fault | Нет | Команды SSE и SSE25 |
| 14h-1Fh | - | Заререзвировано 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](http://sasm.narod.ru/docs/pm/pm_app/app_1/df.htm).   
        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](http://sasm.narod.ru/docs/pm/pm_app/app_1/app_1.htm) с ссылками на подробные описания каждого исключения, а в этой главе она предназначена для ознакомления.

Глава 4. Таблица дескрипторов прерываний. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_int/chap_3.htm;j=false;s=1280*1024;d=24;rand=0.4385424901513394

        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 P 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 специальном 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.

http://sasm.narod.ru/docs/pm/pm_img/pmint_41.gif  
**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.

Глава 5. Аппаратные прерывания. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_int/chap_4.htm;j=false;s=1280*1024;d=24;rand=0.5051234211660491

        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 прерываних 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

;--------------------------------------------------------------------------

Глава 6. Установка IDT в программе. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_int/chap_5.htm;j=false;s=1280*1024;d=24;rand=0.3569585987931235

        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 исходнике - 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 исходника, 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 & db 0

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

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 зацикливание - 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.

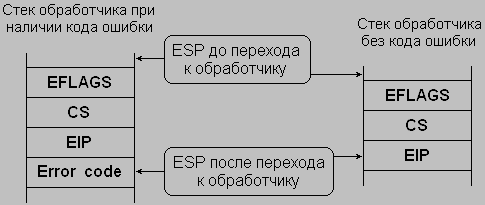
        В этом примере определена IDT из 32h (т.е. 50) дескрипторов. Для примера - этого вполне хватает, для других программ - может не хватить, но главное, чтобы вы знали о возможности увеличить/уменьшить IDT простой заменой значения макроса idt\_descr\_n.   
        Для программных прерываний в примере определено 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](http://www.sasm.narod.ru/docs/pm/pm_int/examp_4.zip) (7230 байт)

Глава 7. Заглушки. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_int/chap_6.htm;j=false;s=1280*1024;d=24;rand=0.8165042527686044

        Заглушками 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 заглушек 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"](http://sasm.narod.ru/docs/pm/pm_app/app_1/app_1.htm)).

        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 сохранятся 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 видеопамяти 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,1ah

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](http://sasm.narod.ru/docs/pm/pm_int/examp_5.zip) (9177 байт).

Глава 8. Обработчики аппаратных прерываний. http://top.list.ru/counter?id=213421;js=13;r=http%3A//sasm.narod.ru/docs/pm/pm_int/chap_7.htm;j=false;s=1280*1024;d=24;rand=0.3403997952917459

        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 мультизадачность 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 дисковода 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-а

and al,0feh ; FEh = 11111110b - сбрасываем 0-й бит.

out 21h,al ; Записываем маску в контроллер. Таймер разрешён.

        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 размаскировывать 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 выводится 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 выводится dd-число.

inc eax ; А здесь оно увеличивается.

cmp timer\_count,18

jb 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](http://sasm.narod.ru/docs/pm/pm_int/examp_6.zip) (9594 байт).