# Архив статей wasm.ru. Том 10: Защищённый режим, алгоритмы и платформы

Собрание русскоязычных статей сайта wasm.ru о программировании, ассемблере, реверс-инжиниринге и компьютерной безопасности. Том 10 из 11. Разделы: 19. Защищенный режим; 20. Алгоритмы; 21. Консоли и КПК; 22. Байт-код; 23. Linux/Unix. Все приложенные файлы находятся в wasm_files.zip.

# 19. Защищенный режим

# Процессор Intel в защищенном режиме #1

**Автор:** Broken Sword

**Дата:** 2002-10-15

Рекомендуется читать запоем от начала и до конца, но вдумчиво…

**Prelude**

Здравствуй, уважаемый подписчик! Перед тобой первый выпуск рассылки. Несколько слов о том, о чем здесь вообще пойдет речь в дальнейшем. Речь, как ты уже догадался и без подсказок, пойдет о программировании в защищенном режиме процессоров Intel, причем даже не столько о программировании, сколько о самой архитектуре, о сущности PM и др. глубоких материях :). (хотя не пугайся, последнего не будет, упаси боже! А тех, кому без высоких материй не живется на свете – попрошу обратить свой взор на «Низкоуровневое программирование для dzенствующих».

Сразу давай договоримся: защищенный режим и программирование под Windows – это две большие разницы. Windows – это ОС, защищенный режим – режим работы процессора. Это все равно что программирование в реальном режиме называть как-нибудь на манер «программирование под Нортон Коммандер», согласись – глупо, но однако почему то это укоренившееся «в народе» мнение. Итак, никакого виндоуса не будет!

Ты, конечно же, согласен, что реальный режим – прошлый день (или, даже точнее – прошлый век). Но задумывался ли ты о том, что и защищенный режим, как это не прискорбно, остался позади ?.. Год с небольшим назад Intel анонсировала процессор с принципиально новой архитектурой – Itanium; все предыдущие процы (до Itanium-a) были аккуратно сложены в одну большую коробку в кучу (причем, вообще как то даже совсем абстрагировавшись от каких-либо различий между 8086 и Pentium-ом IV), сверху поставлено клеймо – IA32 (Intel Architecture 32) и все это чудо благополучно похоронено…Конечно же, все не совсем так, IA-32 будет жить еще долго (по крайней мере, у нас :).

Первая и главная проблема, с которой мне пришлось сразу же столкнуться: С ЧЕГО НАЧАТЬ? Казалось бы, проблема на ровном месте, однако это не так. Если ты уже довольно неплохо шаришь в защищенке (а таких я не встречал почему то вообще), то думаю, согласишься. Как выразился небезызвестный тов. Зубков: «… управление защищенным режимом в современных процессорах Intel – это самый сложный раздел программирования…», и можешь поверить ему на слово, однако «нет ничего невозможного для человека с интеллектом».

Вся проблема в том, что защищенка – это замкнутый круг, откуда бы я ни начал – останется что-то позади, и впереди… и еще где нибудь сбоку. Но я буду стараться.

Главная причина, побудившая меня к созданию данной рассылки – полное отсутствие русскоязычной более мене толковой документации и учебной литературы о PM. В Зубкове, конечно, все хорошо, но поднимите руку те, для кого PM так же прозрачен и чист как RM? Кто не пытался вникнуть сам, а учился только по известным книжкам, для того PM – темный лес.

Вся «гадость» еще и в том, что в изучении защищенного режима ну никак не обойтись без сухих и нудных описаний полей дескрипторов, структур системных сегментов и т.п., и в конце концов кажется, что вообще вся рассылка будет являть из себя скучное руководство пользователя. Я, по мере своих возможностей, постараюсь преподносить материал в понятном и доступном для ВСЕХ виде, избегая чисто справочных данных (где это возможно). Реализовать сию затею будет не так то просто (особенно поначалу), но если у вас есть хоть какие-то познания в области реального режима Intel – будет проще всем.

Ну и в заключении хочется отметить, что по мере выхода новых выпусков будут опубликовываться интересные статьи из журналов по данной теме (ко всем сразу просьба – у кого есть такая инфа (итересуют только «частные» статьи, желательно на английском, а не ксероксы с книжек Рудаков&Финогенов) – прошу высылать мне на мыло (см. внизу).

Let's go

**Организация памяти в защищенном режиме.**

Что вообще такое защищенный режим и почему он так называется? То, что я сейчас скажу – звучит очень просто, но в то же время – это ОЧЕНЬ ВАЖНО ПОНЯТЬ!

Дело в том, что реальный режим процессора Intel – однозадачная среда, в данный момент времени в ней может выполняться ТОЛЬКО ОДНА, конкретная задача. Безусловно, можно сэмулировать многозадачность и в реальном режиме, но все дело в том, что именно в защищенном режиме вся многозадачность реализована АППАРАТНО, это важный момент, и, по сути, является ЕДИНСВТЕННЫМ принципиальным отличием между режимами. Не правда ли, это шокирует? А ты думал, что PM это монстр с дескрипторами между голов и селекторами вместо рук? Ты правильно думал, но до них дело дойдет, не волнуйся…

Для начала рассмотрим общие положения об организации памяти в PM. В процессорах Intel организацию памяти разделяют на две части: сегментация (segmentation), и страничная организация (paging).

Сегментация позволяет изолировать модули кода, данных, стека и позволяет работать нескольким задачам и программам на одном процессоре (как говорится, multitasking) и не конфликтовать между собой – это и есть по большому счету, вся революция по сравнению с 8086.

Страничная организация памяти, вообще говоря, сложнее и громозще сегментации, поэтому во многой литературе ее просто опускают, но я бы особо хотел обратить твое и свое внимание именно на нее (хотя бы потому, что у такой организации больше возможностей, и именно на ней работает Мастдай).

В процессоре нет такого бита, который бы четко отвечал за переключение между этими двумя режимами. Элементы сегментной организации, в любом случае, присутствуют всегда. А вот уже использовать страничную организацию или нет – выбор за нами (за это отвечает флаг PG, бит 31 регистра CR0). Не пугайся, CR0 – это обычный регистр (такой же, как AX, BP и т.д.), подробнее поговорим о нем чуть позже.

Сегментация – это механизма разделения адресного пространства процессора (по-русски: память, которую «видит» процессор) на отдельные защищенные друг от друга кусочки (сегменты). У новичка сразу возникнет вопрос: а от чего защищать то :)? Очень важный момент, но пока не бери в голову, скоро он отпадет сам собой.

Непосредственно сегмент может содержать в себе код, данные, стек, системные структуры данных (TSS, LDT, но о них позже). Если запущено сразу несколько программ :), то каждой программе принадлежит своя, личная группа сегментов.

Итак, что мы имеем. Память, разделена на группы сегментов, у каждой группы есть свой владелец (программа). Программы вынуждены сидеть в оперативе, прижавшись друг к другу, как цыплята в инкубаторе, вследствии чего они потихоньку друг друга ненавидят и не подпускают чужие проги к сегментам из своей группы. При любой попытке пресечь границы вылазит некое подобие «руки правосудия» и нажимает на кнопку ALARM. Вследствие чего вдруг откуда ни возьмись возникает исключение #GP (General Protection) и все проги падают ниц перед ним, а виновника изгоняют вон :) Но что то я увлекся, все эти прекрасы ждут нас в будущем…

Нет, ну вообще, как ты относишься к изложению в таком духе?

Для того, чтобы обратиться к любому байту в любом сегменте в памяти мы должны сформировать ЛОГИЧЕСКИЙ АДРЕС (aka дальний указатель).

Что же оно из себя такое представляет, этот логический адрес? А представляет оно селектор и смещение, и больше ничего! Селектор – это УНИКАЛЬНЫЙ идентификатор сегмента. У КАЖДОГО СЕГМЕНТА ЕСТЬ СВОЙ СЕЛЕКТОР! Это нужно запомнить раз и навсегда!!! Скажу по секрету, что селектор содержится в сегментом регистре (да да, те самые DS, CS, ES …). Селектору будет посвящена отдельная глава, а пока запомни только то, что «… у каждого сегмента он свой». Зная селектор и зная смещение мы можем получить ЛИНЕЙНЫЙ АДРЕС – место, где РЕАЛЬНО расположен нужный нам байт. Как его получить – пока сказать не могу, потому как придется вплетать еще одно СУПЕРВАЖНОЕ(!) определение – ДЕСКРИПТОР (структура, описывающая сегмент, его паспорт, свидетельство о рождении, водительские права и вообще всю подобную бюрократию).

Существует также такое понятие, как физическое адресное пространство процессора. Не следует на нем особо заострять внимания, скажу только, что физический адрес – это адрес, который проц может выставить на адресную шину, и в случае СЕГМЕНТНОЙ организации памяти (но не СТРАНИЧНОЙ!) СОВПАДАЕТ с линейным адресом! Вот этот момент в литературе всячески перевирается и искажается, но запомни: в случае СЕГМЕНТНОЙ организации памяти ЛИНЕЙНЫЙ адрес ВСЕГДА СОВПАДАЕТ с ФИЗИЧЕСКИМ. Пока больше ничего.

**Виды памяти в защищенном режиме**

**Простая плоская модель** Действительно, самая простая модель: вся память представляет одно ОГРОМНОЕ адресное пространство, никакого механизма распределения, никаких сегментов, ничего нет! Пустыня! Нет, не совсем конечно… Должно быть МИНИМУМ два дескриптора (черт! Опять эти дескрипторы…Это сильно усложняет мне задачу… ладно, продолжим дальше, прими во внимание только то определение которое я давал про дескрипторы выше, и еще дополнительно – в дескрипторе есть поля, в которых указаны начало и конец сегмента). Один из этих дескрипторов ДОЛЖЕН описывать сегмент кода (с началом в 0 и лимитом в 4 Гб), второй – сегмент данных (также с началом в 0 и лимитом в 4 Гб!!!). Как же так? Они же «накладываются друг на друга»? Ну вот так вот, говорю же, никакой защиты, ничего не застраховано, «рука правосудия» спит в гробу. **Защищенная плоская модель** Все отличие от предыдущей модели – база и лимит кода и данных уже не совпадают, и здесь уже #GP может проявить себя в самом разцвете… Более того! Если включить флажочек PG (страничная адресация), мы получим ту самую модель памяти, в которой работает всеми нами любимый мастдай – защищенная плоская модель с страничной адресацией . Но здесь нам уже понадобиться минимум 4 сегмента: для кода и данных на уровне привилегий 3, + для кода и данных на уровне 0 (многие из вас слышали, что мастдай использует только эти два уровня привилегий, оставляя 3 для пользовательских прог, а 0 – для ядра и особ, «особо приближенных к императору»). **Мульти-сегментная модель**

Данная модель использует все возможности проца «на полную катушку», позволяет аппаратно защищать код, структуры данных, задачи и программы друг от друга.

На сегодня хватит, итак много чего нужно запомнить и уловить, в следующем выпуске я хочу подробно остановиться на СТРАНИЧНОЙ АДРЕСАЦИИ, а пока – пиши мне письма, что непонятно, где возникли пробелы, и хотя весь первый выпуск рассылки – по сути сентиментальная чушь, но новичку все эти знания ОЧЕНЬ помогут в понимании на следующих этапах. ПисАть прошу сюда: brokensword@ukr.net и, плз, без речевых (wav) и графических (bmp) посланий.

# Процессор Intel в защищенном режиме #2

**Автор:** Broken Sword

**Дата:** 2002-10-15

В предыдущем выпуске ты в самом первом приближении ознакомился с видами организации памяти в защищенном режиме. Напомню, что существует всего 2 вида: сегментная (segment) и страничная (paging) модели. Также я обещал в этом выпуске более подробно остановиться на страничной организации. На самом деле еще рановато… Давай лучше до конца разберемся с сегментной организацией, потом будет намного проще разобраться со страничной…

Итак, что мы имеем: программа (вернее, программы) загружены в оперативную память, каждая программа владеет группой сегментов (данные, код, стек). Теперь возникает вопрос: откуда же процессор знает, где какой сегмент? Где начало сегмента? И где его конец? И что это за сегмент: данных? кода? А может быть, стека? Или, может быть, всего разом? Ответить на этот вопрос может только одна структура данных: ДЕСКРИПТОР. Повторите это слово про себя несколько раз, запомните его. Напишите на бумажке 10 раз :). ДЕСКРИПТОР – это структура, ОПИСЫВАЮЩАЯ сегмент. Если человек – это сегмент, то паспорт – это его дескриптор. У каждого человека он свой собственный. И разница лишь в том, что человеку паспорт (дескриптор) выдается с 16 лет (на Украине), а у сегмента он с момента рождения.

**Сегментный дескриптор**


Вот он, наш красавец! Посмотри на него повнимательнее, включи зрительную память на максимум… Я его вообще около получаса вымалевывал, и вот какой он получился ненаглядный!

Теперь распишу во всех пикантных подробностях значения полей дескриптора (хочется тебе или нет, но без этого, к сожалению, НИКУДА дальше не уедешь, придется тебе их всех запомнить):

(Смотри одним глазом на фиолетовые слова, другим – ищи их на картинке :) )

*Адрес базы* : адрес нулевого байта описываемого сегмента в 4 Гб линейном адресном пространстве (т.е. адрес, с которого начинается сегмент). Процессор собирает в кучу три поля зеленого цвета :) и образует единый 32-х битный адрес. Почему так убого? Потому что вообще говоря, процессор Интел представляет собой штопанно-перелатанного уродца (достаточно взглянуть на дескриптор). И этому есть несколько причин. Единственное, что могу добавить – это не самое ужасное. Что еще хотелось бы отметить по поводу адреса базы сегмента – желательно он должен быть кратен 16 (так проц при обращении к описываемому сегменту и формировании адреса будет быстрее соображать).

*Лимит сегмента* : определяет размер сегмента. Опять же, проц собирает в кучу поля красного цвета, и формирует конечный, 20- битовый адрес. Реальный лимит сегмента зависит от бита гранулярности (G-granularity);

- если бит гранулярности сброшен (0), то 20-битное значение и будет тем самым лимитом сегмента
- если бит гранулярности установлен (1), то всё 20-битное значение автоматически увеличивается в 1000h раз, т.е. если при G=0 мы измеряем размер в байтах, то при G=1 – в 4Кб единицах, (см. мультфильм «48 попугаев»).

Например, если G=1 и поле «Лимит сегмента» = 0000Fh (15 байт), то реальный лимит (читай – размер) данного сегмента равен 0Fh*1000h=0F000h (около 61 тысячи байт!).

Следует отметить, что если поле «Лимит сегмента» содержит значение равное 0, то это значит, что описуемый сегмент имеет размер в 1 байт (а не ноль!) при G=0, и размер в 4Кб при G=1. Т.е. сегмент никак не может иметь нулевую длину, минимум – 1 байт, максимум – 0xFFFFFh * 4Кб = 4 Гб.

Есть еще один бит, от которого зависит смысловое значение этого поля. Бит направления роста сегмента (B-big));
- если этот бит сброшен (0), то разрешены все смещения от 0 до лимита
- если установлен (1) – то все, кроме от 0 до лимита Но этот бит B несколько затуманивает понятие лимита, поэтому пока не бери в голову :)

В противном случае (если мы попытаемся обратится за пределы лимита) – возникнет исключение главной защиты (#GP) – страшная вещь! Вообще говоря, лимиты для этой цели и придуманы – отслеживать обращения в недоступные адресные пространства.

Еще один момент, который нигде в документации явно не указан. Лимит сегмента отсчитывается ОТ БАЗЫ СЕГМЕНТА, а не от нуля !!! Это совсем не ОЧЕВИДНО! (по крайней мере, я долго заблуждался…)

*Тип:*
Определяет тип сегмента (или шлюза, что такое шлюз – узнаете потом), определяет права доступа к сегменту и направление роста сегмента (помните бит B). Значение этого поля зависит от значения поля «Тип дескриптора» (S-descriptor type). Значение этого поля различно для разных типов сегментов (кода, данных и системного)

*S (descriptor type) – флаг «тип дескриптора»:*
Означает только одно: если сброшен (0), то описуемый сегмент – системный, если установлен (1) – это сегмент данных или кода. Подробности – в след. выпуске. И вообще привыкай: если что то непонятно – узнаешь об этом в следующем выпуске 100%. Или если не можешь спокойно спать – пиши мне (см. мыло винзу).

*DPL (descriptor privilege level) – уровень привилегий дескриптора:*
Определяет уровень привилегий сегмента. Т.к. это поле – двухразрядное, то соответственно может принимать только четыре различных значения (от 0 до 3). Самый крутой – нулевой уровень привилегий (ядро ОС). Это поле нужно для контроля за доступом к сегменту. Во всех подробностях распишу его назначение в последующих главах рассылки. Пока забудь о нем.

*P (segment present flag) – флаг присутствия сегмента:*
Если установлен, значит сегмент присутствует непосредственно в памяти; если сброшен – сами догадайтесь. Зачем оно нада: все видели и трогали мастдай, все знают что такое своп-файл. Вообщем, этот флаг предназначен для организации работы при использовании страничной адресации (дело в том, что когда в сегментный регистр грузят селектор на дескриптор, в котором сброшен этот бит – возникает исключение #NP (segment-not- present exeption)), поэтому, если отлавливать это самое #NP можно вовремя подгрузить новую страницу в ОП из файла подкачки. Страничная адресация ждет нас впереди!

*G (Granularity) – флаг «гранулярности»:*
См. «Лимит сегмента». Еще раз повторю - этот флаг влияет на лимит сегмента: если сброшен – лимит измеряется в байтах, если установлен – в 4 Кб единицах. Еще что запомни раз и навсегда:

на адрес базы этот флаг НИКАКИМ БОКОМ НЕ ВЛИЯЕТ! Только на лимит! База ВСЕГДА измеряется в байтах! Clear? )

*AVL (Available and reserved bits) – зарезервировано:*
Это два битика (21-20 во втором двойном слове). Они вообще не стоят того, чтобы на них заострять внимание, но все же может кому и пригодиться: битик 20 может использоваться как угодно по вашему усмотрению :), битик 21 ВСЕГДА ДОЛЖЕН БЫТЬ РАВЕН НУЛЮ! ИНАЧЕ ГОВРОЯ - РУКАМИ НЕ ТРОГАТЬ!

Остался последний битик. Двадцать второй. D/B. Довольно мутный. Зависит от типа сегмента (в зависимости от этого он называется либо D, либо B). Учитывая то, что про него уже было сказано при описании лимита сегмента, следует отметить еще вот что:

Определяет разрядность сегмента. Если установлен – значит сегмент 32-х разрядный, если сброшен – 16-ти. Это общий случай. Теперь частные:

- **Сегмент кода**. Для сегмента кода данный флаг называется D и устанавливает длину по умолчанию для эффективных адресов и операндов в сегменте. Если установлен – то в сегменте допустимы 32-х битные адреса и 32-х битные ИЛИ 8-ми битные операнды; Если сброшен – 16-битные адреса и 16-битные ИЛИ 8-ми битные операнды. Но никто не мешает нам использовать префикс 66h (для изменения разрядности операндов) и префикс 67h (для изменения разрядности адресов).

- **Сегмент стека**. Для сегмента стека данный флаг называется B (big) флаг и устанавливает длину УКАЗАТЕЛЯ СТЕКА (регистр ESP) для команд push, pop и call. Если установлен – используется 32-х разрядный указатель стека (т.е. регистр ESP использован по максимуму), если сброшен – то используется только 16 бит (регистр SP, уже без буковки E).

Вот и все на сегодня. Кстати, если ты что то упустил или недопонял – ОБЯЗАТЕЛЬНО пиши на brokensword@ukr.net! Я об этом просил еще в первом выпуске и пришло аж ни одного письма! Ну, надеюсь, это потому, что все было ясно, как никогда ).

# Процессор Intel в защищенном режиме #3

**Автор:** Broken Sword

**Дата:** 2002-10-15

Рассылка постепенно набирает обороты! Я по прежнему жду не дождусь писем с вопросами… Вариантов здесь может быть только два: или вам все сходу понятно, или вообще ничего не понятно… Второй вариант как то не особо впечатляет на дальнейшее продолжение. Хотя, пока слишком мало материала, и спрашивать, по сути, не о чем.

**Таблица дескрипторов**

Итак, к чему мы пришли: оказывается, сегменты не раскиданы по памяти как попало и непонятно где, теперь мы с уверенностью можем сказать где какой из сегментов начинается, где заканчивается, что это за сегмент (код, данные или стек), вся эта информация хранится в ДЕСКРИПТОРЕ. Казалось бы, чего нам не хватает для полного счастья? Все же вроде ясно и понятно и не о чем здесь больше говорить! Однако, есть еще кое что… Один вопрос, который уже должен был загореться в твоем подсознании… ГДЕ ЖЕ НАХОДИТСЯ САМ ДЕСКРИПТОР ???!!! :)

Находиться он может в трех местах:

• Глобальная таблица дескрипторов (GDT - Global Descriptor Table)
• Локальная таблица дескрипторов (LDT - Local Descriptor Table)
• Таблица дескрипторов прерываний (IDT – Interrupt Descriptor Table)

Эти таблицы находятся в оперативной памяти (там же, где и собственно сами программы), а не в процессоре, жестком диске или куда еще могут завести тебя фантазии… Поэтому, очевидно, что строить ТАБЛИЦЫ ДЕСКРИПТОРОВ придется «руками», нам самим. И вообще, с этого момента пора привыкать – в защищенном режиме ВСЕ НУЖНО СТРОИТЬ САМОМУ. В этом и заключается сила! :)

Несколько слов по поводу первых двух таблиц (на третьей (IDT) мы остановимся в главе посвященной прерываниям в защищенном режиме).

**Глобальная таблица дескрипторов (GDT):**

ЗАПОМНИ: КАЖДАЯ ОС ДОЛЖНА ИМЕТЬ ОДНУ ТАБЛИЦУ GDT!!! Без нее – никуда! Ей (таблицей) могут пользоваться ВСЕ (!) программы и задачи системы. Что значит «пользоваться таблицей дескрипторов»? Это значит, хранить в ней СВОЙ дескриптор.

Таблица GDT – сама по себе НЕ СЕГМЕНТ! Это структура данных в линейном адресном пространстве (в памяти). НАЧАЛО таблицы GDT храниться В РЕГИСТРЕ GDTR! Регистр GDTR – это самый обыкновенный регистр, такой же обыкновенный, как EAX, EIP, ES, только вот его функция заключается не в хранении каких-то промежуточных данных, а в хранении фиксированного числа – НАЧАЛА ТАБЛИЦЫ GDT. Называние регистра GDTR запомнить очень легко: GDT – это таблица, R – регистр. Не правда ли, пока все просто? )

Да, кстати, не упусти важный момент: мы должны сделать все так, чтобы начало таблицы GDT в памяти было кратно 8. Это связано с архитектурой и так проц быстрее соображает при обращении к таблице.

GDTR:

```
32-битный линейный базовый адрес|16-битный лимит таблицы
```

Вот он, регистр GDTR. Весит 48 бит. Как видишь, он содержит не только адрес начала таблицы GDT в памяти, а еще и ее лимит. Лимит таблицы – 16-битное значение, показывает величину таблицы в байтах + 1. (т.е. все как и в случае с лимитом сегмента: если лимит таблицы в GDTR

равен 0, то на самом деле это означает что реально (в памяти) лимит равен одному байту).

Как ты уже знаешь, сегментный дескриптор ВСЕГДА занимает 8 байт (2 двойных слова). Если уже забыл – посмотри на картинку из 2 выпуска. Следовательно, лимит таблицы дескрипторов – величина, равная 8N-1 байт. Но и N – тоже не резиновая величина. Скоро узнаешь каков предел этого самого N (т.е. сколько всего дескрипторов может содержать таблица)

Первый дескриптор в GDT не используется и называется «нулевой дескриптор» (null descriptor). При обращении к памяти через этот дескриптор возникает уже знакомая нам «рука правосудия» с #GP, и все опять вынуждены пасть ниц :)

Поэтому первый дескриптор в GDT – РУКАМИ НЕ ТРОГАТЬ!

Загрузить/считать значение регистра GDTR можно командами LGDT/SGDT. По умолчанию (т.е. после нажатия на кнопку Reset или включения компа) база GDT равна нулю, а лимит – FFFFh, т.е. фактически по умолчанию выделено максимум места, под FFFFh/8 = 8191 дескрипторов (ну минус один, учитывая null descriptor).

**Локальная таблица дескрипторов (LDT):**

В отличии от GDT совершенно не ОБЯЗАНА присутствовать вообще. И в то же самое время, ПО ЖЕЛАНИЮ, их можно развести великое множество (GDT ДОЛЖНА БЫТЬ только одна). Каждая задача может иметь свою собственную LDT, в то же время несколько задач могут использовать одну LDT на всех.

LDT – это СЕГМНЕТ! (GDT – структура данных). Это принципиально! И вот почему: т.к. LDT – это сегмент, то значит у нее тоже есть свой дескриптор! И где бы вы думали? Да все в той же GDT!!! Да да, звучит несколько странновато – у таблицы дескрипторов есть свой дескриптор!! Но, то ли еще будет! :)

Так же, как и у GDT у LDT тоже есть свой регистр – LDTR. В отличии от GDTR этот регистр, помимо инфы про базу и лимит LDT, содержит еще одно поле – сегментный селектор. Пока что не вникай, просто запомни слово – СЕЛЕКТОР.

LDTR:

```
Сегм. Селектор (16 бит)
32-битный линейный базовый адрес
16-битный лимит сегмента
```

Инструкции LLDT и SLDT позволяют писать/читать регистр LDTR. Точно так же, при reset-е значение базы в LDTR падает в ноль, а лимит – в FFFFh.

Ну и напоследок - ВНИМАНИЕ! КОНКУРС!!! )

Перед тобой пример структуры, которая являет собой таблицу GDT (вместо ИКС-ов значения, которые мы еще не рассматривали, поэтому не смотри на них); Тебе предстоит найти РЕАЛЬНЫЕ ЗНАЧЕНИЯ базы и лимита для каждого из описуемых семи сегментов (именно линейные адреса в памяти). Для одного из них дана подсказка в комментарии. Победителей ожидают ценные призы, а того, кто первым пришлет ответ – КВАРТИРА В МОСКВЕ! Поторопитесь!!!

Чтобы было проще – открой картинку с дескриптором и смотри одним глазом на нее, другим - на значения. Да, и помни: смотреть картинку нужно «задом-наперед» (т.е. сначала младшие байты, потом старшие):

```
GDT:
; нулевой дескриптор (ОБЯЗАТЕЛЬНО ДОЛЖЕН ПРИСУТСВОВАТЬ, НО РУКАМИ НЕ ТРОГАТЬ!)
0.                      db    8 dup (0)

; Сегмент с  базой в 0 и лимитом в 1235h (не забывай про +1 к лимиту!)
; Линейный адрес базы = 0, линейный адрес лимита = 0 + 1235h = 1235h
1. Descr_code           db    34h,12h,00h,00h,00h,XXh,0X000000b,00h
2. Descr_data           db    0C8h,0Dh,36h,12h,00h,XXh,0X100000b,00h
```

```
3. Descr_stack          db    0FFh,00h,00h,20h,00h,XXh,1X000000b,00h
4. Descr_code2          db    0DEh,0BCh,01h,20h,10h,XXh,0X001010b,00h
5. Descr_data2          db    00h,00h,00h,00h,00h,XXh,0X000000b,10h
6. Descr_stack2         db    01h,00h,10h,00h,00h,XXh,0X000001b,10h
7. Descr_LDT            db    04h,00h,00h,00h,00h,XXh,1X000000b,20h

GDT_limit                =    $-GDT

GDTR                    dw    GDT_limit-1
                        dd    ?
```

Дополнительные задания:

1. Найди ошибку! Один из дескрипторов содержит ОШИБКУ!
2. Покажи, как правильно нужно загрузить регистр GDTR.

Ответы присылать сюда: brokensword@ukr.net в таком виде

1. Линейный адрес базы = 0
Линейный адрес лимита = 1235h
2. . . .

Слово «линейный адрес» можешь опустить :)

Ариведерчи

# Процессор Intel в защищенном режиме #4

**Автор:** Broken Sword

**Дата:** 2002-10-15

**FAQ:**

• Непонятно высказывание из выпуска №2: «…если G=1 и поле «Лимит сегмента» = 0000Fh (15 байт), то реальный лимит (читай – размер) данного сегмента равен 0Fh*1000h=0F000h (около 61 тысячи байт!)». Ты же сам говорил, что «реальный» лимит сегмента (его размер) при G=1 на 4Кб больше (а при G=0 - на один байт) (т.е. к 0F000h нужно еще прибавить 1000h, ведь так??!!!)

Абсолютно верно. Ситуация такова: чтобы найти адрес последнего байта сегмента (его лимит), нам нужно в зависимости от бита G к соответствующему значению поля дескриптора (то самое, красного цвета) ПРИБАВИТЬ 1 или 1000h. Просто я об этом говорю чуть дальше по тексту. Т.е. если поле «Лимит» дескриптора содержит число 12345h, а бит G=1, то значит лимит равен 12345h*1000h + 1000h.

Теперь, если поле «Лимит» дескриптора равно 12345h, а бит G=0, то лимит будет равен 12345h*1 + 1.

Вывод, который нужно сделать из всего вышесказанного:

Значение поля дескриптора «Лимит» и РЕАЛЬНЫЙ лимит сегмента – это разные величины (которые при любом раскладе, отличаются друг от друга на 1 либо на 1000h, в зависимости от флага G)

• «Лимит таблицы – 16-битное значение, показывает величину таблицы в байтах + 1 (т.е. все как и в случае с лимитом сегмента: если лимит таблицы в GDTR равен 0, то на самом деле это означает что реально (в памяти) лимит равен одному байту). А далее по тексту: следовательно, лимит таблицы дескрипторов – величина, равная 8N-1 байт. Так минус один или плюс, определись уже.

Вот по данному вопросу хочу публично извинится, на самом деле, конечно же, «показывает величину таблицы в байтах – 1» (в том контексте, в котором я сказал). Просто я имел ввиду, что надо «прибавить» единицу и мы получим РЕАЛЬНОЕ значение лимита :). Извиняюсь. В таких «туманных» вопросах надо быть особо внимательным.

Еще раз: лимит таблицы дескрипторов (РЕАЛЬНЫЙ) ВСЕГДА (!) на единицу больше, чем соответствующее поле регистра GDTR (LDTR). От флага гранулярности G РЕАЛЬНЫЙ лимит НЕ ЗАВИСИТ, поэтому всегда больше ТОЛЬКО НА ЕДИНИЦУ.

**ВНИМАНИЕ! ПРЕДВАРИТЕЛЬНЫЕ РЕЗУЛЬТАТЫ КОНКУРСА!**

Ответы прислали уже два человека, с чем их можно и поздравить. Правда, ни один из не получит ключей от квартиры, потому как не все ответы совпадают с реальными… :). В частности, почему то оба участника решили, что ошибка кроется в 5 дескрипторе (вероятно, сбили с толку нули), но на это и рассчитано! :) На самом деле, лимит МОЖЕТ равняться нулю (в таком случае сегмент займет в памяти ОДИН байт, ничего недопустимого здесь нет); еще многие совершенно забыли про существование бита G и принимали лимит во всей его

ложной личине :) Еще почему то напрочь забыли про старшие биты поля «база» и «лимит» и вообще не обращали на них никакого внимания. Но времени – предостаточно! Думайте! :)

Многим из вас безумно лень копаться в битиках и высчитывать, но без этого - ни шагу вперед! После того, как каждый хотя бы попытается это сделать – сами поймете, насколько все будет легче дальше…

Этот 4 выпуск будет совсем коротеньким и нести в себе чисто справочную информацию. В отличии от предыдущего шевелить извилинами не придется вообще… После этого, 4 выпуска, будет недельное затишье, пусть те кто подписался с 2-3 выпуска догонят остальных (двух человек :) и переварят все 4 выпуска, и мы подведем окончательные итоги конкурса! Убедительная просьба к конкурсантам: БУДЬТЕ ВНИМАТЕЛЬНЫ!

Рассмотрим в подробностях поля дескриптора, которые были помечены «хрестиками».

ВСЕ, О ЧЕМ ПОЙДЕТ РЕЧЬ НИЖЕ, СПРАВЕДЛИВО ТОЛЬКО ПРИ S=1!!! (S - 12 бит во втором двойном слове, т.е. при таком раскладе дескриптор описывает либо данные, либо код).

11 бит во втором двойном слове (он же четвертый бит поля «Тип») показывает, является описуемый сегмент кодом или данными (0 – данные, 1 – код)

Если это сегмент данных, то младшие три бита поля «Тип» (10-8) интерпретируются как бит E (expansion-direction) – рост направления, бит W (write-enable) – запись разрешена и бит A (accessed) – доступен. Я умышлено привожу английские интерпретации, чтобы знатоки языка сами сделали надлежащие выводы без моих «вольных переводов».

Посмотрите мимолетно на табличку, и все станет понятно:

| № | Поле Тип | | | | Тип дескриптора | Описание |
|---|---|---|---|---|---|---|
| | 11 | 10<br>E | 9<br>W | 8<br>A | | |
| 0 | 0 | 0 | 0 | 0 | Данные | Только для чтения |
| 1 | 0 | 0 | 0 | 1 | Данные | Только для чтения, доступен |
| 2 | 0 | 0 | 1 | 0 | Данные | Для чтения/записи |
| 3 | 0 | 0 | 1 | 1 | Данные | Для чтения/записи, доступен |
| 4 | 0 | 1 | 0 | 0 | Данные | Только для чтения, растет вниз |
| 5 | 0 | 1 | 0 | 1 | Данные | Только для чтения, растет вниз, доступен |
| 6 | 0 | 1 | 1 | 0 | Данные | Для чтения/записи, растет вниз |
| 7 | 0 | 1 | 1 | 1 | Данные | Для чтения/записи, растет вниз, доступен |
| | | C | R | A | | |
| 8 | 1 | 0 | 0 | 0 | Код | Только для исполнения |
| 9 | 1 | 0 | 0 | 1 | Код | Только для исполнения, доступен |
| 10 | 1 | 0 | 1 | 0 | Код | Для исполнения/чтения |
| 11 | 1 | 0 | 1 | 1 | Код | Для исполнения/чтения, доступен |
| 12 | 1 | 1 | 0 | 0 | Код | Только для исполнения, подчинен |
| 13 | 1 | 1 | 0 | 1 | Код | Только для исполнения, подчинен, доступен |
| 14 | 1 | 1 | 1 | 0 | Код | Для исполнения/чтения, подчинен |
| 15 | 1 | 1 | 1 | 1 | Код | Для исполнения/чтения, подчинен, доступен |

И вот здесь вас подстерегает один приятный сюрприз: оказывается, нет такого специального типа сегмента, как сегмент стека. Сегмент стека – это ни что иное, как сегмент данных, растущий вниз!!! Не правда ли, вы в шоке? :) Сегмент стека ОБЯЗАТЕЛЬНО должен быть доступен для чтения/записи! Т.е. мы должны учесть это при описании сегмента стека.

ДЛЯ сегмента кода младшие три бита поля «Тип» интерпретируются, как C (conforming) – подчиненный, R (read enable) – чтение разрешено и A (accessed) – доступен. Как видите, сегмент кода может быть доступен как только для исполнения, так и для исполнения/чтения. Для исполнения/чтения он может быть доступен в том случае, когда мы храним какие-либо константы или другие статические данные в памяти, непосредственно в сегменте кода. Ну это и так понятно. А теперь ВНИМАНИЕ! В ЗАЩИЩЕННОМ РЕЖИМЕ В СЕГМЕНТ КОДА ПИСАТЬ НЕЛЬЗЯ! Если ты заметил, даже нет такого битика, чтоб никто и не пытался…

Что это за загадочный бит C (conforming)? Этот относится к уровням привилегий и пока не будем его трогать вообще. А что это за подозрительный бит A (accessed)? Что значит «доступен» или не доступен? А об этом мы поговорим в СЛЕДУЮЩЕМ, очень важном выпуске…

ТЕПЕРЬ, ЕСЛИ S=0 (описуемый сегмент является системным сегментом):

Процессор различает 6 типов системных дескрипторов:

• Дескриптор сегмента «таблица LDT»
• Дескриптор сегмента «состояние задачи» (TSS, о нем позже)
• Дескриптор шлюза вызова (Call-Gate) – как зубная паста :)
• Дескриптор шлюза прерывания (Interrupt-Gate)
• Дескриптор шлюза ловушки (Trap-Gate)
• Дескриптор шлюза задачи (Task-Gate)

В свою очередь, данные 6 дескрипторов делятся на две группы: дескрипторы системных сегментов (первые два) и дескрипторы шлюзов (все остальные).

Самый первый (дескриптор сегмента «таблица LDT») нам уже хорошо знаком, а вот с остальными мы еще не скоро познакомимся…
Вообщем, смотри на таблицу:

| № | Поле «Тип» | | | | Описание |
|---|---|---|---|---|---|
| | 11 | 10 | 9 | 8 | |
| 0 | 0 | 0 | 0 | 0 | Зарезервировано |
| 1 | 0 | 0 | 0 | 1 | 16-битный TSS (свободен) |
| 2 | 0 | 0 | 1 | 0 | LDT |
| 3 | 0 | 0 | 1 | 1 | 16-битный TSS (занят) |
| 4 | 0 | 1 | 0 | 0 | 16-битный шлюз вызова |
| 5 | 0 | 1 | 0 | 1 | Шлюз задачи |
| 6 | 0 | 1 | 1 | 0 | 16-битный шлюз прерывания |
| 7 | 0 | 1 | 1 | 1 | 16-битный шлюз ловушки |
| 8 | 1 | 0 | 0 | 0 | Зарезервировано |
| 9 | 1 | 0 | 0 | 1 | 32-битный TSS (свободен) |
| 10 | 1 | 0 | 1 | 0 | Зарезервировано |
| 11 | 1 | 0 | 1 | 1 | 32-битный TSS (занят) |
| 12 | 1 | 1 | 0 | 0 | 32-битный шлюз вызова |
| 13 | 1 | 1 | 0 | 1 | Зарезервировано |
| 14 | 1 | 1 | 1 | 0 | 32-битный шлюз прерывания |
| 15 | 1 | 1 | 1 | 1 | 32-битный шлюз ловушки |

Вот вы наверно думаете, что это за беспредел такой – чуть что то неизвестное и непонятное – так сразу «с этим мы познакомимся попозже».

Нет, я конечно могу рассказать про дескрипторы шлюзов поподробнее прямо сейчас, просто для понимания это окажет обратный эффект… Остановимся на них попозже, и так мне кажется, информации в четырех выпусках предостаточно! Было бы неплохо бегло перечитать их всех еще раз, с первого. Просто посмотри на все совершенно другими глазами.

Ну все, прощаюсь на неделю (а может и на две)! Больше не буду доставать своими дескрипторами. Служба ответов на ваши вопросы (brokensword@ukr.net) работает круглосуточно.

Ариведерчи (Z)

# Процессор Intel в защищенном режиме #5

**Автор:** Broken Sword

**Дата:** 2002-10-15

**Prelude:** Мда... Чето не везет мне с рассылкой. Вот новый дизайн придумали (спасибо Maverick-у), а он почему то ни перед кем не хочет во всей красе предстать... Самое интересное то, что ко мне, как к подписчику, пришел полноценный, рабочий вариант, и даже в архив специально лазил – все прекрасно открывается. Но вот уже у трех человек траблы, поэтому возвращаю рассылке старый, убогий и огромный прикид... (кстати, огромен он оттого, что клепаю я ее в ворде, а он почти к каждой букве лепит шрифт, размер и цвет, поэтому и весит под мамонта). Может кто подскажет что с этми всем беспределом делать? Интересно, есть ли хоть один читатель у которого не было с пятым выпуском никаких проблем? Кстати, здесь есть 3 важные картинки, поэтому дождитесь их полной загрузки... **FAQ:**

- Чето ты с этими лимитами нездорового напорол в спецвыпуске... Какие то «реальные лимиты», «лимиты в натуре» и т.п. Нельзя ли обойтись без подобных медитаций, просто, как есть на самом деле? Вообще все это напоминает недавний бум с миллениумом: в 1999 году все СМИ как один гудели про то, что новое тысячелетие начнется в 2000 году. Потом посчитали и оказалось, что в 2000 начнется только ПОСЛЕДНИЙ ГОД УХОДЯЩЕГО ТЫСЯЧЕЛЕТИЯ, а новое начнется только в 2001... Вот и я туда же - начал выдумывать какие то "реальные лимиты", к-рые соотвествуют "концу последнего байта в сегменте" и т.п. медитации. Хотел как лучше, получилось как всегда... :( Вот КАК В ШЕСТИ СТРОЧКАХ ВСЕ ВЫГЛЯДИТ НА САМОМ ДЕЛЕ, без вольных импровизаций и медитаций:

```
При G=0:
Адрес последнего байта сегмента = Значение поля «База сегмента» + Значение поля
«Лимит сегмента»
Размер сегмента = Значение поля «Лимит сегмента» + 1

При G=1:
Адрес последнего байта сегмента = Значение поля «База сегмента» + Значение поля
«Лимит сегмента» * 1000h + 0FFFh
Размер сегмента = Значение поля «Лимит сегмента» * 1000h + 1000h
```

Вообще, ВОТ С ЭТОГО и стоило начинать, и на этом и закончить, ато расписал на 2 выпуска + спецвыпуск какой то мути, всех запутал и напугал новичков... Вообще на этой проблеме не стоит сосредотачиваться, нас ждут вещи поважнее.

**РЕЗУЛЬТАТЫ КОНКУРСА:**

Правильные ответы:

```
1. Descr_code           db    34h,12h,00h,00h,00h,XXh,0X000000b,00h
; сегмент с базой = 0 и размером = 1235h

2. Descr_data           db    0C8h,0Dh,36h,12h,00h,XXh,0X100000b,00h
; сегмент с базой = 1236h и размером = 0DC9h

3. Descr_stack          db    0FFh,00h,00h,20h,00h,XXh,1X000000b,00h
; сегмент с базой = 2000h и размером = 100000h

4. Descr_code2          db    0DEh,0BCh,01h,20h,10h,XXh,0X001010b,00h
; сегмент с базой = 102001h и размером = 0ABCDFh

5. Descr_data2          db    00h,00h,00h,00h,00h,XXh,0X000000b,10h
; сегмент с базой = 10000000h и размером = 1

6. Descr_stack2         db    01h,00h,10h,00h,00h,XXh,0X000001b,10h
; сегмент с базой = 10000010h и размером = 10002h
7. Descr_LDT            db    04h,00h,00h,00h,00h,XXh,1X000000b,20h
```

```
; сегмент с базой = 20000000h и размером = 5000h
```

Дополнительные задания:

Ошибка во втором дескрипторе: бит 21 во втором двойном слове дескриптора ДОЛЖЕН ВСЕГДА равняться нулю.

Те, кто знакомы с компиляторами типа TASM без труда смогут загрузить GDTR так:

```
mov eax,offset GDT
mov dword ptr GDTR+2,eax
lgdt fword ptr GDTR
(но вообще это забегая вперед)
```

**Краткое содержание предыдущих серий…** Итак, сначала ты узнал, что программа состоит из сегментов и все они расположены в памяти. Каждый сегмент описывает специальная структура – дескриптор. Дескриптор хранится в специальной таблице. Найти в океане памяти таблицу можно по специальному регистру (GDTR, LDTR). Это как в той сказке: на острове – дуб, на дубе ларец, в ларце – яйцо, в яйце – игла и т.д. Ну и с битом гранулярности вроде разобрались (слава Богу!), теперь пора двигать дальше. **Селектор**

«Все это хорошо и понятно» - скажешь ты,- «но вот что-то я ничего пока не слыхал про сегментные регистры (ну те самые – CS, DS, SS…). Что-то они себя пока никак не проявили, а мне казалось, что именно ОНИ, как никто другие, должны служить нам при обращении к памяти и все такое…»

Если ты заметил, то мы все это время спускаемся вниз по ступенькам:

сегмент в памяти <---- дескриптор <---- таблица дескрипторов …

Следующей ступенью будет СЕЛЕКТОР. Не правда ли, где-то это слово уже встречалось? Так вот: селектор – это 16-битная структура данных (что??!! ОПЯТЬ??!!!...), которая является идентификатором сегмента.

- Боже ж ты мой! Сколько это будет продолжаться??!! У каждого сегмента есть свой дескриптор, мы уже прекрасно знаем где этот чертов дескриптор расположен, а ты нам подсовываешь еще какой-то «селектор»!!!

… Селектор указывает не на САМ сегмент в памяти, а на его дескриптор, в таблице дескрипторов… СЕЛЕКТОР ЖИВЕТ В СЕГМЕНТНОМ РЕГИСТРЕ (CS, DS…).

-Неужели…?! Ну и хрена с того ?

Спокойно!!!! Вот он:

Поле ИНДЕКС (биты 3-15): указывает на один из 8192 дескрипторов в таблице дескрипторов (GDT или LDT). Почему 8192? А какое максимальное число по-твоему влезет в «биты 3-15»? Во-во…

- Подожжи, подожжи… не так быстро… Ведь дескриптор же занимает 8 байт так? А если индекс равен двойке? Так что это, значит, селектор указывает на второй байт дескриптора в таблице или что?

ИЛИ ЧТО! ПРОЦЕССОР УМНОЖАЕТ значение поля ИНДЕКС НА 8 И ДОБАВЛЯЕТ к полученному значению АДРЕС БАЗЫ ТАБЛИЦЫ. Т.е. процессор умножит «двойку» на 8, а потом прибавит значение регистра таблицы – и мы благополучно указываем НА НАЧАЛО ДЕСКРИПТОРА, как не крути!!!

- Как же узнать, из КАКОЙ ИМЕННО ТАБЛИЦЫ дескриптор?

Вот для этого нужен флажок TI (table indicator) (второй бит). Если он = 0, то прибавится значение регистра GDTR (т.е. дескриптор расположен в таблице GDT), если же установлен – LDTR.

- А шо за RPL?

(Requested Privilege Level) Запрашиваемый уровень привилегий… пока его лучше не трогать. Но видишь, пока что, все что нам не знакомо относится к каким то загадочным уровням привилегий…

Теперь внимательно следи: допустим мы ложим в DS число 0000000000110 0 00b. Что это значит? А смотри: сразу разбиваем DS на кусочки (15-3 биты – индекс, 2 – TI, 1-0 – пока лучше не смотрим на них). Индекс равен 6. Значит, шестой по счету дескриптор. А где? В GDT конечно! (TI=0).

Ложим (кладем :) в ES число 0000000001000 1 00b. Восьмой дескриптор! На этот раз в LDT! Разобрались!

- А учитывается ли нулевой дескриптор (null descriptor) при «счете»?

ОБЯЗАТЕЛЬНО! БОЛЕЕ ТОГО! МЫ ДАЖЕ МОЖЕМ ПОЛОЖИТЬ В сегментный регистр (DS, SS…) СЕЛЕКТОР С ПОЛЕМ ИНДЕКС и TI РАВНЫМИ 0!!! Т.е. фактически мы выбираем НУЛЕВОЙ ДЕСКРИПТОР В ТАБЛИЦЕ GDT!!

-Но это же невероятно и невозможно!!!! :)

Возможно… Ничего страшного не произойдет до тех пор, пока мы не ОБРАТИМСЯ К ПАМЯТИ, используя ТАКОЙ сегментный регистр… А так он может хоть сто лет там пролежать, но как только мы обратимся к памяти используя такой регистр (с индексом = 0) – ВСЕ! ХАНА! #GP!!!!

**Преобразование логического адреса в линейный**

И вот мы подобрались к самому важному моменту: как же все таки процессор формирует адрес и знает куда обращаться? Для этого нужно собрать все полученные нами знания в кучу. Что для этого нужно сделать? Уважаемый читатель, представь, что ты процессор... Допустим, сейчас ты выполняешь какой то код в оперативной памяти. И вот как ты размышляешь: о местоположении в памяти инструкции тебе ничего не известно, кроме того, что на нее указывает CS:EIP. По сути - это логический (т.е. некий абстрактный адрес). Как же найти линейный адрес в памяти, руководствуясь только этими двумя значениями: CS и EIP? Теперь мы можем это сделать! Сразу смотрим в CS и ищем в нем поле "Индекс" (см. на селектор выше). Смотрим в поле индекс и тут же узнаем о местоположении нужного дескриптора в таблице дескрипторов. Далее нам нужно узнать АДРЕС БАЗЫ сегмента. Узнали. Что теперь? Теперь осталось только одно: сложить этот адрес базы с EIP - и мы получим линейный адрес инструкции в памяти (к-рый при сегментной организации совпадает с физическим, мы об этом говорили ранее). Еще раз: селектор-->дескриптор-->база... +EIP = ЛИНЕЙНЫЙ адрес.

## Преобразование логического адреса в линейный

**Сегментный регистр**

В архитектуре процессоров Intel существуют ШЕСТЬ сегментных регистров:

CS, DS, SS, ES, GS и FS. Каждый из этих регистров отвечает за свой сегмент в памяти (кода, данных или стека). Итак, даже если программа состоит из ТЫСЯЧИ сегментов, ТОЛЬКО 6 из них могут быть доступны В ДАННЫЙ МОМЕНТ ВРЕМЕНИ. Другие сегменты станут доступны ТОЛЬКО ПОСЛЕ ЗАГРУЗКИ СООТВ. селекторов в сегментные регистры.

Помнишь, как в реальном режиме формировались линейные адреса? Значение сегментного регистра умножалось на 10h и прибавлялось смещение. Т.е. никакого селектора явно не существовало, никаких дескрипторов, ничего! Только сегментный регистр и смещение! Один шаг до формирования линейного адреса! В защищеном режиме нужно пройти сквозь огонь, воду и медные трубы (селектор-дескриптор- база...) чтобы докопаться до линейного адреса... Итак, в защищенном режиме сегментный регистр - это 16 битный регистр, содержащий информацию о дескрипторе (а именно - местоположении) и запрашиваемом уровне привилегий. Стоп! Здесь сразу же возникает вопрос: неужели же процессор при каждом обращении к памяти все время повторяет одни и те же действия (ищет дескриптор, потом ищет базу, затем прибавляет к базе смещение...), и так при выполнении фактически каждой команды... КОНЕЧНО ЖЕ НЕТ! НА САМОМ ДЕЛЕ сегментный регистр - это 80-битный регистр (!!! да да !!!), нам же доступны ТОЛЬКО младшие 16 бит, которые и называются СЕЛЕКТОРОМ!!! Остальные 64 бита называются "Теневым регистром" (Shadow register) или "Дескрипторным кэшем" (Descriptor cache), ОНИ И СОДЕРЖИТ ТУ САМУЮ БАЗУ, которую проц по идее должен был бы высчитывать на каждом шаге. Кроме базы этот самый "теневой регистр" содержит еще и лимит, и права доступа. Еще раз: как только мы загружаем в ВИДИМУЮ ЧАСТЬ (в селектор) соответствующее значение, процессор СРАЗУ же по селектору (а конкретно - по полю индекс) выпасает базу, лимит и права доступа из дескриптора и заносит их в "теневую часть" сегмнетного регистра, тем самым облегчая себе жизнь в дальнейшем.

А ТЕПЕРЬ ВНИМАНИЕ!!!

Если мы вдруг решим неожиданно поменять значение базы в дескрипторе для какого-либо сегмента, селектор которого в данный момент УЖЕ находится в сегментном регистре, то мы также должны позаботиться и о ПЕРЕЗАГРУЗКЕ сегментного регистра, т.к. в теневой части останутся СТАРЫЕ ЗНАЧЕНИЯ базы и лимита, и фактически процу абсолютно наплевать на то, что твориться в таблице дескрипторов, он руководствуется только ТЕКУЩИМ ЗНАЧЕНИЕМ БАЗЫ И ЛИМИТА В ТЕНЕВОЙ ЧАСТИ. Короче запомни золотое правило: поменял базу (или лимит) в дескрипторе - перегрузи соотв. сегментный регистр!!!

Загрузить сегментные регистры (явно или неявно) позволяют 16 команд:

ЯВНО:

```
MOV  - ну это и так ясно
POP  - значение из стека
LDS  - загрузить DS
LES  - загрузить ES
LSS  - загрузить SS
LGS  - загурзить GS
LFS  - загрузить FS
```

НЕЯВНО:

CALL, JMP, RET, SYSENTER, SYSEXIT, IRET, INTn, INTO, INT3. Чаще всего "неявные" команды изменяют значение именно CS-регистра, но в некторых случаях и других.

Кстати, как ты понимешь, в реальном режиме архитектура проца никуда не девается, при загрузке селектора в сегм. регистр процессор сам создает соответствующий дескриптор в его сркытой части. Он описывает 16-битный сегмент, начинающийся по указанному адресу с границей 64 Кб.

p.s. если инструкции LCS не существует в документации Интела, это не значит что ее не существует вообще. Я где то читал про это, просто интел в таблицах опкодов оставила для нее рядом с соотв. инструкциями lds, lss... пустой квадратик. Сейчас найти не могу. Кто об этом что то знает - мыльните пожалста.

На сегодня все. Пишите письма... brokensword@mail.ru.

# Процессор Intel в защищенном режиме #6

**Автор:** Broken Sword

**Дата:** 2002-10-15

**Prelude** Спешу всех порадовать – в самых ближайших выпусках мы начнем непосредственно программировать и использовать все полученные знания на практике. Знаний у нас пока немного, но достаточно для реализации простых действий. Поэтому настоятельно рекомендую ознакомиться со всеми выпусками рассылки (архив лежит здесь: http://subscribe.ru/archive/comp.soft.prog.intelpm). Подписчикам, которые внимательно читают все выпуски с самого начала, не помешает все же еще раз бегло пробежаться по оным (тогда на практике не возникнет лишних вопросов). **СТРАНИЧНАЯ АДРЕСАЦИЯ**

Ну вот, наконец-то мы добрели и до страничной адресации (не прошло и месяца :). Стоит отметить, что подписчики, у которых не возникло проблем с сегментной адресацией также легко и просто освоят страничную.

Прежде всего, нужно понять очень важную вещь: при использовании страничной адресации структуры из сегментной адресации (как то – таблицы дескрипторов, селекторы, регистры таблиц дескрипторов) НИКУДА НЕ ДЕВАЮТСЯ! Все остается на своих местах! Так в чем же тогда заключается страничная адресация? Где же она себя проявляет?

ВАЖНЫЙ МОМЕНТ: ЕДИНСТВЕННОЕ МЕСТО, ГДЕ СТРАНИЧНАЯ АДРЕСАЦИЯ ДЕЙСТВИТЕЛЬНО ВКЛИНИВАЕТСЯ В ПРОЦЕСС РАБОТЫ ПРОЦЕССОРА – ПРИ ПЕРЕВОДЕ ЛИНЕЙНОГО АДРЕСА В ФИЗИЧЕСКИЙ! Вот в этом вся соль! ЭТИМ НУЖНО ПРОНИКНУТЬСЯ!

Придется немного прокрутить пленку назад. Вспомни три понятия: логический, линейный и физический адрес. Логический адрес – это некий абстрактный, на деле ничего не значащий адрес, грубо говоря, CS:EIP – это и есть логический адрес, в самом деле, что он может нам сообщить? Ничего. Только то, что если мы вытащим из CS поле индекс, а затем по этому полю найдем в таблице дескрипторов соотв. дескриптор, а затем к базе из этого дескриптора прибавим EIP – то вот только тогда получим ЛИНЕЙНЫЙ адрес. При использовании сегментной адресации физический адрес совпадает с линейным (физический – это адрес который проц выставляет уже на адресную шину). Т.е. при сегментной адресации проц просто берет линейный адрес и без выкрутасов выставляет на адресную шину. При использовании страничной адресации именно на этапе перевода линейного адреса в физический в действие вступают новые силы, о коих и будет поведано ниже...

При использовании страничной адресации линейный адрес не совпадает с физическим, как в случае с сегментной адресацией. Т.е. мы имеем дело с виртуальной памятью. Процессор делит линейное адресное пространство на страницы фиксированного размера (длиной 4Кб, 2Мб или 4Мб), которые, в свою очередь, уже отображаются в физической памяти (или на диске). Когда программа (или задача) обращается к памяти через логический адрес, процессор переводит его в линейный и затем, используя механизму страничной адресации, переводит его в соответсвтующий физический адрес. Если страницы в данный момент нет в физической памяти, то возникает исключение #PF. Это по сути кульминационный момент: обработчик этого исключения (#PF) должен выполнить соответствующие манипуляции по устранению данной проблемы, т.е. подгрузить страницу с харда (или наоборот – скинуть ненужную страницу на диск).

Вообщем, все знают что такое своп-файл в винде? Вот по сути это и есть те самые странички памяти на харде, которые после возникновения #PF должны быть загружены в оперативку (или наоборот). По сути - #PF это не есть нечто ужасное и недопустимое (как #GP). НАОБОРОТ! #PF «нам строить и жить помогает!!!» Без него вообще ничего бы и не получилось по сути то дела!

Как только страница была благополучно водворена на место, то выполнение прерванной проги продолжается с той самой инструкции, которая вызвала #PF. А кстати, какая инструкция может вызвать #PF? Да в принципе любая, которая так или иначе обращается к памяти...

Страничная организация отличается от сегментной еще и тем, что все страницы имеют фиксированный размер (в сегментной размер сегментов абсолютно произволен). Также при сегментной адресации все сегменты обязательно должны присутствовать непосредственно в оперативе, а при страничной возможна ситуация, когда кусок сегмента находится в памяти, а другой кусок фактически того же сегмента – на харде (т.е. другими словами – часть страниц сегмента находится в оперативе, а часть – валяется в то же время на харде).

Проц всегда, где это возможно, облегчает себе жизнь. Вспомни про «теневую» часть сегментного регистра. Также и в случае со страничной адресацией – страницы, к которым проц чаще всего обращается кэшируются в процессоре, в области, которая называется буфер с ассоциативной выборкой (TLB - Table lookaside buffer). Т.е. не все такие страницы целиком, а только записи, которые необходимы для доступа к ним (см. ниже).

Несколько слов по поводу TLB. Начиная с P6 процев сущестувуют кэши специально для записей, описывающих код и данные, также разные кэши для 4Кб и для 4Мб страниц. Инструкция CPUID позволяет определить размер этих самых TLB в твоем проце. Чем они больше – тем быстрее соответственно будет работать проц. В отличии от теневой части сегментного регистра, куда мы не могли лезть руками и вообще не имели никакого доступа к ней, в TLB мы можем кое что изменять (естественно только в том случае, если мы на самом крутом уровне привилегий, на нулевом). Для этого существует команда INVLPG и др. приемы, на которых не стоит заострять внимания (если кому интересно в подробностях – пишите).

Итак, страничная организация непосредственно в проце управляется тремя флажками:

1. Флаг PG (paging): бит 31 в регистре CR0. Появился в 386 проце.
2. Флаг PSE (page size extensions): бит 4 в регистре CR4. Появлися в пнях.
3. PAE (physical address extension) flag: бит 5 в регистре CR4. Появился в пентиум pro процах.

Теперь подробнее. Флаг PG разрешает страничную адресацию. Сразу после установки его в единицу страничная адресация включена.

Флаг PSE, если его установить, позволяет использовать страницы больших размеров (4Мб и 2Мб). Если сброшен – страницы имеют размер 4Кб.

Флаг PAE позволяет расширить физический адрес до 36 бит (стандартно он 32-х битный). Данный флаг можно использовать ТОЛЬКО в режиме страничной адресации. Узнать, поддерживает ли твой проц данное расширение можно по 17 биту в EDX после CPUID).

Кстати, использовать 36-разрядную адресацию можно как с помощью PAE-флага, так и с помощью PSE-флага, это два разных метода, мы их рассмотрим дальше.

**Каталоги и таблицы страниц.**

Ну все, хватит лирики, пора приступать к делу. При трансляции линейного адреса в физический (при включенной страничной адресации) процессор использует 4 структуры данных:

1. Каталог страниц – массив 32-битных записей (PDE – page-directory entry), к-рые хранятся в 4Кб странице. Напоминает таблицу дескрипторов, не так ли? А сколько 32-битных записей помещается в 4Кб? Правильно, 1024 штуки. Т.е. всего 1024 PDE-шки...

2. Таблица страниц – массив 32-битных записей (PTE – page-table entry), которые также все расположены в одной 4Кб странице. Т.е. PTE-шек тоже может быть всего 1024 штуки. Забегая вперед – для 2Мб и 4Мб страниц таблица страниц вообще никак не используется – все решают только PDE-шки.

3. Сама страница – 4Кб, 2Мб или 4Мб кусок памяти :)

4. Указатель на каталог страниц – массив 64-битных записей, каждая из которых указывает на каталог страниц. Эта структура данных используется процем только при использовании 36-битной адресации.

Все эти таблицы позволяют обращаться к 4Кб или к 4Мб страницам при 32-х битной адресации и к 4Кб, 2Мб или 4Мб страницам при 36-битной адресации. Смотри на таблицу, какие флаги на что влияют:

| PG (CR0) | PAE (CR4) | PSE (CR4) | PS (PDE) | PSE-36 (CPUID) | Размер старницы | Размерность физического адреса (бит) |
|---|---|---|---|---|---|---|
| 0 | X | X | X | X | - | - |
| 1 | 0 | 0 | X | X | 4 Кб | 32 |
| 1 | 0 | 1 | 0 | X | 4 Кб | 32 |
| 1 | 0 | 1 | 1 | 0 | 4 Мб | 32 |
| 1 | 0 | 1 | 1 | 1 | 4 Мб | 36 |
| 1 | 1 | X | 0 | X | 4 Кб | 36 |
| 1 | 1 | X | 1 | X | 2 Мб | 36 |

Непонятно тут может быть только одно – что за колонка PSE-36? Дело в том, что этот режим работы (PSE-36) появился только в третьих пнях, а доступен он или нет можно узнать посредством CPUID. Как видишь, при использовании PSE-36 механизма доступны только страницы размером 4Мб. Фактически таким макаром можно адресовать 64 Gb физического адресного пространства.

Конечно, на первый взгляд получается несколько туманно и путано, столько флажков, какие то разрядности, но на самом деле все довольно четко и логично, просто надо вникнуть, а потом задать вопросы ).

Вообщем, расклад таков: при страничной организации существуют 3 способа адресации: 32-х разрядная, 36-разрядная с использованием флага PAE и 36-разрядная с использованием флага PSE.

Для начала рассмотрим самую простую и наглядную – 32-х разрядную адресацию. Все о чем пойдет дальше речь – справедливо только для нее.

**Линейная адресная трансляция (4Кб страницы)**

Обязательно дождитесь загрузки картинки! (Кто не хочет – пишите, вышлю по мылу). Вот она, механизма:

И что мы видим? А видим мы следующее: линейный адрес (тот самый, который получается в результате сложения базы сегмента со смещением при сегментной организации) при страничной организации с 4Кб страницами делится на три части:

• Номер записи в каталоге страниц: биты 22-31.
• Номер записи в таблице страниц: биты 12-21.
• Смещение в самой 4Кб странице: биты 0-11.

Для справки: в регистре CR3 (см. рисунок) только биты 12-31 содержат смещение каталога страниц в памяти. Где же младшие 12? А они ВСЕГДА равны нулю. Т.е. как не верти, а расположить каталог страниц ниже 4 Кб не удастся... Подумай над этим ).

На самом деле, может создаться ошибочное мнение, что каталог страниц – один единственный, и указывает на него регистр CR3 (как в случае с глобальной таблицей дескрипторов и регистром GDTR, помнишь?). На самом же деле все обстоит совсем не так, каталогов страниц может быть несколько, даже у каждой задачи свой, а регистр CR3 меняется после переключения задачи (подробнее – в выпуске, про переключение задач).

**Линейная адресная трансляция (4Мб страницы)**

Помнишь, чуть выше я упоминал, что для 4Мб страниц никакой таблицы страниц не существует? Смотри, как это выглядит в реале:

Видишь, фактически за счет размера страниц мы ничего не теряем и опять же, можем адресовать 4Gb, как и в случае с 4Кб страницами. 4Мб размер страницы получается в случае установки флагов PSE и PS. Флаг PSE ты уже знаешь где (в CR4), а что за флаг PS? А флаг PS находится непосредственно в записи каталога страниц. Увидишь его в следующем выпуске.

В случае 4Мб страниц линейный адрес делится на ДВЕ части:

• Номер записи в каталоге страниц (биты 22-31)
• Смещение в самой 4Мб-странице (биты 0-21)

**РЕГИСТР CR3**

Еще раз: младшие 12 бит ВСЕ ВРЕМЯ РАВНЫ НУЛЮ (их вообще то и нет даже на картинке по сути, флаги PCD и PWT не входят в эти младшие 12 бит, а существуют сами по себе. Все что не обозначено – зарезервировано.

PCD-флаг (page level cache disabled) – четвертый бит, контролирует кэширование каталога страниц. Если он установлен, то кэширование не происходит. Если сброшен – каталог страниц может кэшироваться. Этот флаг влияет только на L1 и L2 (внутренние кэши процессора). Процессор игнорирует этот флаг, если страничная адресация отключена (флаг PG в CR0 сброшен) или если

вообще отключен кэш (флаг CD в CR0).

PWT-флаг (page-level writes transparent) – третий бит, controls the write-through or write-back caching policy of the current page directory :). Переведи лучше сам... Также влияет только на L1 и L2 и зависит от PG и CD.

Регистр CR3 (главным образом, хранящий начало каталога страниц текущей задачи) также называют PDBR (page directory base register). Если мы собираемся использовать страничную адресацию, мы должны загрузить этот регистр в процессе инициализации. Загрузить его можно либо непосредственно командой mov, либо он сам загрузиться при переключении задач.

Может возникнуть ситуация, когда каталог страниц в момент переключения отсутствует в физической памяти, поэтому ОС перед переключением задачи должна устранить это недоразумение. Каталог страниц должен находится в памяти столько времени, сколько активна задача, использующая этот каталог страниц.

На сегодня все. В следующем выпуске познакомимся с структурами записей в каталоге и таблице страниц.

p.s. данный выпуск, как никакой другой, покажется тебе затуманенным и неясным. На самом деле это нормально. Главное – не заострять внимания на флажочках и битиках, а уловить общую суть – при страничной адресации линейный адрес распадается на n-частей (на 3 в случае 4Кб страниц и на 2 в случае 4Мб страниц), каждая из этих частей выполняет свою роль, и т.д.

p.p.s. есть ли среди подписчиков люди, серьезно занимающиеся демо-мейкерством (glide, 3D и т.п.)? Прошу откликнуться, есть одна бредовая идейка.

# Процессор Intel в защищенном режиме #7

**Автор:** Broken Sword

**Дата:** 2002-10-15

**Предисловие** Все. Это последний выпуск, посвященный нудной теории. Со следующего начинаем кодить. Поэтому всем еще раз настоятельно рекомендую перечитать предыдущие выпуски (архив лежит здесь: http://subscribe.ru/archive/comp.soft.prog.intelpm). **Элементы каталогов и таблиц страниц**

Напомню, что при страничной организации линейный адрес (база из дескриптора + offset) не соответствует физическому (т.е. адресу, который проц выставляет на адресную шину). Прежде, чем попасть на адресную шину он проходит ряд преобразований.

Итак, линейный адрес делится на три части в случае 4Кб страниц, и на две в случае 4Мб страниц.

Рассмотрим случай с 4Кб страницами.

Биты 22-31 линейного адреса – это НОМЕР элемента (записи) в КАТАЛОГЕ СТРАНИЦ. Адрес начала каталога страниц в памяти содержит регистр CR3 (см. предыдущий выпуск). Каждый элемент в каталоге страниц «весит» 32 бита. Рассмотрим его под микроскопом:

Структура элемента таблицы страниц (для 4Кб страниц)

Поле «Адрес базы таблицы страниц» содержит адрес базы таблицы страниц… :). Т.е. не адрес элемента из таблицы страниц, а только адрес НАЧАЛА ТАБЛИЦЫ ЭТИХ ЭЛЕМЕНТИКОВ… Сам элемент (его номер в таблице страниц) определяют биты 12-21 линейного адреса. Вот его структура:

## Структура элемента каталога страниц (для 4Кб страниц)

Очень похож на элемент каталога страниц, но все же есть отличия… И заключительный штрих – структура элемента каталога 4Мб старниц.

## Структура элемента каталога страниц (для 4Кб страниц)

P-флаг, бит 0: указывает, находится ли страница (или таблица страниц) в физической памяти. Если он равен 1 – то все в порядке, страница в памяти и можно продолжать формирование физического адреса, если же он сброшен – процессор вызывает исключение #PF (Page Fault), обработчик которого должен загрузить ее в память. Вообще то процессор никогда не меняет этот флаг сам – этим должна заниматься ОС.

R/W-флаг, бит 1: определяет привилегии чтения/записи для страницы или группы страниц (в случае, когда элемент каталога страниц указывает на таблицу страниц). Если флаг сброшен – страница доступна ТОЛЬКО ДЛЯ ЧТЕНИЯ. Если установлен – для чтения и записи. Данный флаг тесно связан с флагами U/S и WP из CR0. Об этом в выпуске о защите страниц.

U/S-флаг, бит 2: определяет привилегии пользователя/супервизора для страницы или группы страниц. Если флаг сброшен – страница доступна только для супервизорского уровня привилегий, если установлен – для пользователя и супервизора. Данный флаг связан с флагами U/S и WP.

PWT-флаг, бит 3: контролирует кэширование страницы (write- through и write-back). Если установлен – write-through кэширование разрешено, если сброшен – тогда разрешено write-back кэширование. Проц игнорирует этот флаг, если установлен CD (cache disabled) флаг в CR0.

PCD-флаг, бит 4: контролирует кэширование страницы. Если установлен – кэширование страницы отключено, если сброшен – разрешено. Нужен для страниц, которые не «делают погоды» и по сути могут только засорить кэш. Также, как и с PWT-флагом, значение PCD-флага игнорируется, если установлен CD-флаг в CR0.

A-флаг, бит 5: (аналог флага A в дескрипторе) показывает, было ли произведено обращение к странице с момента загрузки ее в память. По идее, ОС при загрузке страницы в память в первый раз должна сбросить этот бит в ноль. Затем, после ее загрузки в память, проц отлавливает момент, когда к ней в первый раз произойдет обращение и CAM устанавливает этот флаг в единицу. Все. Больше он его не трогает. Флаг A (также как и D) нужны для управления и контроля за страницами и таблицами страниц.

D-флаг, бит 6: показывает была ли произведена запись в страницу. Этот флажок игнорируется в элементах каталога страниц, которые указывают на таблицы страниц, т.е. при 4Кб-страничной организации. ОС должна обнулить данный флаг как только страница загружена в память, затем проц уже CAM установит его, как только в страницу будет произведена первая запись и больше его трогать не будет. Т.е. он будет торчать до тех пор, пока его не скинет ОС. Вообще данный флаг по сути схож с флагом A.

PS-флаг, бит 7 (для 4Кб страниц): определяет размер страницы. Если сброшен – страница 4Кб и элемент каталога страниц указывает на таблицу страниц. Если установлен – страница 4Мб (при 32-х битной адресации) или 2Мб (если используется EPA) и элемент каталога страниц указывает на САМУ СТРАНИЦУ. Если элемент каталога страниц указывает на таблицу страниц, то все страницы, на которые ссылаются элементы таблицы страниц также являются 4Кб-ными.

PAT-флаг, бит 7 в элементе таблицы страниц (для 4Кб страниц) или бит 12 в элементе каталога страниц (для 4Мб страниц): появился только в III-пнях (также есть в Xeon-ах и четвертых пнях). PAT – это таблица, расширение IA-32 архитектуры, вообще это все к MTRR-регистрам и к нам не относится. На процах, не поддерживающих данную фичу этот флаг зарезервирован и должен равняться нулю.

G-флаг, бит 8: появился в Pentium Pro процессорах. Указывает, что описываемая страница является ГЛОБАЛЬНОЙ, если установлен. Теперь смотри, для чего все это нужно. Если данный флаг установлен, а также установлен PGE-флаг в CR4, то соотв. элементы каталога и таблицы страниц никогда не будут выкинуты из TLB-кэша. Страницы, содержащие код ядра ОС и т.п. святые мощи желательно помечать глобальными, тогда их никто не посмеет выкинуть из TLB. Флаг G можно установить/сбросить ТОЛЬКО ПРОГРАММНО. Для элементов каталога, которые указывают на таблицу страниц, этот бит игнорируется, зато точно такой же битик есть и в соотв. элементе таблицы страниц, вот там проц на него уже смотрит.

Зарезервированные и доступные для программ биты: существуют во всех IA-32 процах. Биты 9,10 и 11 доступны для использования программно. Если бит P (бит присутствия сегмента) сброшен, то доступны все 32 бита :). Для элемента каталога страниц, который указывает на таблицу страниц, бит 6 должен всегда равняться нулю! Если PSE и PAE флаги в CR4 – установлены, то проц будет ругаться #PF, если зарезервированные биты не сброшены в нули.

Для вторых пней и более ранних: бит 7 в элементе таблицы страниц зарезервирован и должен равняться нулю. В элементах каталогов страниц для 4Мб страниц биты 12-21 должны быть равны нулю.

Для третьих и более поздних пней: В элементах каталогов страниц для 4Мб страниц биты 13-21 должны быть равны нулю.

Вот и все. Осталось рассмотреть всю эту братию при 36-битной адресации, займемся этим как-нибудь потом. А пока приготовься применить все полученные знания на практике. До следующего выпуска.

# Процессор Intel в защищенном режиме #8

**Автор:** Broken Sword

**Дата:** 2002-10-15

В этом выпуске мы наконец-то применим часть полученных знаний на практике. Сейчас мы попробуем переключиться в защищенный режим. И все. Возвращаться назад (в реальный) мы не будем... Ты сам убедишься, что для переключения в защищенный режим нужно выполнить ряд простых и незамысловатых по сути действий. Конечно, тебе наверняка хотелось бы побыть ТАМ подольше :), выполнить какие-нибудь невероятные, головокружительные трюки, чего-нибудь такое, чего в реальном сделать просто нереально. К сожалению, пока тех знаний, которыми мы обладаем, недостаточно, поэтому ограничимся малым - выведем на экран надпись.

Итак, для начала определимся с моделью памяти. Пусть это будет flat-модель (все сегменты имеют базу ноль и лимит 4 Гб). Рассмотрим сегментную адресацию. Страничную – в следующем выпуске.

Главное – это начать; начать можно с чего угодно, но я предлагаю с заполнения таблиц дескрипторов (надеюсь, еще помнишь что это такое? Дескриптор – описывает сегмент, селектор – указатель на дескриптор… Ну сейчас по ходу дела все вспомнится).

Поехали!

Таблица, которую нам нужно заполнить – таблица глобальных дескрипторов (GDT). У нас не будет таблицы локальных дескрипторов. Пусть в дескрипторах GDT будут описаны следующие сегменты:

• сегмента кода (код, который будет исполняться в защ. режиме)
• сегмент данных (данные, которые мы будем выводить на экран)
• сегмент видеопамяти (фактически это сегмент, в который мы будем выводить наш текст)

```
Глобальная таблица дескрипторов:
GDT:

; нулевой дескриптор (обязательно должен присутствовать в GDT!)
NULL_descr  db 8 dup (0)

; дескриптор 32-разрядного сегмента кода:   база = 00000000h, размер = FFFFFFFFh
CODE_descr  db 0FFh, 0FFh, 00h, 00h, 00h, 10011010b, 11001111b , 00h

; дескриптор 32-разрядного сегмента данных: база = 00000000h, размер = FFFFFFFFh
DATA_descr  db 0FFh, 0FFh, 00h, 00h, 00h, 10010010b, 11001111b , 00h

; дескриптор сегмента видеопамяти:          база = 000B8000h, размер = 0000FFFFh
VIDEO_descr  db 0FFh, 0FFh, 00h, 80h, 0Bh, 10010010b, 01000000b , 00h

; размер таблицы GDT:
GDT_size  db  $-GDT

; а следующие три слова (размер GDT и линейный адрес начала таблицы) мы должны
будем попозже загрузить в GDTR:
GDTR   dw  GDT_size-1
   dd  ?
```

Вычислим линейный адрес таблицы GDT. Зачем? Чтоб загрузить регистр GDTR. Линейный адрес GDT = линейный адрес базы сегмента RM_CODE (потому что GDT расположена именно в этом сегменте у нас в программе) + смещение метки GDT в нем.

```
xor  EAX,EAX  ; обнуление регистра EAX
mov  AX,RM_CODE  ; теперь AX = номер сегмента RM_CODE
shl  EAX,4   ; сдвигаем значение в EAX на 4 влево
; вот теперь EAX = линейный адрес базы сегмента RM_CODE

add  AX,offset GDT ; теперь EAX = линейный адрес GDT
; линейный адрес GDT кладем в заранее подготовленную переменную:
```

```
mov  dword ptr GDTR+2,EAX

; собственно, загрузка регистра GDTR:
lgdt  fword ptr GDTR
```

(вообще желательно просмотреть весь исходник целиком, потому что RM_CODE сбивает с толку)

Теперь нам еще нужно вычислить точно таким же макаром линейный адрес т.н. ТОЧКИ ВХОДА В ЗАЩИЩЕННЫЙ РЕЖИМ. Дело в том, что после переключения в защ. режим мы окажемся в неком подвешенном состоянии – когда регистр CS еще содержит номер сегмента из реального режима, а не селектор. Мы не можем просто сделать mov CS, селектор, CS можно изменить только дальним jmp-ом либо iret-ом, но щас не про то. Поэтому нам все же придется делать дальний jmp для изменения CS. НО КУДА? Вот именно на ТОЧКУ ВХОДА В ЗАЩИЩЕННЫЙ РЕЖИМ, см. исходник.

Мы уже почти готовы переключится в защ. режим, единственное, что мы обязаны еще сделать – ЗАПРЕТИТЬ ВСЕ ПРЕРЫВАНИЯ (причем, как маскируемые, так и немаскируемые), потому что пока мы не умеем с ними работать, и у нас нет ни одного обработчика, первый же тик таймера после переключения в PM завесит нам всю систему...

```
; запрещаем сначала маскируемые прерывания:
 cli

; а затем и немаскируемые:
 in  AL,70h ; читаем 70h порт
 or  AL,80h ; ставим восьмой бит
 out  70h,AL ; запихиваем на место
; теперь все прерывания запрещены
```

нада посидеть на дорожку :)... все. Теперь можно. Для переключения в PM нужно установить нулевой бит регистра CR0:

```
 mov  EAX,CR0 ; EAX = CR0
 or  AL,1  ; ставим нулевой бит
 mov  CR0,EAX ; пихаем назад...
```

Кто играл в Half Life, помните, в первой части, когда взрыв на заводе в самом начале, Гордон падает в обморок, а потом на несколько секунд приходит в себя и оказывается в другом мире, вокруг какие то существа едят траву, вот что то типа этого произошло сейчас... :)

Теперь давайте осмотримся кругом, что мы видим? Ага, вон таблица GDT во тьме. Память оперативная память на месте. Регистры, где вы? Мы все тута! Ну слава Богу... мы здесь не одни. Где таблица векторов? Но нет ответа... Ладно, и без нее обойдемся...

```
; загрузим в CS селектор на подготовленный сегмент кода. Мы не можем просто
взять и написать mov CS,селектор:
 dd  66h   ; префикс изменения разрядности операнда
 db  0EAh   ; опкод команды jmp far
 dd  ?    ; смещение в сегменте, на которое мы jmp-аем
 dw  00001000b  ; селектор сегмента, в который мы jmp-аем
```

Итак, давай разберемся, куда же мы все-таки прыгнули. Селектор равен 00001000b. Младшие два бита – нули, это уровень привилегий, на него не смотрим. Второй бит (TI) – нолик. Вспоминаем. Значит этот селектор указывает на дескриптор из таблицы GDT... На какой же? Смотрим левее... 001! На первый! А что у нас описывает первый дескриптор в GDT? Правильно, сегмент кода.

; значит, мы перелетели на некую «точку входа в защищенный режим», смещение которой мы уже высчитали (см. исходник):

```
ENTRY_POINT:
; загрузим сегментные регистры селекторами на соответствующие дескрипторы:
 mov  AX,00010000b ; AX = селектор дескриптора данных (№2)
 mov  DS,AX   ; кладем его в DS
 mov  AX,00011000b ; AX = селектор дескриптора видеопамяти (№3)
 mov  ES,AX   ; кладем его в ES
xor         SI,SI              ; обнуляем SI
```

```
 mov         SI,PM_DATA          ; SI = номер сегмента PM_DATA
 shl         ESI,4               ; ESI = линейный адрес сегмента PM_DATA
 add       ESI,offset message  ; ESI = линейный адрес строки message
      xor         EDI,EDI       ; EDI = позиция на экране (относительно 0B8000h)
 mov       ECX,mes_len         ; длина текста в ECX

; вывод на экран:
 rep         movsb    ; DS:ESI (наше сообщение) -> ES:EDI (видеопамять)
 jmp         $        ; погружаемся в вечный цикл
```

Теперь единственное, что мы можем сделать – это перезагрузить компьютер (причем, только кнопкой RESET). Я умышленно убрал часть кода, которая отвечает за переключение обратно в реальный режим, потому что она сильно все затуманит (надо в таблице GDT еще подготовить 16-битные дескрипторы), но если кому интересно – пишите.

Да, еще что. В полном исходнике (см. ниже) сразу может напугать «Линия A20». Что это такое? Дело в том, что после запуска компа для совместимости с 8086 используются 20-разрядные адреса (адресные линии A0-A19), т.ч. попытка записать что-то по линейному адресу 100000h приведет к записи по адресу 0h. Вот этот режим нам и нужно отменить (установив бит 2 в 92h порту) для использования 32-х разрядной адресации.

Самое главное – СПРАШИВАЙТЕ, СПРАШИВАЙТЕ И ЕЩЕ РАЗ СПРАШИВАЙТЕ! Приветствуются любые вопросы, начиная с «Что такое привилегированные инструкции?» и заканчивая «Почему так много пробелов в конце выводимой фразы?». На второй вопрос отвечу сразу. Дело в том, что при организации блоков повторений через irpc TASM почему то неправильно вычисляет его длину, если мы делаем это через equ. Посмотрите сами и напишите, если вы знаете в чем тут причина.

Скачать исходник целиком можно здесь: http://brokensword.hotbox.ru/PM.asm.

```
; ------------------------CUT HERE----------------------
; TASM:
; TASM /m PM.asm
; TLINK /x /3 PM.obj
; PM.exe

; MASM:
; ML /c PM.asm
; LINK PM.obj,,NUL,,,
; PM.exe

  .386p                                          ; разрешить привилегированные инструкции i386

; СЕГМЕНТ КОДА (для Real Mode)
; ---------------------------------------------------------------------------------------------
    -------
RM_CODE  segment  para public 'CODE' use16
  assume   CS:RM_CODE,SS:RM_STACK
@@start:
; очистка экрана:
                mov             AX,3
                int             10h

; открываем линию A20 (для 32-х битной адресации):
  in  AL,92h
  or  AL,2
  out  92h,AL

; вычисляем линейный адрес метки ENTRY_POINT (точка входа в защищенный режим):
  xor  EAX,EAX    ; обнуляем регистра EAX
  mov  AX,PM_CODE   ; AX = номер сегмента PM_CODE
  shl  EAX,4    ; EAX = линейный адрес PM_CODE
  add  EAX,offset ENTRY_POINT   ; EAX = линейный адрес ENTRY_POINT
  mov  dword ptr ENTRY_OFF,EAX  ; сохраняем его в переменной
; (кстати, подобный "трюк" называется SMC или Self Modyfing Code - самомодифицирующийся код)

; теперь надо вычислить линейный адрес GDT (для загрузки регистра GDTR):
  xor  EAX,EAX
```

```
  mov  AX,RM_CODE   ; AX = номер сегмента RM_CODE
  shl  EAX,4    ; EAX = линейный адрес RM_CODE
  add  AX,offset GDT   ; теперь EAX = линейный адрес GDT

; линейный адрес GDT кладем в заранее подготовленную переменную:
  mov  dword ptr GDTR+2,EAX
; а подобный трюк назвать SMC уже нельзя, потому как по сути мы модифицируем данные :)

; собственно, загрузка регистра GDTR:
  lgdt  fword ptr GDTR

; запрет маскируемых прерываний:
  cli

; запрет немаскируемых прерываний:
  in  AL,70h
  or  AL,80h
  out  70h,AL

; переключение в защищенный режим:
  mov  EAX,CR0
  or  AL,1
  mov  CR0,EAX

; загрузить новый селектор в регистр CS
  db  66h    ; префикс изменения разрядности операнда
  db  0EAh    ; опкод команды JMP FAR
ENTRY_OFF  dd  ?    ; 32-битное смещение
  dw  00001000b   ; селектор первого дескриптора (CODE_descr)

; ТАБЛИЦА ГЛОБАЛЬНЫХ ДЕСКРИПТОРОВ:
GDT:
; нулевой дескриптор (обязательно должен присутствовать в GDT!):
NULL_descr db  8 dup(0)
CODE_descr db  0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
DATA_descr db  0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
VIDEO_descr      db              0FFh,0FFh,00h,80h,0Bh,10010010b,01000000b,00h
GDT_size equ           $-GDT    ; размер GDT

GDTR  dw  GDT_size-1   ; 16-битный лимит GDT
  dd  ?    ; здесь будет 32-битный линейный адрес GDT
RM_CODE         ends
; ---------------------------------------------------------------------------------------------------
    -------


; СЕГМЕНТ СТЕКА (для Real Mode)
; ---------------------------------------------------------------------------------------------------
    -------
RM_STACK       segment          para stack 'STACK' use16
        db        100h dup(?)                    ; 256 байт под стек - это даже много
RM_STACK       ends
; ---------------------------------------------------------------------------------------------------
    -------


; СЕГМЕНТ КОДА (для Protected Mode)
; ---------------------------------------------------------------------------------------------------
    -------
PM_CODE  segment  para public 'CODE' use32
  assume  CS:PM_CODE,DS:PM_DATA
ENTRY_POINT:
; загрузим сегментные регистры селекторами на соответствующие дескрипторы:
                mov        AX,00010000b               ; селектор на второй дескриптор (DATA_descr)
         mov        DS,AX                     ; в DS его
         mov        AX,00011000b              ; селектор на третий дескриптор (VIDEO_descr)
         mov        ES,AX                     ; а этого в ES

         xor            SI,SI                     ; обнуляем SI
         mov            SI,PM_DATA                ; SI = номер сегмента PM_DATA
         shl            ESI,4                     ; ESI = линейный адрес сегмента PM_DATA
```

```
        add         ESI,offset message          ; ESI = линейный адрес строки message
              xor             EDI,EDI                     ; EDI = позиция на экране (относительно
    0B8000h)
        mov         ECX,mes_len                 ; длина текста в ECX

; вывод на экран:
        rep             movsb                       ; DS:ESI (наше сообщение) -> ES:EDI (
    видеопамять)
        jmp             $                           ; погружаемся в вечный цикл
PM_CODE         ends
; ---------------------------------------------------------------------------------------------------
    -------

; СЕГМЕНТ ДАННЫХ (для Protected Mode)
; ---------------------------------------------------------------------------------------------------
    -------
PM_DATA         segment         para public 'DATA' use32
  assume        CS:PM_DATA

; сообщение, которое мы будем выводить на экран (оформим его в виде блока повторений irpc):
message:
irpc            mes,
                db              '&mes&',0Dh
endm
mes_len         equ             $-message     ; длина в байтах
PM_DATA         ends
; ---------------------------------------------------------------------------------------------------
    -------

                end         @@start

; ------------------------CUT HERE----------------------
```

Находчивый подписчик сразу спросит: «А не могли бы мы, допустим, сделать базу видеосегмента в нуле, а при выводе на экран значение 0B8000h поместить в EDI?». Тот, у кого возникнет данный вопрос – уже видит защищенный режим насквозь. Конечно могли бы, это то же самое по сути!

Ну и напоследок еще одна деталька (на всякий случай): ВЫ МОЖЕТЕ ПЕРЕКЛЮЧИТСЯ В ЗАЩИЩЕННЫЙ РЕЖИМ НАХОДЯСЬ ТОЛЬКО В РЕАЛЬНОМ РЕЖИМЕ !!!

# Процессор Intel в защищенном режиме #9

**Автор:** Broken Sword

**Дата:** 2003-05-12

ПРОЦЕССОР INTEL В ЗАЩИЩЕННОМ РЕЖИМЕ Выпуск №9

После длительного затишья мы продолжаем изучать защищенный режим. В предыдущем выпуске был рассмотрен пример переключения процессора в защищенный режим БЕЗ использования страничной адресации. В данном выпуске будет рассмотрен код перевода проца в защищенный режим С использованием страничной адресации. По сути - код остался прежним, я лишь опишу изменения, которые необходимо внести.

Приступим. Следует иметь ввиду, что все изменения для включения страничной адресации производятся уже ПОСЛЕ ПЕРЕХОДА В ЗАЩИЩЕННЫЙ РЕЖИМ !!! И еще - страницы у нас будут 4-килобайтными!!!

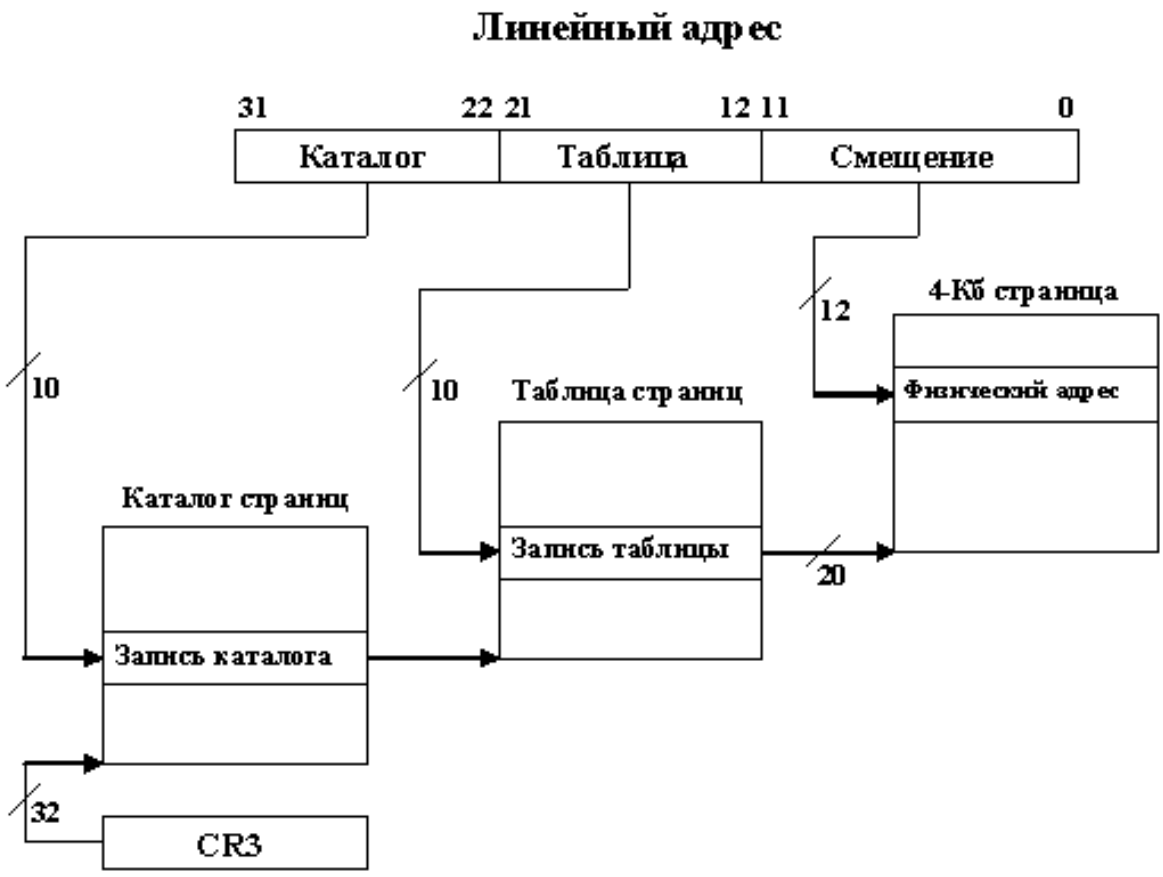
И вот они, эти изменения:

1. Подготовить каталог страниц
2. Заполнить таблицу страниц
3. Адрес каталога страниц поместить в регистр CR3
4. Включить страничную адресацию (установка бита 31 в регистре CR0)

THAT'S ALL!

Видишь, как все просто? А ты боялся!

Вот картинка (она уже у нас фигурировала), из которой видно для чего нужны эти 4 пункта:



Прежде, чем приступить, хочу вкратце рассказать, чего же мы будем творить. Вопрос в том, КАК НАГЛЯДНО ПОКАЗАТЬ РАЗНИЦУ МЕЖДУ СЕГМЕНТНОЙ И СЕГМЕНТНО-СТРАНИЧНОЙ адресацией? Оказывается, это не так-то уж и просто... Но вот как я попытаюсь этого добиться:

В предыдущем выпуске рассылки мы рассмотрели код, который выводит строку на экран, т.е. все свелось в конце концов к тому, что по адресу ES:0 вывелась надпись, где ES - селектор на предварительно подготовленный дескриптор, база которого равна 0B8000h (т.е. начало видеопамяти в текстовом режиме). Все однозначно и понятно. Теперь, как ты уже знаешь, при страничной адресации адрес 0B8000h препарируется на три части: старшие 10 бит - номер элемента в каталоге страниц, средние 10 бит - номер элемента в таблице страниц, и, наконец, младшие 12 бит - это СМЕЩЕНИЕ В СТРАНИЦЕ.

Недолго думая, можно умозаключить, что, попытавшись что либо записать по адресу 0B8000h при СТРАНИЧНОЙ адресации, мы не обнаружим там никакого начала видеопамяти, разумеется, если ПРЕДВАРИТЕЛЬНО (!) мы корректно не настроим каталог и таблицы страниц. Вот она, виртуальность! Вроде, вот же он: адрес начала видеопамяти! 0B8000h! У нас перед носом! Ан, нет... Это всего лишь иллюзия, т.к. этот адрес - не линейный, а виртуальный. Вот разбить его на три части, пройтись по каталогу и таблице страниц - вот тогда сформируется ОКОНЧАТЕЛЬНЫЙ АДРЕС (ВЫСТАВЛЯЕМЫЙ ПРОЦЕМ НА ШИНУ), который (при страницах в 4Кб) В ДАННОМ, КОНКРЕТНОМ СЛУЧАЕ (0B8000h) может принять значение от 0 до 1Мб!!! Надеюсь, понятно почему именно эти значения?

Поэтому вот оно, золотое правило: ПРИ СТРАНИЧНОЙ АДРЕСАЦИИ ЛИНЕЙНЫЙ АДРЕС ФОРМИРУЮТ ЭЛЕМЕНТЫ КАТАЛОГОВ И ТАБЛИЦ СТРАНИЦ, ну и конечно "проводником" по этим таблицам служит сам ВИРТУАЛЬНЫЙ адрес...

Значит, это все к чему? А к тому, что если представить все это дело НАОБОРОТ, то придем точно к такому же выводу! Т.е. я хочу сказать, что если 0B8000h - это на самом деле нечто совершенно иное, то с точно таким же успехом, ЛЮБОЙ ВИРТУАЛЬНЫЙ АДРЕС ПОСЛЕ ПРЕОБРАЗОВАНИЙ МОЖЕТ ПРЕВРАТИТЬСЯ В 0B8000!!! НАМ СТОИТ ТОЛЬКО ЗАХОТЕТЬ ЭТОГО! А что значит захотеть? А это значит, соответствующим образом заполнить элементы каталога и таблицы страниц, только и всего... На примере все станет совершенно ясно.

Вот что мы еще сделаем: вообще уберем дескриптор сегмента видеопамяти из таблицы GDT (итого у нас останется два дескриптора - кода и данных). Для чего - станет ясно ниже.

Итак, где-нибудь сразу после перехода в защ. режим и загрузки сегментных регистров соотв. селекторами нужно создать КАТАЛОГ СТРАНИЦ.

```
ENTRY_POINT:
; загрузим сегментные регистры селекторами на соответствующие дескрипторы:
          mov  AX,00010000b    ; селектор на второй дескриптор (DATA_descr)
     mov  DS,AX          ; в DS его
     mov  ES,AX            ; и в ES его же
```

вот теперь создадим каталог страниц. Что это такое? Каталог страниц - это набор 32- разрядных записей (элементов); вообще, структура записей подробно представлена в седьмом выпуске рассылки, поэтому кто забыл - посмотрите.

Начало каталога страниц в оперативной памяти будет располагаться по адресу 1Мб (100000h).

Теперь секундочку внимания! В нашем примере единственное, что мы делаем полезного - выводим на экран некоторую надпись. По сути, при этом используется один единственный адрес - ES:ESI, или 0B8000h. Следовательно, в данном примере для демонстрации возможностей страничной адресации достаточно заполнить лишь ОДИН ЕДИНСТВЕННЫЙ элемент каталога страниц. Так и сделаем:

```
      mov  EDI,100000h ; начало каталога страниц (1Мб)
      mov  EAX,101007h ; это наш один единственный значащий элемент
```

```
      stosd    ; ES:[EDI] <- EAX
      mov  ECX,1023  ; остальные 1023 элементов
      xor  EAX,EAX
      rep  stosd  ; забьем нулями все остальные элементы каталога
```

кстати, почему такой загадочный элемент - 101007h? Открываем формат элемента каталога страниц и все становится ясно. (не ленимся, не ленимся! Открываем 7 выпуск и смотрим на картинку). 7 (установлены младшие 3 бита адреса) - означает, что страница присутствует в оперативной памяти (бит P), доступна для чтения записи (бит R/W) и доступна с любого уровня привилегий (бит U/S). А что такое 1010? Не забыл, что младшие 12 бит для АДРЕСА таблицы страниц ВСЕГДА РАВНЫ нулю? Значит, это 1010000h, что соответствует 1Мб + 4Кб, а 4 Кб - потому что САМ каталог занимает столько.

все. С каталогом покончено. Теперь примемся за ТАБЛИЦУ СТРАНИЦ

```
      mov  EAX,00000007h ; первая запись - адрес нулевой страницы равен 0
      mov  ECX,1024  ; кол-во страниц в таблице

fill_page_table:
      stosd    ; запишем первый элемент
      add  EAX,1000h  ; добавим 4 Кб
      loop  fill_page_table ; и повторим для всех элементов таблицы страниц

      mov  EAX,00100000h ; базовый адрес = 1 Мб
      mov  CR3,EAX  ; в CR3 его! (база каталога страниц ВСЕГДА должна лежать в CR3)

; включить страничную адресацию
           mov  EAX,CR0
      or  EAX,80000000h
      mov  CR0,EAX
; а теперь изменить физический адрес страницы 12000h на 0B8000h
           mov  EAX,000B8007h
      mov  ES:00101000h+012h*4,EAX
```

вот здесь надо проникнуться всем существом. Чего это мы такого сотворили? А вот чего: мы, в таблице страниц, заменили адрес страницы, начало которой равно 12000h байт НА АДРЕС НАЧАЛА ВИДЕОПАМЯТИ (0B8000h)!!! Как это у нас получилось? Очень просто. АДРЕС НАЧАЛА ТАБЛИЦЫ СТРАНИЦ = 101000h, так? А сколько занимает один элемент? 4 байта!!! (32 бита). Поэтому 12*4 - это и есть ЗАПИСЬ ТАБЛИЦЫ СТРАНИЦ, которая указывает на страницу, начинающуюся по адресу 12000h!!!

А вот теперь демонстрация силы и мощи страничной адресации:

```
; вывод mes1 по стандартному адресу (начало видеопамяти 0B8000h)
      mov  EDI,0B8000h    ; для команды movsw, EDI = начало видепамяти
      mov      ESI,PM_DATA
      shl      ESI,4
      add      ESI,offset mes1  ; ESI = адрес начала mes1
      mov  ECX,mes_len      ; длина текста в ECX
      rep      movsw           ; DS:ESI (наше сообщение) -> ES:EDI (видеопамять)

; вывод mes2 по НЕСТАНДАРТНОМУ АДРЕСУ 12000h:
         mov      EDI,0120A0h ; 12000h (уже можешь считать, что это 0B8000h) + A0h
        mov       ESI,PM_DATA
        shl       ESI,4
        add       ESI,offset mes2     ; ESI = адрес начала mes2
        mov       ECX,mes_len         ; длина текста в ECX
        rep       movsw               ; DS:ESI (наше сообщение) -> ES:
```

А теперь небольшое партийное задание: переделать прогу так, чтобы начало видеопамяти совпадало с адресом 66000h.

Ответы присылать на brokensword@mail.ru

Кто разберется и пришлет правильный ответ - тот проникся страничной адресацией.

Напоследок - код всей проги (запускать ТОЛЬКО в реальном режиме :)

```
; TASM:
; TASM /m PM.asm
; TLINK /x /3 PM.obj
; PM.exe

; MASM:
; ML /c PM.asm
; LINK PM.obj,,NUL,,,
; PM.exe

  .386p                                          ; разрешить привилегированные инструкции i386

; СЕГМЕНТ КОДА (для Real Mode)
; --------------------------------------------------------------------------------
RM_CODE  segment  para public 'CODE' use16
  assume   CS:RM_CODE,SS:RM_STACK
@@start:
                mov             AX,03h
                int             10h            ; текстовый режим 80x25 + очистка экрана

; открываем линию A20 (для 32-х битной адресации):
  in  AL,92h
  or  AL,2
  out  92h,AL

; вычисляем линейный адрес метки ENTRY_POINT (точка входа в защищенный режим):
  xor  EAX,EAX    ; обнуляем регистра EAX
  mov  AX,PM_CODE   ; AX = номер сегмента PM_CODE
  shl  EAX,4    ; EAX = линейный адрес PM_CODE
  add  EAX,offset ENTRY_POINT   ; EAX = линейный адрес ENTRY_POINT
  mov  dword ptr ENTRY_OFF,EAX  ; сохраняем его в переменной
; (кстати, подобный "трюк" называется SMC или Self Modyfing Code - самомодифицирующийся код)

; теперь надо вычислить линейный адрес GDT (для загрузки регистра GDTR):
  xor  EAX,EAX
  mov  AX,RM_CODE   ; AX = номер сегмента RM_CODE
  shl  EAX,4    ; EAX = линейный адрес RM_CODE
  add  AX,offset GDT   ; теперь EAX = линейный адрес GDT

; линейный адрес GDT кладем в заранее подготовленную переменную:
  mov  dword ptr GDTR+2,EAX
; а подобный трюк назвать SMC уже нельзя, потому как по сути мы модифицируем данные :)

; собственно, загрузка регистра GDTR:
  lgdt  fword ptr GDTR

; запрет маскируемых прерываний:
  cli

; запрет немаскируемых прерываний:
  in  AL,70h
  or  AL,80h
  out  70h,AL

; переключение в защищенный режим:
  mov  EAX,CR0
  or  AL,1
  mov  CR0,EAX

; загрузить новый селектор в регистр CS
  db  66h    ; префикс изменения разрядности операнда
  db  0EAh    ; опкод команды JMP FAR
ENTRY_OFF  dd  ?    ; 32-битное смещение
  dw  00001000b   ; селектор первого дескриптора (CODE_descr)

; ТАБЛИЦА ГЛОБАЛЬНЫХ ДЕСКРИПТОРОВ:
GDT:
; нулевой дескриптор (обязательно должен присутствовать в GDT!):
NULL_descr db  8 dup(0)
CODE_descr db  0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
DATA_descr db  0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
```

```
GDT_size equ   $-GDT    ; размер GDT

GDTR  dw  GDT_size-1   ; 16-битный лимит GDT
  dd  ?    ; здесь будет 32-битный линейный адрес GDT
RM_CODE         ends
; ---------------------------------------------------------------------------



; СЕГМЕНТ СТЕКА (для Real Mode)
; ---------------------------------------------------------------------------
RM_STACK       segment          para stack 'STACK' use16
         db    100h dup(?)         ; 256 байт под стек - это даже много
RM_STACK       ends
; ---------------------------------------------------------------------------



; СЕГМЕНТ КОДА (для Protected Mode)
; ---------------------------------------------------------------------------
PM_CODE  segment  para public 'CODE' use32
  assume  CS:PM_CODE,DS:PM_DATA
ENTRY_POINT:
; загрузим сегментные регистры селекторами на соответствующие дескрипторы:
                 mov        AX,00010000b      ; селектор на второй дескриптор (DATA_descr)
  mov        DS,AX                        ; в DS его
  mov        ES,AX                        ; его же - в ES

; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; создать каталог страниц
                mov        EDI,00100000h            ; его физический адрес - 1 Мб
 mov        EAX,00101007h            ; адрес таблицы 0 = 1 Мб + 4 Кб
 stosd                       ; записать первый элемент каталога
 mov        ECX,1023                    ; остальные элементы каталога -
 xor        EAX,EAX                    ; нули
 rep                stosd
; заполнить таблицу страниц 0
                mov        EAX,00000007h            ; 0 - адрес страницы 0
 mov        ECX,1024                    ; число страниц в таблице
fill_page_table:
 stosd                       ; записать элемент таблицы
 add        EAX,00001000h            ; добавить к адресу 4096 байтов
 loop               fill_page_table     ; и повторить для всех элементов
; поместить адрес каталога страниц в CR3
                mov        EAX,00100000h            ; базовый адрес = 1 Мб
 mov        CR3,EAX
; включить страничную адресацию,
                mov        EAX,CR0
 or        EAX,80000000h
  mov        CR0,EAX
; а теперь изменить физический адрес страницы 12000h на 0B8000h
                mov        EAX,000B8007h
 mov        ES:00101000h+012h*4,EAX
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; вывод mes1 по стандартному адресу (начало видеопамяти 0B8000h)
 mov        EDI,0B8000h                ; для команды movsw, EDI = начало видепамяти
 mov            ESI,PM_DATA                ;
 shl            ESI,4
 add            ESI,offset mes1            ; ESI = адрес начала mes1
  mov        ECX,mes_len                ; длина текста в ECX
 rep            movsw                      ; DS:ESI (наше сообщение) -> ES:EDI
                                           ; (видеопамять)

; вывод mes2 по НЕСТАНДАРТНОМУ АДРЕСУ 12000h:
                mov            EDI,0120A0h     ; 12000h (уже можешь считать, что это
                                   ; 0B8000h) + A0h
                mov            ESI,PM_DATA
                shl            ESI,4
                add            ESI,offset mes2 ; ESI = адрес начала mes2
                mov            ECX,mes_len     ; длина текста в ECX
                rep            movsw           ; DS:ESI (наше сообщение) -> ES:12000h
```

```
                                ;(типа видеопамять)

 jmp            $                       ; погружаемся в вечный цикл
PM_CODE         ends
; ------------------------------------------------------------------------------------


; СЕГМЕНТ ДАННЫХ (для Protected Mode)
; ------------------------------------------------------------------------------------
PM_DATA         segment        para public 'DATA' use32
  assume        CS:PM_DATA

; сообщение, которое мы будем выводить на экран (оформим его в виде блока повторений irpc):
mes1:
irpc            mes1,          ; <This string was outputted to standart adress 0B8000h...>
                 db            '&mes1&',0Dh
endm
mes2:
irpc            mes2,          ;<And this one - to dummy adress 0120A0h. Cool?
                               ; Now press RESET...>
                db             '&mes2&',0Bh
endm
mes_len         equ            66             ; длина в байтах
PM_DATA         ends
; ------------------------------------------------------------------------------------------
                end         @@start
```

# Процессор Intel в защищенном режиме #10

**Автор:** Broken Sword

**Дата:** Не указана

*Об этом мало кто знает...*

**Можно пропустить**

Все, о чем пойдет речь в данном выпуске представляет большой интерес для разработчиков ОС и тех, кому просто хочется видеть все насквозь.

На создание данного выпуска побудила огромная неразбериха, творящаяся в литературе по описанию архитектуры различных ОС. Многие авторы, которые пишут свои книги со слов разработчиков ОС (Хелен Кастер) или сами являющиеся оными (Мэт Питрек) очень часто недоговаривают и перевирают некоторые наиважнейшие понятия (в некоторых случаях это делается специально, ввиду засекреченности того или иного аспекта), но путать три совершенно различных понятия адресного пространства - просто недопустимо. Авторы практически 90% книг копируют друг у друга различные определения, получается нечто вроде "испорченного телефона", а страдаем в конечном итоге мы.

Все дело во взаимосвязи и сути трех наиважнейших понятий в идеологии адресации: ВИРТУАЛЬНОЕ, ЛИНЕЙНОЕ и ФИЗИЧЕСКОЕ адресные пространства. Все эти понятия - отдельные сущности, взаимодействуют между собой, НО НИ В КОЕМ СЛУЧАЕ нельзя путать эти термины, поскольку на их совокупности зиждется любая ОС.

Например, очень часто в литературе по ОС можно встретить такие высказывания: "виртуальное (линейное) адресное пространство процесса..." или некоторые еще хитрее - не указывают, какое именно адресное пространство имеется ввиду - просто "адресное пространство процесса..."

**Виртуальное адресное пространство процессора Intel (IA32) в защищенном режиме**

Виртуальный адрес (он же логический) в процессорах с 32-х разрядной архитектурой (IA32) - совокупность ВСЕХ возможных адресов вида:

```
сегментный_регистр:смещение
```

Например, CS:12345678h - виртуальный (логический) адрес

В старой ОФИЦИАЛЬНОЙ документации (1995 г.) адреса такого вида назывались интеловцами виртуальными, в новой (2002 г.) они дают новое определение - логический, но по сути - это синонимы.

Почему ВИРТУАЛЬНЫЙ? Где здесь виртуальная сущность? Виртуальная сущность в том и состоит, что глядя на нечто вроде 6666h:12345678h НИКОГДА НЕЛЬЗЯ СКАЗАТЬ НАВЕРНЯКА, ОДНОЗНАЧНО, что на самом деле представляют из себя эти два числа. Знакомые с защищенным режимом сразу определят, что это некий адрес, и по левой части (селектору) можно найти число (базу), прибавляя к которой правую часть получится некий другой адрес, в общем, 6666h:12345678h - это некий виртуальный адрес, на самом деле (физически) его и нет вовсе...

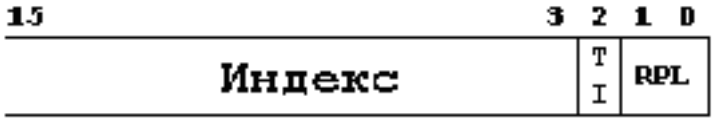
Теперь давайте разберемся, каков размер виртуального (логического) адресного пространства процессора. Многие тут же не задумываясь ответят - 4 Гб, потому что так написано во многих книгах, но, однако, внимательный читатель рассылки не будет делать столь скоропалительных выводов.

Итак, рассмотрим правую часть виртуального адреса. Здесь все и так понятно: 2^32 = 4Гб, т.е. правая часть виртуального адреса может принять 4Гб различных значений.

НО ЕСТЬ ЕЩЕ И ЛЕВАЯ ЧАСТЬ ВИРТУАЛЬНОГО АДРЕСА!!! (о которой почему то авторы многих книг забывают).

Здесь самое время привести уже подзабытые иллюстрации:

**Селектор:**




Селектор - это и есть та самая левая часть виртуального адреса.

Теперь, избавимся от битов, которые НИКАКИМ БОКОМ НЕ ОТВЕЧАЮТ за адресацию. Это два бита RPL. Многие с горяча захотят подмахнуть сюда еще и TI, но этого делать не нужно, т.к. TI - это индикатор таблицы дескрипторов, и он ЕЩЕ КАК отвечает за адрес, который в конце концов вылезет на адресную шину... (но это забегая вперед).

Теперь:

```
GDT может содержать 8192 дескриптора.
LDT может содержать 8192 дескриптора.

Нулевой дескриптор в GDT - недоступен.

Значит, имеем 8191 - в GDT и 8192 - в LDT.

Всего - 16383 дескрипторов. И каждый - 4Гб.
Всего: 16383 * 4Гб = 65532 Гб  (64 Тб)
```

Иначе говоря, виртуальный адрес - НЕ 32-х РАЗРЯДНЫЙ!!!

ОН - 46 РАЗРЯДНЫЙ!!! (32 бита смещения + 1 бит (TI) + 13 бит (индекс) )

2^46 байт = 65536Гб = 64 ТЕРАБАЙТ !!! - это и есть виртуальное адресное пространство процессора Интел в защищенном режиме.
(но реально, за вычетом нулевого дескриптора из GDT, имеем на 4Гб меньше адресов - 65532Гб)

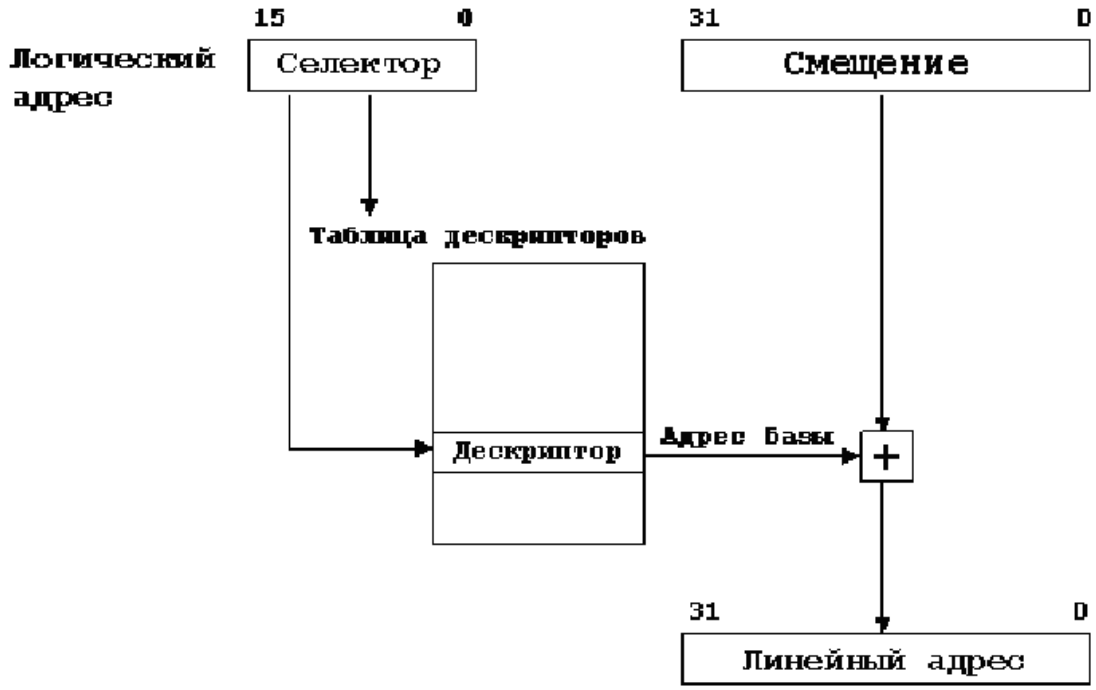
Всеобщие заблуждения: 64 терабайтное виртуальное адресное пространство, якобы, получается в Pentium Pro процессорах при использовании механизма PAE и PSE-36. На самом деле, PSE-36 ЭТО РАСШИРЕНИЕ ФИЗИЧЕСКОГО АДРЕСНОГО пространства, о нем далее, а на ВИРТУАЛЬНОЕ АДРЕСНОЕ ПРОСТРАНСТВО все эти расширения никак не влияют - оно ВСЕГДА = 64 терабайтам.

(цитата из ОФИЦИАЛЬНОГО мануала Интел-а 2002 г.: "the PAE paging mechanism and support for 36-bit physical addressing were introduced into the IA-32 architecture in the Pentium Pro processors...)

А теперь внимание вопрос на засыпку: чему, по вашему мнению, равен размер виртуального адресного пространства процессора Intel в РЕАЛЬНОМ РЕЖИМЕ? Ответы мыльте на ящик внизу.

Линейное адресное пространство процессора Intel (IA32) в защищенном режиме

Линейный адрес - это 32-х разрядный адрес, получаемый путем прибавления 32-х разрядного смещения (правая часть виртуального адреса) к базе сегмента.




Преобразование логического адреса в линейный

Линейный адрес - ВСЕГДА 32-х разряден. В этом легко убедиться, попытавшись опровергнуть это утверждение. Попробуйте создать сегмент с базой в 4Гб и сделать что то типа jmp селектор_этого_сегмента:2Гб (т.е. фактически должен возникнуть линейный адрес = 4 Гб (база) + 2 Гб (смещение) = 6 Гб). Вместо этого мы получим #GP. Вот и все.

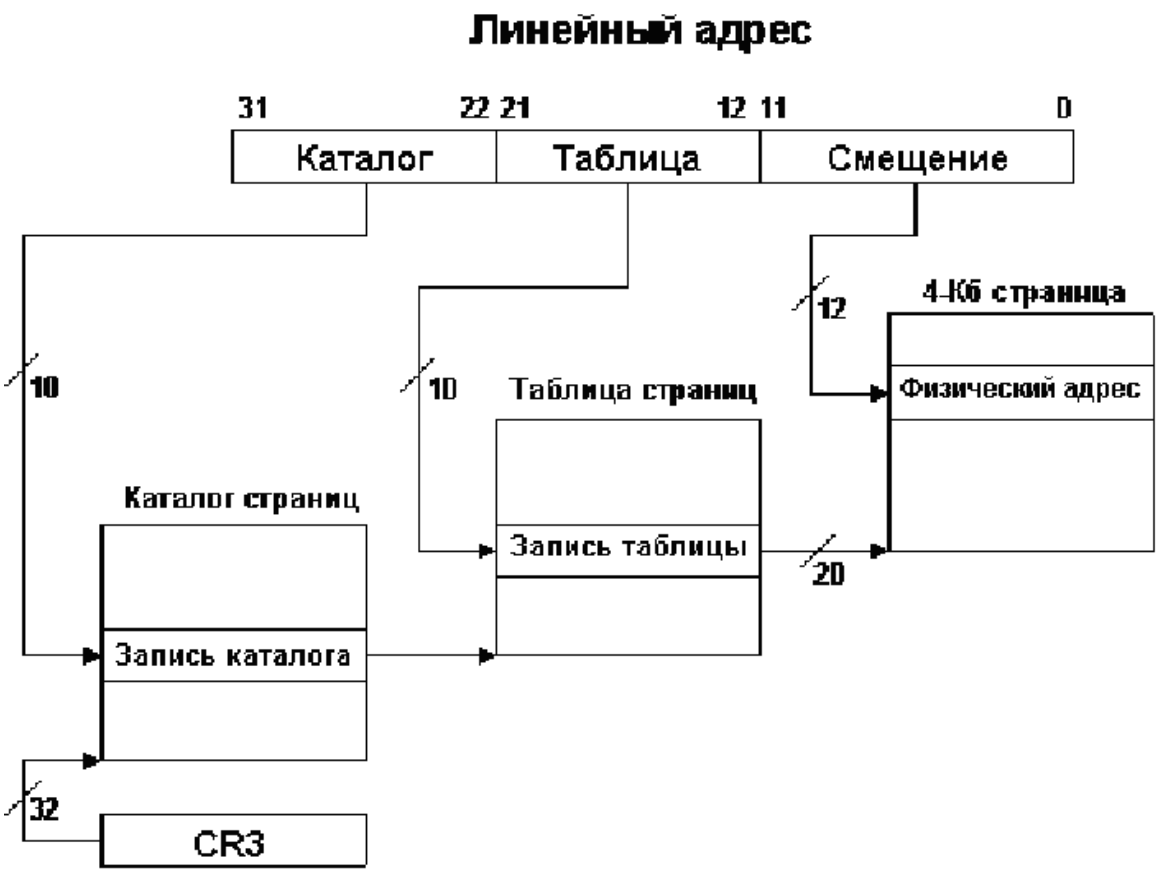
Поэтому, ЛИНЕЙНОЕ АДРЕСНОЕ ПРОСТРАНСТВО процессора Intel в защищенном режиме составляет 4 Гб.

**Физическое адресное пространство процессора Intel (IA32) в защищенном режиме**

Здесь возможны два варианта:

1. В режиме сегментной адресации физический адрес ВСЕГДА СОВПАДАЕТ с линейным.
2. В режиме сегментно-страничной адресации физический адрес НЕ СОВПАДАЕТ с линейным, а получается путем разбиения линейного адреса на 3 части и путешествия по каталогам и таблицам страниц.

По сути, при включенной сегментно-страничной адресации, линейный адрес также таит в себе некую "виртуальную сущность", потому что в принципе может быть представлен в виде X:Y:Z (действительно, он состоит из трех совершенно разных чисел: номера записи в каталоге страниц, номера записи в таблице страниц и смещения в странице), однако не стоит углубляться в такие дебри.




Физический адрес - это тот адрес, который процессор ВЫСТАВЛЯЕТ НА АДРЕСНУЮ шину процессора.

Вот здесь возникает еще одно заблуждение: многие считают, что физический адрес и адрес в оперативной памяти - это одно и то же. На самом деле, если разобраться, физический адрес - МОЖЕТ соответствовать адресу ячейки оперативной памяти (если она существует), либо НЕ МОЖЕТ (если ячейки по такому адресу не существует). В любом случае, следует помнить, что физический адрес - ЭТО НЕ ЕСТЬ АДРЕС ЯЧЕЙКИ ОПЕРАТИВНОЙ ПАМЯТИ, хотя и МОЖЕТ совпадать с ним.

Например, окончательно сформированный процессором из линейного адреса физический адрес составляет 678 Мб, он выставляется на адресную шину. А оперативной памяти на машине стоит, предположим, 256 Мб. Вот и ответьте сами себе: является ли ВЫСТАВЛЕННЫЙ физический адрес одновременно адресом ячейки в оперативной памяти? Конечно же нет! Такой ячейки попросту НЕ СУЩЕСТВУЕТ! Другой интересный вопрос - ЧТО ПРОИЗОЙДЕТ в таком случае дальше? Здесь уже все зависит от чипсета материнской платы - либо процессору будет послан сигнал #RESET (что и происходит в 90% случаях), либо он просто получит фиктивные данные. Но это не суть важно - важно, что множество АДРЕСОВ ЯЧЕЕК ОПЕРАТИВНОЙ ПАМЯТИ ВХОДИТ в множество ФИЗИЧЕСКИХ АДРЕСОВ.

За бортом остались такие интересные вопросы, как, например: "...А как ОС узнает о размере оперативной памяти, установленной на компьютере? Это число не хранится ни в одной ячейке области данных БИОС". Об этом, надеюсь, мы поговорим в следующих выпусках.

**К чему я все это веду...**

Очень часто, в 99% книг, можно встретить такое высказывание: "...Виртуальное (линейное) адресное пространство процесса составляет 4Гб..."

В одном предложении - сразу 2 ошибки, причем непростительные. Первая ошибка - ВИРТУАЛЬНОЕ и ЛИНЕЙНОЕ адресные пространства - взаимосвязаны, но это НИ В КОЕМ СЛУЧАЕ НЕ ОДНО И ТО ЖЕ!

Вторая ошибка: А ЧЕМУ ЖЕ СОБСТВЕННО, РАВНО ВИРТУАЛЬНОЕ АДРЕСНОЕ ПРОСТРАНСТВО ПРОЦЕССА? Действительно ли 4 Гб? Давайте разберемся...

Как известно, при создании процесса сегм. регистры загружаются такими значениями: CS = X, DS=SS=FS=GS=ES=Y, т.е. CS отличается от всех остальных (что не удивительно - попробуйте загрузить в сегм. регистр кода селектор сегмента данных или наоборот - #GP гарантировано). Поэтому имеем такую картину: ДВА РАЗНЫХ СЕЛЕКТОРА! Какой вывод мы должны сделать из этого? Может, он покажется шокирующим, но это факт - ВИРТУАЛЬНОЕ АДРЕСНОЕ ПРОСТРАНСТВО ПРОЦЕССА В ОС WINDOWS = 8 Гб !!! Т.е. процесс использует 8Гб виртуального адресного пространства из доступного 64 терабайтного! А если бы значения в DS, SS, FS, GS, ES также различались??? Ну что ж, тогда виртуальное адресное пространство процесса в ОС Windows составляло бы 24 Гб (6 сегм. регистров х 4 Гб)!

Итак, ВИРТУАЛЬНОЕ адресное пространство процесса в Win - 8 Гб! А все потому, что мы имеем ДВА РАЗНЫХ СЕЛЕКТОРА, а ведь именно они и участвуют в ФОРМИРОВАНИИ ВИРТУАЛЬНОГО АДРЕСНОГО ПРОСТРАНСТВА, оспорить это невозможно...

Замечание: на самом деле, в ОС Win, при создании процесса в сегм. регистр FS кладется ОТЛИЧНОЕ от DS, SS, GS, и ES значение, и этот регистр указывает на некоторую структуру (сегмент с базой 7FFDE000h и лимитом FFFh), связанную с процессом (структура TEB - Thread Information Block). Данная структура используется при структурной обработке исключений SEH и содержит кучу другой полезной инфы о процессе. Но все дело в том, что размер этой структуры - всего 4 Кб, поэтому его в расчет не берем, чтоб еще больше все не запутать. Но на самом деле, это надо учесть, поэтому вирт. адр. пр-во процесса в ОС Win составляет 8Гб + 4Кб.

А как же обстоит дело с линейным адресным пространством? Здесь все еще проще: линейное адресное пространство процесса ТЕОРЕТИЧЕСКИ могло бы составлять максимум от процессорного - 4 Гб, но т.к. верхние 2 Гб ОС Win NT защищены от доступа на уровне СТРАНИЦ, то получается, что ЛИНЕЙНОЕ АДРЕСНОЕ ПРОСТРАНСТВО ПРОЦЕССА В ОС Win NT = 2 Гб. В Win 9x - чуть больше, 3 Гб.

Вообще, судить о том, принадлежит ли данный адрес к тому или иному адресному пространству процесса можно по тому, возникнет ли исключение при его формировании или не возникнет. По этому признаку можно определить является ли данный адрес одним из адресов адресного пространства процесса или не является.

Например, при создании процесса исключения НЕ ВОЗНИКНЕТ при формировании процессором любого из следующих ВИРТУАЛЬНЫХ АДРЕСОВ:

```
DS:смещение
CS:смещение
FS:смещение
GS:смещение
SS:смещение
```

(а это, как мы знаем, 8Гб + 4Кб адресов, а теоретически могло быть и 24 Гб), значит, все эти адреса ВХОДЯТ в виртуальное адресное пространство процесса.

НО ИСКЛЮЧЕНИЕ ВОЗНИКНЕТ (!) при формировании ЛЮБОГО ЛИНЕЙНОГО АДРЕСА, БОЛЬШЕГО 2 ГБ (в NT) и большего 3 Гб (в 9x). Следовательно, та часть ЛИНЕЙНЫХ адресов, при формировании которых ВОЗНИКНЕТ #GP - НЕ ВХОДЯТ В ЛИНЕЙНОГО АДРЕСНОЕ ПРОСТРАНСТВО ПРОЦЕССА, а другие - ВХОДЯТ. В этом соль и это важно ПОНЯТЬ и ОСОЗНАТЬ.

Еще заблуждения:

Во многих книгах можно прочитать что то типа: "...считать данные из страницы", или "сбросить страницу в файл подкачки". Это некорректно. На самом деле, страница - это блок линейного адресного пространства. Страница НЕ МОЖЕ СОДЕРЖАТЬ ДАННЫХ. Данные содержатся в СТРАНИЧНОМ ФРЕЙМЕ (страничный фрейм - это блок физического адресного пространства). Т.е. нужно говорить "считать данные из страничного фрейма".

Таких тонкостей - море. Только до конца разобравшийся в них может считаться специалистом.

**Виртуальное адресное пространство процессора Intel (IA32) в реальном режиме.**

Да да! Оказывается, и в реальном режиме существует такое понятие.

Вообщем, все очень просто.

Виртуальный адрес в РЕАЛЬНОМ РЕЖИМЕ тоже состоит из двух частей:

сегмент:смещение

Максимальный виртуальный адрес в реальном режиме равен FFFFh:FFFFh, это очевидно. А сколько их ВСЕГО, этих ВИРТУАЛЬНЫХ АДРЕСОВ?

```
0000:0000
0000:0001
0000:0002
...
1234:5678
...
FFFF:FFFF

ровно FFFFFFFFh штук :)
а это есть 4Гб
```

Т.е. вывод: виртуальное адресное пространство процессора Интел в реальном режиме составляет 4 Гб.

Теперь, вы конечно же слышали про параграфы. Значит, можете совершенно справедливо заметить, что адрес 0000h:0010h и адрес 0001h:0000h преобразуются в ОДИН И ТОТ ЖЕ ЛИНЕЙНЫЙ АДРЕС 00000010h!!! и будете СОВЕРШЕННО ПРАВЫ! В один и тот же ЛИНЕЙНЫЙ АДРЕС! Но фактически, на него указывают ДВА РАЗНЫХ виртуальных адреса.

```
;*****************************************************************
ПРАВИЛО ПОЛУЧЕНИЯ ЛИНЕЙНОГО АДРЕСА ИЗ ВИРТУАЛЬНОГО В РЕАЛЬНОМ РЕЖИМЕ:

shl сегм.регистр, 4
add сегм.регистр, смещение

Если кто не знаком с асмом то я поясню: shl - это сдвиг "влево" на 4 бита, add - это прибавление.
На примере все станет ясно.

Пример 1:
1234h:5678h
Сдвинуть 1234h на 4 бита влево. Получаем 12340h.
Прибавляем 5678h. Получается: 12340h+5678h=179B8h

Т.е. вирт. адрес 1234h:5678h = лин. адресу 179B8h

Пример 2:
0001h:0000h

Сдвинуть 0001h на 4 бита влево. Получаем 0010h.
Прибавляем 0000h. Получается: 0010h+0000h=0010h

Т.е. вирт. адрес 0001h:0000h = лин. адресу 0010h
;*****************************************************************
```

Ну прям как в защ. режиме, только там вместо сдвига берется база и т.д.

Кстати, вот еще вопросец (пусть ответит, кто хочет): приведите пример линейного адреса, который можно получить максимально возможным количеством РАЗЛИЧНЫХ виртуальных адресов.

Ну вот вышеприведенный пример, линейный адрес 00000010h можно получить ТОЛЬКО (!) из двух разных виртуальных адресов. Но ведь, допустим, линейный адрес 00000020h можно получить уже ИЗ ТРЕХ РАЗЛИЧНЫХ ВИРТУАЛЬНЫХ АДРЕСОВ, а именно:

```
0000h:0020h
0001h:0010h
0002h:0000h
```

и все эти три виртуальных адреса - это линейный адрес 00000020h !!

В РЕАЛЬНОМ РЕЖИМЕ ЛИНЕЙНЫЙ АДРЕС ВСЕГДА СОВПАДАЕТ С ФИЗИЧЕСКИМ (ВЫСТАВЛЯЕМЫМ НА АДРЕСНУЮ ШИНУ) (кстати, как и в случае сегментной адресации в защ. режиме).

Вывод: линейный адрес НЕ совпадает с физическим ТОЛЬКО при страничной адресации в защ. режиме.

Было бы очень недурно, если бы и вы, уважаемые подписчики, включились в дискуссию, привнесли свое понятие и развили бы эту тему дальше, т.к. она исключительно важна и почему то еще более исключительно затуманена в литературе...

www.wasm.ru - сайт, посвященный программированию на ассемблере под Win, много инструментов и статей. Есть своя рассылка, мощный форум и прекрасно отлаженная команда разработчиков :).

rusfaq.ru - задай любой вопрос по асму (и не только...), и тебе ответят.

# Процессор INTEL в защищённом режиме #11

**Автор:** Broken Sword

**Дата:** 2003-05-19

**Вступление**

Здравствуйте уважаемые читатели (перейдем на "вы" :). Рассылка вернулась из зимней спячки. Как ни странно, процессоры Intel с архитектурой IA-32 до сих пор не ушли со сцены и продолжают лидировать в хит-парадах (и, по крайней мере, еще года 2 картина не изменится), поэтому рассылка не потеряла свою актуальность и можно продолжать изучение защищенного режима дальше.

Даже после того, как Itanium-ы полностью вытеснят со сцены IA-32 (хотя Itanium больше похож на проходную модель и скорее всего не займет процессорную нишу надолго) актуальность рассылки не пропадет: дело в том, что базовые понятия, которые вы накопите при изучении защ. режима IA-32, будут использованы при построении новых моделей процессоров еще очень долго. Уже сейчас, при разработке новых ОС используются знания и наработки 20-ти летней давности, то же самое ожидает и процессоры.

Да, вот еще что. Чтобы продолжить изучение дальше нужно хотя бы для приличия бегло просмотреть предыдущие выпуски, т.к. уверяю вас - те кто начал изучение вместе с первым выпуском рассылки и сейчас уже не помнит что такое "селектор" (а таких я уверен - 90%) - дальше ему будет нечего делать. Итак, для тех кто помнит что такое "селектор" - приступим :).

**Защита**

Данный выпуск я бы назвал самым идейным и важным из всего цикла, т.к. он повествует о самой сердцевине защищенного режима и любой современной ОС - о самом механизме защиты. Более того - в аннотации к рассылке можно увидеть слова из небезызвестной книги Зубкова С.В.: "...управление защищенным режимом в современных процессорах Intel - это самый сложный раздел программирования". От себя хочу добавить, что в этом разделе программирования глава "Механизм защиты" является самой сложной (хотя на самом деле все очень просто, сами убедитесь в этом).

Самый (опять это слово :), оно еще не раз встретится в данном выпуске, тема обязывает) главный вопрос, от которого и нужно плясать - так что же от чего все таки нужно защищать?

Для того чтобы понять суть происходящего обязательно нужно взглянуть на дело с high level. Представим картину: на столе стоит компьютер, из колонок льются шедевры самой лучшей mp3-коллекции в мире, на экране открыто несколько окон любимого браузера, где-то в углу досматривается фильм (с выключенным звуком естественно, чтоб не мешать музыке), пара документов свернута в панель задач, тут же на столе остывает чашка кофе... (кофе тут не при чем, просто для полноты картины). И тут возникает вопрос: как все это безобразие может сосуществовать в одном ящике? Как бедный несчастный процессор обрабатывает такой огромный поток данных и успевает все вовремя просчитать и не перепутать ни одного байта? Все это и многое другое обеспечивается благодаря механизму защиты.

Как можно убедиться на практике, связать аппаратные механизмы защиты с программными разработчикам из Microsoft (по крайней мере до Win2000) удавалось крайне плохо: в 99% "синих окон смерти" повинны дыры и недоработки именно в реализации защитных механизмов. Отчасти это вызвано кривыми руками разработчиков ОС, отчасти - кривой реализацией аппаратной защиты (да да! Почему то все любят спорить о глючности той или иной ОС, и никто никогда не скажет: "да этот процессор глючный, в нем много дыр и недоработок". Просто следует иметь ввиду, что аппаратная часть обычно не лишена недостатков программной. Это интересная тема, ее можно

развить и дальше, однако в конце концов можно прийти к выводу, что защита в IA-32 реализована настолько слабо и непродуманно (или что ее как-то не очень удобно "связать" с программной частью), что нет никакого смысла вообще о ней говорить. Это не так (во всяком случае, пока я набираю этот текст в Word-е, комп ни разу не завис).

Не нам об этом судить, но придет время, когда все современные методы и аппаратные механизмы вызовут лишь улыбку у гения инженеров будущего…

Итак, давайте не будем о грустном и, наконец, перейдем к делу (уже давно пора). В процессорах с архитектурой IA-32 защитные механизмы реализованы как на сегментном, так и на страничном уровнях (т.е. после включения страничной адресации нам предоставляются некоторые дополнительные возможности защиты, которые, вообще говоря, без страничной адресации не имеет смысла применять). Процессор различает четыре уровня сегментной защиты (3-0) и два - страничной (здесь номеров нет, но условно их два). Чем уровень ниже (численно), тем сегмент круче. Так, весь код ядра ОС, дрова устройств и другие распальцованные сущности помещают в сегменты с НУЛЕВЫМ (самый крутой) уровнем привилегий. Если перенести картину в нашу реальность, то в таких сегментах можно смело размещать код депутатов Госдумы, работников СБУ и всех, кто как говориться "когда идет по улице задевает всё пальцами". Простым смертным сюда вход строго воспрещен (только по пропускам, выданным вышеперечисленными ребятами, хотя "имея связи" в таблице дескрипторов прерываний (все как в жизни) сюда может запросто проникнуть любой директор колхоза "Красные маки" с 3 уровня привилегий (кстати, это не дыра, просто по другому было невозможно реализовать...), но об этом в следующем выпуске).

Механизм защиты обеспечивает проверку КАЖДОГО (!) обращения к памяти (естественно, до цикла чтения/записи). Любое нарушение границ и других законов, предписанных УК IA-32 (все статьи внимательно рассмотрим) - генерация исключения (что может привести к любым последствиям - вплоть до высшей меры наказания - расстрела приложения-нарушителя). Пожизненного заключения УК не предусматривает, т.к. процессор полностью свободен от морально-нравственных и т.п. сомнительных ценностей (последнее можно запросто вырезать цензурой).

Следует отметить, что все проверки производятся ПАРАЛЛЕЛЬНО с процедурой преобразования адресов, поэтому задержка на проверку отсутствует. Условно УК IA-32 можно разделить на 6 разделов:

• Проверка границ
• Проверка типов
• Проверка уровней привилегий
• Ограничения на адресацию
• Ограничения на использование инструкций
• Ограничения на точки входа в процедуры (?)

(название последнего раздела - мой кривой перевод из официального мануала, по ходу разберемся, что именно интеловцы здесь имели ввиду :).

Как уже говорилось ранее, нарушение любой статьи из любого раздела УК IA32 влечет генерацию прерывания и в зависимости от степени нарушения - соответствующие последствия для провинившегося. В ОС типа Win 9x любого нарушителя ждала одна и та же участь - расстрел на месте (поэтому Win 9x больше подходит под определение ОС 30-х годов прошлого века). В Win NT к власти пришли более разборчивые и творческие личности, здесь нарушитель в зависимости от степени нарушения может отделаться "легким испугом" J либо небольшим заключением в местах не столь отдаленных (подождет, когда ему будет "можно"). Этим NT более подходит под определение современной гуманной ОС.

Подводя итог вышеизложенного (по существу было сказано два предложения, однако не всякая мишура лишена смысла...) можно сказать, что процессор - очень напоминает то, что юристы понимают под термином "государство": суверенная, универсальная организация политической власти, призванная обеспечить нормальную жизнедеятельность людей, имеющая свою территорию, аппарат принуждения и взимающая налоги, необходимые для осуществления внутренних и внешних функций (запоминать не нужно). Процессор полностью подходит под это определение (разве что только налоги не взимает :). Как и в любом государстве в процессоре своя вертикаль власти, и самой важной составляющей все же является служба безопасности.

Очень важно понять: государство (читай - процессор) НЕ принимает решения самостоятельно, он просто констатирует уже свершившиеся факты, ПРОЦЕССОР НЕ МОЖЕТ убить программу (насильно завершить ее выполнение) - все решения принимают люди, населяющие данное государство (читай - программная реализация). Мозг власти (Госдума) - это не что иное, как обработчик исключения #GP и смежных с ним ведомств и подведомств (обработчики других исключений типа fault). Все зависит только от их реализации.

Вообще другой вывод (не менее важный) напрашивается сам собой - все сложные системы похожи друг на друга как близнецы.

На этом всё. Надеюсь, за неделю вы перечитаете предыдущие выпуски рассылки и мы сможем продолжить наше увлекательное интерактивное обучение

По возможности - обязательно посетите wasm.ru, найдете много нового и полезного.

http://subscribe.ru/archive/comp.soft.prog.intelpm/ - здесь можно скачать архив рассылки.

www.wasm.ru - сайт, посвященный программированию на асме под win (и не только), самая крупная коллекция статей в рунете, все самое новое и нужное (Aquila, положите плз исходник RLE, в нете на асме под Intel его нет нигде 100%).

rusfaq.ru - задай любой вопрос по асму (и жди ответа).

# Процессор INTEL в защищённом режиме #12

**Автор:** Broken Sword

**Дата:** 2003-05-21

**Уголовный кодекс процессоров Intel с архитектурой IA32 (с изменениями и дополнениями)**

В выпуске:

• Глава I: ОБЩИЕ ПОЛОЖЕНИЯ
• Глава II: НАРУШЕНИЕ ГРАНИЦ

а также: Зубков тоже ошибается...

**Глава I ОБЩИЕ ПОЛОЖЕНИЯ (важно прочитать и запомнить!)**

В это трудно поверить, но весь механизм защиты процессора Intel обеспечивается 3-мя флагами (+ еще 2 дополнительных при использовании страничной адресации) и 5-тью полями. Казалось бы, какую защиту могут обеспечить эти несчастные 8 (+2 до кучи) ребят? На самом же деле все довольно продумано и все чего надо они обеспечивают.

Флаги, обеспечивающие защитный механизм: (просто запомните их)

• Тип дескриптора (S) - бит 12 во втором двойном слове дескриптора сегмента. Собственно, отвечает за тип сегмента: системный, кода или данных.
• Флаг гранулярности (G) - бит 23 во втором двойном слове дескриптора сегмента. Вместе с полем "Лимит" дескриптора и флагом E (флаг направления роста) определяют размер сегмента.
• Флаг направления роста сегмента (E) - бит 10 во втором двойном слове дескриптора сегмента. Работает на пару с флагом G и полем "Лимит".
• Флаг пользователь/супервизор (U/S) - бит 2 элемента таблицы или каталога страниц. Определяет тип страницы: пользовательская или супервизорная. В прошлом выпуске упоминалось, что при использовании страничной адресации доступны некоторые дополнительные защитные механизмы. Этот флаг - один из них.
• Флаг чтения/записи (R/W) - бит 1 элемента таблицы или каталога страниц. Определяют тип доступа к странице: только для чтения или для чтения/записи. Флаг R/W - второй (он же последний) дополнительный механизм защиты при использовании страничной адресации.

Поля, обеспечивающие защитный механизм: (просто запомните их)

• Поле "Тип" - биты 8-11 во втором двойном слове дескриптора сегмента. Определяет тип сегмента кода, данных или системного сегмента. В принципе, работает на пару с флагом S (см. выше).
• Поле "Лимит" - биты 0-15 первого двойного слова и биты 16-19 второго двойного слова дескриптора сегмента. Вмести с флагом G и E определят размер сегмента. С этим полем все ясно.
• Поле "Уровень привилегий дескриптора" (DPL) - биты 13 и 14 во втором двойном слове дескриптора сегмента. Название поля немного сбивает с толку - на самом деле поле DPL задает уровень привилегий описываемого им сегмента, дескриптору уровень привилегий по сути ни к чему. Чем меньшее значение содержит это поле, тем описываемый сегмент круче (0 - самый крутой, 3 - сегмент-лох). Чем же крутой сегмент круче сегмента-лоха? Да всем. Сегмент с нулевым уровнем привилегий может исполнять ЛЮБЫЕ инструкции, обращаться к ЛЮБЫМ данным. По задумке, в сегментах с 0-ым уровнем должно располагаться ядро ОС, другие части ОС - в сегментах с 1-ым уровнем, драйвера - со 2-ым, пользовательские программы - с 3-им. На деле программисты из Microsoft при проектировании ОС "Винды" решили "упростить" модель до двухуровневой: все что не пользовательские программы - то в 0-ом уровне, пользовательские - в третьем. 1-й и 2-й уровни не используются. По сути, если бы Intel IA32 эксплуатировался только под осью Windows (что и происходит в 90% случаях), то для полей DPL, RPL и CPL с головой хватило бы одного бита вместо

двух :) и инженеры Intel-а немного перестарались.
• Поле "Запрашиваемый уровень привилегий" (RPL) - биты 0 и 1 ЛЮБОГО (!) сегментного селектора. Кто у кого чего запрашивает и зачем вообще это поле - это отдельная песня, она будет в следующей главе.
• Поле "Текущий уровень привилегий" (CPL) - биты 0 и 1 сегментного регистра CS (!). Фактически определяет уровень привилегий ТЕКУЩЕЙ исполняемой программы (процедуры), в общем - уровень привилегий исполняемого в данный кода.

THAT'S ALL! (это все!). Вся аппаратная защита такого монстра, как, скажем, Pentium 4, целиком и полностью стоит на описанных выше битиках! (я от нечего делать даже подсчитал - всего 35 бит! Т.е. все аппаратные защитные структуры умещаются (почти) в одно двойное слово! Честно говоря, это немного шокирует...

Если кому-то вздумается взглянуть на картинки - они есть в предыдущих выпусках.

**Глава II НАРУШЕНИЕ ГРАНИЦ**

Поле "Лимит" дескриптора сегмента призвано уберечь программы (процедуры) от незаконного обращения к памяти за пределами, установленными этим полем. Эффективное (реальное) значение лимита зависит от флага гранулярности G. Для сегментов данных, лимит также зависит от флага E (направление роста) и флага B дескриптора.

Влияние бита G на поле "Лимит" изучено и рассмотрено со всех сторон в предыдущих выпусках, но напомню: если бит G=0, то лимит сегмента совпадает со значением одноименного поля дескриптора; очевидно, что в данном случае сегмент может иметь длину от 0 до FFFFFh (1Мб). Если флаг G=1, то реальный лимит сегмента равен содержимому поля "Лимит" дескриптора, умноженному на 4Кб (FFFh). Очевидно, сегмент в этом случае может занимать от 4Кб до 4Гб.

ВАЖНО! Если бит G=1, то младшие 12 бит адреса НЕ ПРОВЕРЯЮТСЯ НА ПРЕВЫШЕНИЕ ЛИМИТА! Т.е. если создать дескриптор сегмента с G=1, и полем "Лимит" равным 0, то смещения от 0 до FFFh ВСЕ РАВНО ДОСТУПНЫ ДЛЯ ОБРАЩЕНИЯ! Т.е. процессор ИГНОРИРУЕТ младшие 12 бит при проверке на превышение лимита. Вообще, как где-то было верно подмечено, бит гранулярности - это чисто радиолюбительский трюк. Что еще небезынтересно знать - умножение на 4Кб равносильно сдвигу на 12 влево, причем младшие 12 бит при этом заполняются единицами.

Итак, процессор сгенерирует исключение #GP (самое страшное и главное, недаром ему присвоили 13 номер) в следующих случаях:

• Обращение к байту по смещению БОЛЬШЕМУ, чем эффективный лимит
• Обращение к слову по смещению БОЛЬШЕМУ, чем (эффективный лимит-1)
• Обращение к двойному слову по смещению БОЛЬШЕМУ, чем (эффективный лимит-3)
• Обращение к учетверенному слову по смещению БОЛЬШЕМУ, чем (эффективный лимит-7)

Это 4 заповеди данной главы.

Существует еще один размытый момент, который мы сейчас обсудим во всех нюансах. Этот момент - растущие "ВНИЗ" сегменты данных. Тут сразу нужно сказать - сегмент никуда не растет, это образное выражение. Все дело в том, что у растущих "ВНИЗ" сегментов поле "Лимит" в дескрипторе определяет НИЖНЮЮ границу сегмента, а ВЕРХНЯЯ граница ВСЕГДА (!) (подчеркнуть три раза) равна FFFFh (1Мб) если бит B=0 и FFFFFFFFh (4Гб) если бит B=1. Откуда взялся бит B? Вспоминаем структуру дескриптора, помните флаг D/B? Вот это он и есть. Теперь у некоторых возникает вопрос: если у нас определена НИЖНЯЯ граница сегмента (поле "Лимит") и верхняя (1Мб или 4Гб), то нахрена тогда поле БАЗА в дескрипторе? Спокойно! Стоит лишь чуть-чуть подумать и все станет ясно: поле "Лимит" хоть и определяет нижнюю границу, но все равно ОТСЧИТЫВАЕТСЯ ОТ БАЗЫ! Т.е. фактически нижняя граница у растущих ВНИЗ сегментов равна: содержимое поля "Лимит" дескриптора + содержимое поля "База" дескриптора. Вот для того нам база и нужна... (вообще с тем что поле "Лимит" всегда зависит от поля "База" интеловцы imho

погорячились, т.к. весь этот механизм в конечном счете выгоден только при использовании 36-разрядного механизма адресации PAE-36. У кого есть какие-нибудь соображения на этот счет - прошу поделиться).

Здесь интересен другой момент (подумайте и вышлите свои соображения на brokensword@mail.ru): если у нас есть растущий ВНИЗ сегмент, у которого сброшен бит B, т.е. верхняя граница этого сегмента равна FFFFh, а поле "База" в дескрипторе ПРЕВЫШАЕТ значение 1Мб. Кто может ответить - какие смещения доступны в таком сегменте?

Наконец, позволю себе привести фрагмент из книги тов. Зубкова С.В. по поводу растущих вниз сегментов: "Бит направления роста сегмента (E) обращает смысл лимита сегмента. В сегментах с этим битом, сброшенным в ноль, разрешены абсолютно все смещения от 0 до лимита, а если этот бит 1, то допустимы все смещения, кроме от 0 до лимита. Про такой сегмент говорят, что он растет сверху вниз, так как если лимит, например, равен -100, допустимы смещения от -100 до 0, а если лимит увеличить, станут допустимыми еще меньшие смещения". Здесь уважаемый Зубков запорол страшную ерунду, т.к., во-первых, не "от 0 до лимита", а от "(базы) до (база+лимит)", и в случае растущих вниз сегментов, помимо той же самой ошибки, он рассмотрел только случай когда B=1 (т.е. верхняя граница равна 4Гб), тогда действительно доступны ВСЕ смещения, кроме от база+лимит до 0 ("вниз"). Да и с примером он тоже намудрил. Если кто-то не согласен - пишите, обсудим.

Еще раз нелишне повторить, что суть проверок лимитов заключается в том, чтобы не дать программам разгуляться по всей памяти, каждый сверчок должен знать свой шесток и не вылазить за отведенные ему границы.

Данная глава была бы не полной, если не вспомнить о том, что процессор (помимо проверки лимитов сегментов) проверяет также лимиты таблиц дескрипторов, инфа о которых содержится в регистрах GDTR, IDTR, LDTR и TR. #GP будет сгенерировано, если произойдет попытка обращения к дескриптору, лежащему за пределами таблицы дескрипторов.

Вот теперь по теме "Проверка границ" больше добавить действительно нечего. В следующем выпуске мы рассмотрим следующую главу УК IA-32 под названием "Проверка типов".

http://subscribe.ru/archive/comp.soft.prog.intelpm/ - здесь можно скачать архив рассылки.

www.wasm.ru - сайт, посвященный программированию на асме под win (и не только), самая крупная коллекция статей в рунете, все самое новое и нужное.

rusfaq.ru - задай любой вопрос по асму (и жди ответа).

# Процессор INTEL в защищенном режиме #13

**Автор:** Broken Sword

**Дата:** 2003-06-26

**Zashita**

Вновь приветствую тебя, уважаемый читатель! И пусть никого не отпугнет номер выпуска (кстати, сегодня еще к тому же и пятница тринадцатое); сегодня мы продолжим интереснейшую тему – а именно тему защиты в защищенном режиме (простите за каламбур), которую предоставляет любой современный процессор из семейства Intel (вид x86). Несмотря на номер выпуска, ничего общего с мистикой данная тема не имеет, все до безобразия просто и прозаично. В качестве развлечения в сегодняшнем выпуске объявляется начало конкурса на отыскание ошибки... в самОм официальном мануале от Intel-a! (см. в конце)

Небольшое отступление. Помниться мне, что когда я только-только познакомился с ассемблером и где-то спустя год захотел освоить защищенный режим – ничего не получалось. Никак не шли все эти дескрипторы и уровни привилегий – нужно было постоянно держать в голове кучу информации + запоминать еще большую кучу новой, все перемешивалось и в итоге представлялось нагромождением бестолковых технических терминов и определений. На некоторое время я забросил все попытки, а потом и вовсе забыл про все это дело. Лишь спустя еще год я как-то от нечего делать заказал техническую документацию от Intel-a, полистал, почесал репу – вроде пошел процесс. Единственное объяснение которое я вижу – в течении года «застоя» были накоплены некоторые знания в других областях, причем даже близко не смежных с защищенным режимом (в частности – это могли быть какие то прочитанные книги, просмотренные фильмы – неважно, главное что все это не имело никакого отношения не то что к защ. режиму или ассемблеру, но даже к техническим наукам).

Все это к чему – знания, приобретенные в одной области самым непредсказуемым образом могут повлиять на процесс приобретения оных в других (возможно, абсолютно не относящихся к данной) областях. Что тоже важно - возможен и обратный эффект: чрезмерное увлечение, например, ассемблером может вызвать беспокойный сон :).

**ПРОВЕРКА ТИПА СЕГМЕНТА**

Это небольшой и простой раздельчик УК IA-32. Теперь самое время вернуться во 2-ой выпуск и посмотреть на картинку с дескриптором. Здесь нас интересуют два поля: бит S (бит 12 второго двойного слова) и поле «Тип» (биты 8-11 второго двойного слова). Если кто-то забыл про их назначение – прошу в выпуск №4.

Итак, процессор проверят эти два поля в следующих двух случаях:

1. При загрузке селектора в сегментный регистр
2. При обращении к сегменту

Чтобы не растекаться мыслью по древу сразу скажу: #GP возникнет если:

• в регистр CS загрузить селектор на дескриптор, который описывает отличный от сегмента кода сегмент;
• в регистры DS,ES,FS или GS загрузить селектор на дескриптор, который описывает «НЕЧИТАЕМЫЙ» сегмент кода;
• в регистр SS загрузить селектор на дескриптор, описывающий отличный от «ЧИТАЕМЫЙ сегмент ДАННЫХ» сегмент;
• в регистр LDTR загрузить селектор, не указывающий на LDT;
• в регистр TR загрузить селектор, не указывающий на TSS;

• при записи в «ИСПОЛНЯЕМЫЙ» сегмент (т.е. писать в сегмент кода нельзя через CS, через DS - пожалуйста);
• при записи в сегмент данных, который «ТОЛЬКО ДЛЯ ЧТЕНИЯ»;
• при чтении из «НЕЧИТАБЕЛЬНОГО ИСПОЛНИМОГО» сегмента;
• дескрипторы в таблице IDT должны быть шлюзами прерывания, ловушки или задачи;

список можно продолжить, но идея уже должна быть ясна. Бит S и поле «Тип» в дескрипторе говорят процессору, чего с сегментом делать можно, а чего нельзя. Если кто-то попытается сделать то, чего делать нельзя, то обработчик #GP (который мы сами и пишем) должен по идее нарушителя наказать.

**НУЛЕВОЙ СЕЛЕКТОР**

Это чуть ли не самый маленький раздел УК – содержит всего два положения:

Если в CS или SS загрузить селектор, который указывает на нулевой дескриптор (т.е. 16-разрядное число, биты 15-3 которого равны 0), то #GP возникнет немедленно;

Кстати, почему именно биты 15-3 – см. 5-й выпуск, а именно на картинку селектора.

Если нулевой селектор загрузить в DS,ES,FS или GS то #GP возникнет при попытке обращения к данным через такой селектор (иначе – он там может хоть сто лет пролежать). Вообще нулевой селектор придуман в качестве подстраховки от ненужных обращений к сегментам.

**УРОВНИ ПРИВИЛЕГИЙ**

Ну, вот мы и добрались до святая-святых механизма защиты – уровней привилегий. Иногда (в частности – в ОС Windows) их называют кольца защиты (Protection Rings). Как уже было неоднократно сказано – всего уровней привилегий 4: 0 – самый крутой, 3 – самый несчастный и забитый. В одном из предыдущих выпусков я уже разводил ахинею насчет Госдумы и других крутых ребятах, которых можно разместить в 0 кольце. В мире ОС в нулевом кольце располагается код ядра ОС, драйвера, обработчики исключений (тот же #GP). Хотя по задумке инженеров Intel дрова и другие сервисные программы должны располагаться в 1 или 2 кольце. Однако ОС Windows (кстати, и *nix тоже) используют только два кольца: 0 и 3. В третьем – пользовательские приложения и служебные сервисы ОС, в нулевом – все остальное (хотя это грубое разделение, но примерно так все и выглядит).

Здесь, во избежание дальнейших недоразумений, следует раз навсегда договориться: если я пишу «уровень привилегий задачи А БОЛЬШЕ уровня привилегий задачи В», то это значит, что численное значение привилегий задачи А МЕНЬШЕ чем у В (хотя это и так очевидно, 0<3, 0 круче 3).

С номерами разобрались. Поехали дальше. Всего существует три типа уровней привилегий (и их нужно знать как «Отче наш»): это CPL (Current Privilege Level), RPL (Requested Privilege Level) и DPL (Descriptor Privilege Level). Название каждого говорит само за себя (кроме, пожалуй, RPL – в смысл этого друга нужно еще попытаться въехать). Начнем по-порядку. Итак, CPL.

CPL – это текущий уровень привилегий, иначе говоря – уровень привилегий кода (задачи) исполняющейся в данный момент. Очень важно – именно в данный момент! Где же он храниться? Внимательный читатель сразу вспомнит формат селектора... Да. Именно в битах 1-0 он и храниться. Т.е. под CPL отведено 2 бита – всего могут принять 4 значения (0 – самый крутой, 3 – лоховской). Пока все сходиться. Теперь ВНИМАНИЕ! Где находиться сам селектор, хранящий CPL? В CS !!! В CS и нигде больше!!! (правда, он еще храниться и в SS, но пока, пожалуйста, представьте, что он только в CS). Т.е. если в данный момент исполняется код обработчика #GP (и он в раздумьях что сделать с нашалившей программой) то CPL кода (этого самого обработчика #GP) = 0 (хотя и необязательно – все зависит от фантазии разработчика ОС, но логично предположить, что CPL обработчика #GP равен 0). Пока про CPL информации хватит.

DPL – уровень привилегий СЕГМЕНТА. Храниться в дескрипторе каждого сегмента в таблице дескрипторов. Вообще вот посмотришь на предыдущий тип уровня привилегий – CPL. CPL – уровень привилегий кода, исполняющегося в данный момент. Но ведь код тоже находиться в каком-то сегменте! Так чем же они отличаются? Фразой «в данный момент». Вот их главное отличие. Ну и помимо того CPL относится ТОЛЬКО К СЕГМЕНТАМ КОДА (исполняющимся в данный момент :), а DPL содержится в КАЖДОМ ДЕСКРИПТОРЕ, неважно что описывает этот дескриптор (кстати, открываем второй выпуск и смотрим на картинку с дескриптором). Как ни странно, под него также отведено 2 бита. Процессор бросает свой гордый взор на поле DPL только при обращении к сегменту. Например, исполняется какая-то пользовательская программа. Все тихо и спокойно. Вдруг процессор встречает инструкцию:

```
mov dword ptr DS:[00000000h],0 ; программа хочет записать ноль по адресу DS:00000000h
```

(а DS равно все равно чему, например, 1234h)

Тут процессор естественно занервничает: а можно ли этой подозрительной программе с CPL=3 записывать данные туда, куда она намеревается это сделать? Тут же из селектора 1234h извлекается индекс (старшие 13 (!) бит - 246h) и в таблице дескрипторов (в данном случае – текущей LDT) находится нужный дескриптор, в котором процессор ищет в первую очередь поле DPL. И тут он видит, что в поле DPL содержится ноль! Т.е. уровень привилегий сегмента, в который хочет писать программа явно выше чем тот, который у подозрительной программы. Тут же генерируется #GP, обработчик которого приговаривает нарушителя к «...наказанию через расстреляние». У некоторых может возникнуть бестолковый вопрос: а если программа «успеет» записать нолик до тех пор, пока процессор соображает что к чему и возиться с привилегиями? Ответ: не успеет. #GP- это исключение типа ошибки (fault), оно ВСЕГДА генерируется ДО исполнения инструкции-нарушителя (об этом мы поговорим в выпуске про исключения).

И тут опять всплывает тот самый пресловутый «внимательный читатель рассылки», который в данной модели заприметит одно упущение: селектор храниться в КАЖДОМ сегментном регистре (будь то CS, SS, DS, FS…). С CS и SS вроде разобрались, а что же тогда хранят DS, FS, ES и GS ? Ведь даже в вышеприведенном примере использовался DS – так неужели же только для того чтобы посмотреть на DPL и найти нужный сегмент? Что хранят младшие два бита этого селектора??!!

RPL – уровень привилегий запроса. Мда. Здесь, чтобы понять для чего вообще нужен этот RPL, нужно немного поднапрячься. В принципе, возможно было построить процессор с архитектурой, использующей только два типа привилегий (CPL и DPL). Это доказывает хотя бы тот факт, что RPL в любом сегм. регистре любая пользовательская программа может изменить себе сама. Зачем Intel-овцы подмахнули RPL – скорее всего «для гибкости». В чем эта гибкость заключается – тема следующего выпуска.

На словах смысл CPL, DPL и RPL практически не улавливается (это нормально). Для понимания их сути идеально подходят зрительные образы. Следующий выпуск будет пестреть картинками, по которым даже школьник сможет понять смысл и суть этих наиважнейший понятий. Кроме того, в следующем выпуске мы найдем еще одну ошибку в книге Зубкова С.В. (а также опечатку в официальном мануале от Intel-а, из-за которой (возможно) и возникла ошибка у Зубкова). А посему пока объявляется обещанный конкурс на нахождение опечатки в официальном мануале Intel-а (подсказка: третий том (System Programming Guide), главы 4.8.1.1 и 4.8.1.2 (они небольшие, имена победителей будут объявлены в следующем выпуске).

И еще - я обзавелся сайтом (ну, это, конечно, громко сказано – сайт из серии «за 5 минут», пока что являет собой ссылки на все выпуски рассылки (наконец-то с картинками!) и некоторые мои поделки. Заходите, смотрите, может че полезного для себя и углядите: brokensword.narod.ru

# Прерывания в защищенном режиме процессора IA-32

**Автор:** Great

**Дата:** 2007-03-18

Вообщем-то на написание этой статьи меня толкнула незавершённость цикла статей http://www.wasm.ru/series.php?sid=20 "Процессор Intel в защищенном режиме". К сожалению, автор не успел рассмотреть детально сам процесс переключения режимов, смены адресации, а так же обработку прерываний, без которой невозможна полноценная работа программы в защищенном режиме. Писать мы будем код для транслятора FASM, сгенерируем чистый бинарный файл и используем его как образ загрузочной дискеты в эмуляторе Bochs или запишем на дискету и загрузим с нее реальный компьютер.

После Power-On-Self-Test процессор генерирует прерывание 19h, обработчик которого управляет дальнейшим ходом загрузки. Он находит первый (в порядке приоритетов, устанавливаемых в BIOS Setup) загрузочный диск, считывает его первый сектор по линейному адресу 07C00 и передает ему управление.

Поскольку процессор при загрузке работает в реальном режиме с 16битной адресацией, нашей задачей будет переключение процессора в защищенный режим с 32битной адресацией, установка обработчиков прерываний и считывание символов с клавиатуры. Так же мы рассмотрим процесс переключения из защищенного режима обратно в реальный.

Итак, сначала некоторые подготовительные действия. Поскольку наше тело загрузят по адресу 7C00 и мы вряд ли уместимся в пределы одного сектора (512 байт), надо обеспечить загрузку всего остального кода с диска. Этот код приведу без дополнительных пояснений, потому что он не совсем в тему нашей статьи:

```
ORG 0x7C00
use16

start:
    jmp init

....

init:
    ; очистка экрана
    mov  ax,3
    int  10h

    ; иницализация RM-сегментов и стека
    mov  ax, cs
    mov  ds, ax
    mov  es, ax
    mov  ss, ax
    mov  sp, start
    mov  bp, sp

    ; сие безобразие никак не умещается в пределы одного сектора (512 байт)...
    ; поэтому подгрузим остальные сектора в память
    mov  ah, 2  ; AH = Function                          : Read sectors
    mov  al, 10 ; AL = Number of sectors to read (1-128) : 10 ( с запасом ;-) )
    xor  ch, ch ; CH = Cylinder number (0-1023)          : 0
    mov  cl, 2  ; CL = Sector number (1-17)              : 2
    xor  dx, dx ; DH = Head number (0-15)                : 0
                ; DL = Drive number (0-A:, 1-B:)         : 0 (A:)
    mov  bx, start + 512
    int  13h
    jnc  continue_loading
```

```
    ; ошибка чтения. покажем сообщение и грохнем процессор
    jmp display_read_error
read_error db 'R',7, 'e',7, 'a',7, 'd',7, ' ',7, 'e',7, 'r',7, 'r',7, 'o',7, 'r',7
read_err_l dw $-read_error
display_read_error:
    mov  ax, 0B800h
    mov  es, ax
    xor  di, di
    mov  si, read_error
    mov  cx, word [read_err_l]
    rep  movsb
    jmp  $

ends: rb 510-(ends-start)
db 055h, 0aah

    ; чтение успешно, продолжаем инициализацию
continue_loading:
```

Теперь в памяти аккуратно располагается весь наш код. Далее нам необходимо включить отключенные (опять же для совместимости) адресные линии, потому что после включения компьютера функционируют только адресные линии A0-A19. Для использования полноценной 32битной адресации нам нужно включить адресную линию A20, установкой бита 1 на порту ввода-вывода 92h:

```
    ; открываем адресную линию A20
    in   al, 92h
    or   al, 2
    out  92h, al
```

На время переключения режимов обязательно надо отключить все прерывания, ибо первый же тик таймера свалит нашу систему. Мы отключим не только аппаратные прерывания, но и замаскируем NMI установкой 7-го бита (отсчет веду с нулевого, как всегда; по счету он, конечно, восьмой) в порту 70h:

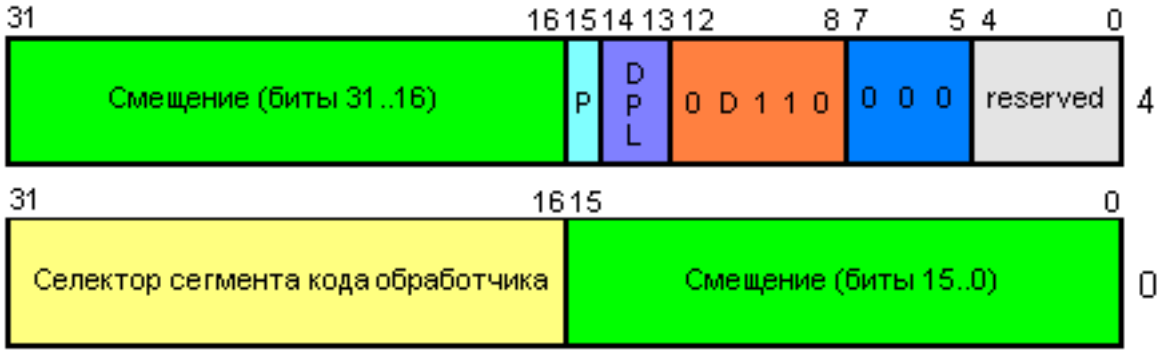
```
    ; запрет всех прерываний
    cli
    in   al, 70h
    or   al, 80h
    out  70h, al  ; запрет NMI
```

Далее нам необходимо построить GDT и IDT. Если с GDT все понятно, то про IDT стоит рассказать подробнее. И в реальном и в защищенном режиме в регистре IDTR процессор хранит адрес и лимит таблицы прерываний.

В реальном режиме база IDTR = 00000h, а лимит - 3FFh (размер 400h байт минус еденица). По адресу 00000 находится так называемая таблица векторов прерываний (Interrupt Vector Table), состоящая из 256 векторов. Каждый вектор содержит смещение и сегмент своего обработчика. Обе компоненты занимают 2 байта, таким образом общий размер таблицы составляет 25622 = 1024 = 400h байт.

В защищенном режиме дело обстоит совершенно по-другому. IDTR должен указывать на так называемую дескрипторную таблицу прерываний (Interrupt Descriptor Table, IDT), состоящую из 8байтных дескрипторов для каждого прерывания, которая может содержать шлюзы задачи, прерывания и ловушки. Мы рассмотрим только шлюз прерывания.

Шлюз прерывания описывается следующей структурой:




Как нетрудно заметить, его формат напоминает дескриптор из GDT/LDT, но есть некоторые изменения. Первое и последнее слова дескриптора шлюза прерывания содержат 32битный адрес обработчика прерывания (зеленое поле "смещение" на картинке). Второе слово содержит селектор сегмента кода, где находится код обработчика. Из дескрипторов сегмента унаследованы только следующие биты:

P (Present) - бит присутствия. Если =1, прерывание обрабатывается, если =0 генерируется исключение общей защиты. DPL (Descriptor Privilege Level) - уровень привилегий, о нем позже. D - разрядность.

При генерации прерывания происходит следующее. Из IDTR извлекается база таблицы дескрипторов прерываний. В этой таблице по номеру прерывания находится дескриптор шлюза прерывания. Если его бит Present сброшен, генерируется исключение общей защиты. Если текущий уровень привилегий отличается от уровня привилегий обработчика, происходит переключение стека и в стеке обработчика сохраняется указатель на стек прерванной задачи (SS и ESP). В стек помещаются регистры EFLAGS, CS, EIP. Для некоторых исключений последним в стек помещается еще и код ошибки, который, кстати, должен вытолкнуть обработчик исключения после обработки. Очищается бит TF, для программного прерывания или исключения сбрасываются биты VM, RF и NT. При вызове обработчика через шлюз прерывания очищается бит IF, блокируя дальнейшие маскируемые аппаратные прерывания.

После обработки прерывания обработчик должен вытолкнуть из стека код ошибки, если он там есть, и выполнить инструкцию IRETD, которая восстанавливает регистр флагов из стека (поле IOPL меняется, если CPL=0, IF меняется, если CPL<=IOPL). Если уровень привилегий прерванной задачи не равен уровню привилегий обработчика, выталкиваются регистры SS и ESP (обратное восстановление стека).

При вызове прерывания действуют механизмы защиты:

- не позволяется передача управления к менее привилегированному коду, если DPL сегмента кода обработчика больше, чем CPL - в отличие от обычной передачи управления, не проверяется поле RPL селектора - поле DPL шлюза прерывания проверяется только при генерации программного прерывания (INT, INT3, INTO). Для исключений и аппаратных прерываний оно игнорируется

Мы создадим таблицу прерываний только из 34 записей. Процессору, грубо говоря, наплевать, что их количество не равно 256. Просто при вызове неопределенных в ней прерываний будет исключение общей защиты. Мы определим обработчики для следующих прерываний:

• 1 - это у нас будет системный сервис вывода строки на экран =) будет выводить с текущего места ASCIIZ-строку, адрес которой лежит в ESI
• 13 - обработчик исключения общей защиты #GP. Покажем строчку "**GENERAL PROTECTION FAULT**" на экране
• 32 - IRQ0 - системный таймер. будем по тику таймера менять буковку на экране (смотрится прикольно ;))

• 33 - IRQ1 - клавиатура. мы будем отображать символы на экране один за одним, не различая регистр и не воспринимая функциональные клавиши, за одним исключением. Когда будет нажата клавиша <Esc>, мы попробуем переключиться в реальный режим, вызвать прерывание 10h BIOS с кодом AH=3 для очистки экрана и вернуться обратно в защищенный.

Вот такие нехитрые обработчики. Теперь самое время составить таблицу дескрипторов прерываний для нашего загрузчика:

```
; Interrupt Descriptor Table
IDT:
         dd 0,0 ; 0
         dw syscall_handler, 08h, 1000111000000000b, 0   ; 1
         dd 0,0 ; 2
         dd 0,0 ; 3
         dd 0,0 ; 4
         dd 0,0 ; 5
         dd 0,0 ; 6
         dd 0,0 ; 7
         dd 0,0 ; 8
         dd 0,0 ; 9
         dd 0,0 ; 10
         dd 0,0 ; 11
         dd 0,0 ; 12
         dw exGP_handler, 08h, 1000111000000000b, 0   ; 13  #GP
         dd 0,0 ; 14
         dd 0,0 ; 15
         dd 0,0 ; 16
         dd 0,0 ; 17
         dd 0,0 ; 18
         dd 0,0 ; 19
         dd 0,0 ; 20
         dd 0,0 ; 21
         dd 0,0 ; 22
         dd 0,0 ; 23
         dd 0,0 ; 24
         dd 0,0 ; 25
         dd 0,0 ; 26
         dd 0,0 ; 27
         dd 0,0 ; 28
         dd 0,0 ; 29
         dd 0,0 ; 30
         dd 0,0 ; 31
         dw int8_handler, 08h, 1000111000000000b, 0   ; IRQ 0 - системный таймер
         dw int9_handler, 08h, 1000111000000000b, 0   ; IRQ 1 - клавиатура

    IDT_size     equ $-IDT
    IDTR         dw IDT_size-1
                 dd IDT
    REAL_IDTR    dw 3FFh
                 dd 0
```

Обращаю внимание на второй IDTR - REAL_IDTR. Он как раз содержит base=0, limit=3ffh, чтобы загрузить его потом при переключении в реальный режим.

Вернемся к загрузке. Мы пока что только разрешили адресную линию A20. Самое время загрузить GDTR и IDTR:

```
    ; загрузка GDTR
    lgdt fword [GDTR]

    ; загрузка IDTR
    lidt fword [IDTR]
```

Далее нам необходимо включить защищенный режим установкой младшего бита служебного регистра CR0:

```
    ; переключение в PM
    mov  eax, cr0
    or   al, 1
```

```
    mov  cr0, eax
```

Что же теперь мы имеем? Процессор уже находится в защищенном режиме, однако код продолжает выполняться в 16битном сегменте (мы ведь помним про так называемые теневые части сегментых регистров. Теневая часть CS все еще хранит 16битный дескриптор сегмента, который нам достался "в наследство" от реального режима).

Самое время перезагрузить дескриптор CS командой дальнего джампа и перейти, наконец, в 32битный режим:

```
    ; загружаем новый селектор в CS
    jmp  00001000b:PROTECTED_ENTRY
```

Теперь смело ставим директиву use32 в fasm'e, указывая ему на то, что теперь код выполняется в 32битном режиме и стоит изменить режим генерации префиксов 66 и 67 у опкодов.

Но мы поменяли только регистр CS, в остальные сегментные регистры так и содержат до сих пор старые значения. Пора бы и их переинициализировать селекторами новых сегментов:

```
use32
PROTECTED_ENTRY:
    ; мы в PM, инициализируем селекторы 32-битных сегментов
    mov  ax, 00010000b  ; DATA
    mov  ds, ax
    mov  ss, ax
    mov  ax, 00011000b  ; VIDEO
    mov  es, ax
```

Раз мы загрузили нормальный регистр IDTR, мы можем со спокойной совестью, наконец, разрешить аппаратные прерывания и NMI, которые мы запретили на время перехода:

```
    ; разрешаем аппаратные прерывания и NMI
    in   al, 70h
    and  al, 7Fh
    out  70h, al
    sti
```

Теперь процессор переведен в полноценный 32битный защищенный режим. Воспользуемся нашим системным сервисом INT 1 и проинформируем об этом пользователя выводом строки "Switched to ProtectedMode. Press to clear display" на экране:

```
    ; выводим строку
    mov  esi, string
    int 1

...

string   db '  Switched to ProtectedMode. Press
 to clear display', 0
```

Далее ставим положение последующего вывода строк на 160 (третья строчка экрана) и переходим в бесконечный цикл ожидания прерываний:

```
    ; переходим на 3 строчку
    mov  dword [cursor], 160

    ; бесконечное ожидание прерываний ...
    jmp  $
```

Теперь рассмотрим наши обработчики. Начнем с простого - системный сервис INT 1:

```
;
; Системный вызов INT 1 - печать строки
;
; входные параметры: DS:ESI указывает на ASCIIZ-строку
;

syscall_handler:
```

```
    pushad
  _puts:
    lodsb
    mov  edi, dword [cursor]
    mov  [es:edi*2], al
    inc  dword [cursor]
    test al, al
    jnz  _puts
    popad
    iretd
```

Я думаю, ничего пояснять не надо ) Кроме того, что селектор ES у нас по-прежнему должен указывать в сегмент видеобуфера.

Далее обработчик #GP - покажем злостные ругательства и возвратим управление. Стоит заметить, что управление возвращается на ту же инструкцию, которая и вызвала исключение. Не забываем так же вытолкнуть из стека 4х байтный код ошибки.

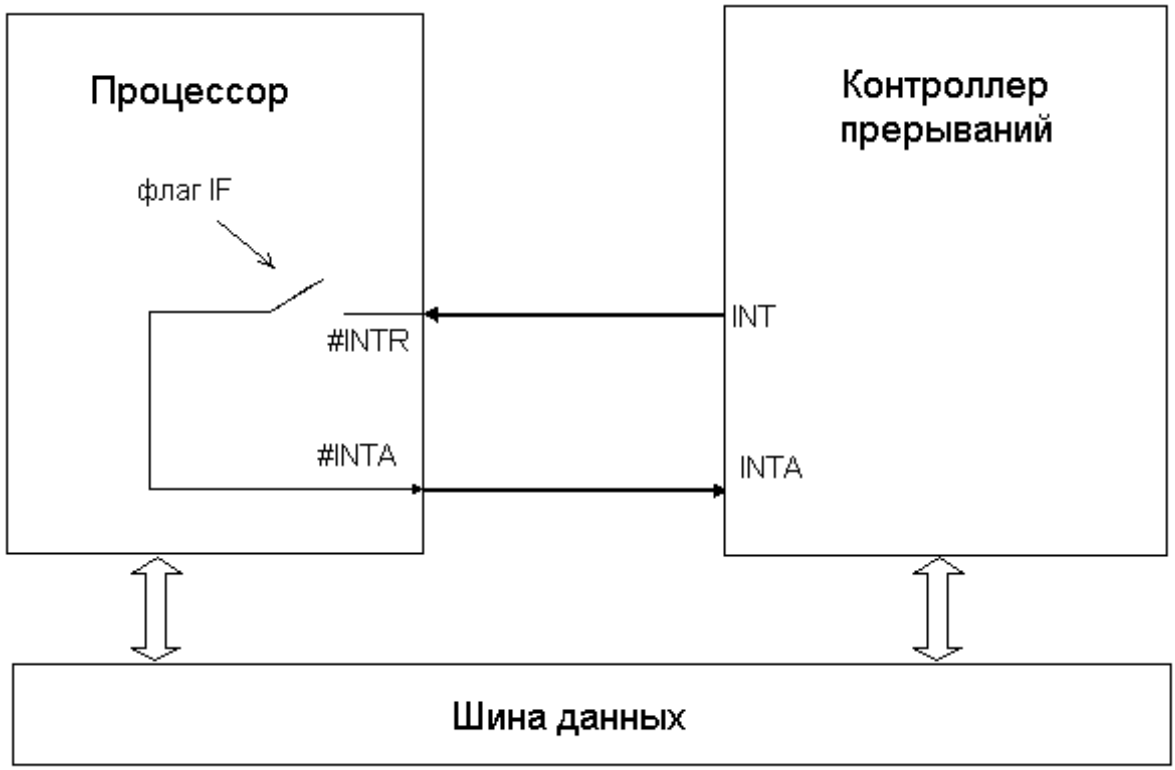
```
;
; Обработчик исключения #GP
;

exGP_handler:
    pop  eax ; код ошибки
    mov  esi, gp
    int  1
    iretd

gp db '**GENERAL PROTECTION FAULT**',0
```

Далее мы напишем обработчик IRQ0 от системного таймера. Мы будем инкрементировать байт ES:[0], который является самым первым байтом видеобуфера и будет отображаться в левом верхнем углу экрана:

```
;
; IRQ 0 обработчик - системный таймер
;

int8_handler:
    inc  byte [es:0] ; увеличим символ в левом верхнем углу экрана
    jmp  int_EOI     ; сбросим заявку на прерывание
```

Где int_EOI - наш "ничегонеделающий" обработчик, который просто сбрасывает заявку в контроллере прерываний.

Рассмотрим подробнее аппаратные прерывания. В компьютере есть программируемый контроллер прерываний, котогрый тесно взаимосвязан с процессором:




При возникновении аппаратного прерывания инициируется выход #INT контроллера. Он напрямую соединен с входом #INTR процессора. Если флаг IF=0, прерывание отбрасыватеся. Процессор опрашивает вход #INTR после выполнения каждой инструкции. Как только обнаруживается сигнал, процессор сразу же подтверждает прерывание через выход #INTA. Контроллер прерываний принимает сигнал INTA и выставляет на шину данных значние номера прерывания. Процессор считывает номер прерывания и входит в прерывание по описанной выше схеме. Контроллер прерываний 8259A имеет восемь входов IRQ0-IRQ7, открытых для внешних источников, выход INT и вход INTA, соединенные с #INTR и #INTA входом и выходом процессора соответственно. При получении внешнего прерывания на шине данных формируется номер прерывания из суммы IRQ-номера входа и некоторого базового значения, которое обычно равно восьми, а в защищенном режиме придется базу сдвинуть до 20h, как именно - смотри ниже. Таким образом, при получении сигнала на входе IRQ0 генерируется прерывания 8 (у нас будет 32), IRQ1 - 9 (33) и так далее.

При поступлении нескольких заявок от разных источников они обрабатываются по порядку начиная с меньшего номера IRQ. Можно также выборочно заблокировать некоторые заявки от отдельных IRQ-входов. Блокировку, или маскирование заявок, а так же выбор заявки с наибольним приоритетом обеспечивают три байтовых регистра контроллера - interrupt Mask register (iMr), interrupt Request register (iRr), interrupt Service register (iSr) и арбитр приоритетов Page Resolver (PR).

Каждый вход IRQ блокируется отдельным битом регистра масок iMr. Если прерывания на входе IRQn разрешены, бит n регистра iMr сброшен. Регистр маски iMr подключен к порту 21h процессора.

Пусть поступление запросов на вход IRQn разрешено - бит n регистра iMr сброшен. Бит n регистра запросов на прерывание iRr установится, когда придет сигнал на вход IRQn. Арбитр приоритетов PR по значению регистра iSr принимает решение о возможности обслуживания запроса. Заявка может быть принята к обслуживанию, если в регистре iSr не зафиксировано заявок с равным или большим номером. Если все нормально, контроллер выставляет сигнал INT. Заявка принимается к обслуживанию при получении сигнала INTA от процессора, на шину данных выставляется номер прерывания, бит n регистра iRr сбрасывается, бит n регистра iSr устанавливается. Установленный бит n регистра iSr теперь блокирует прерывания от входов с номерами большими либо равными n. Блокировку необходимо снять вручную в процедуре обсдуживания прерывания сбросом бита x в

регистре iSr, который подключен к порту 20h (нужно записать значение 3Nh, где N - номер входа). На практике обычно пользуются командой неопределенного сброса, реализуемой посылкой в порт 20h значения 20h. По этой команде в iSr сбрасывается заявка с наименьшим номером - которая обслуживается в данный момент как наиболее приоритетная.

8 входов слишком мало, поэтому чаще всего используются два контроллера прерываний 8259A - так называемые ведомый контроллер, выход INT которого подключен к входу IRQ2 ведущего контроллера, который уже в свою очередь общается с процессором. Это называется каскадным включением контроллеров прерываний. Регистр iMr ведомого контроллера доступен через порт 0a1h, а регистр iSr - через 0a0h.

Теперь стало ясно, как должен выглядеть минимальный обработчик внешних IRQ прерываний - посылка неопределенного сброса контроллеру прерываний обоим контроллерам:

```
;
; Пустой обработчик. сбрасывает заявку в контроллере
;

int_EOI:
    ; сброс заявки в контроллере прерываний: посылка End-Of-Interrupt (EOI) ...
    push ax
    mov  al, 20h
    out  020h, al   ; ... в ведущий (Master) контроллер ...
    out  0a0h, al   ; ... и в ведомый (Slave) контроллер.
    pop  ax
    iretd           ; возврат из прерывания
```

Теперь рассмотрим обработчик прерывания IRQ1 от клавиатуры. Прерывание IRQ1 генерируется контроллером клавиатуры каждый раз при нажатии клавиши. Обработчику клавиатуры скан-код считанной клавиши доступен для чтения через порт 060h. Скан-код нужно преобразовать в соответствующий ему ASCII-код символа (если он печатаем) и отобразить на экране. Преобразование произведем по следующей таблице:

```
; Таблица преобразования печатаемых скан-кодов в ASCII
ascii    db 0,'1234567890-+',0,0,'QWERTYUIOP[]',0,0,'ASDFGHJKL;',"'`",0,0,'ZXCVBNM,./',\
            0,'*',0,' ',0, 0,0,0,0,0,0,0,0,0,0, 0,0, '789-456+1230.', 0,0
```

Эта таблица содержит символы, индексы которых в таблице соответствуют их сканкодам, либо нули, если символы непечатаемые. Поскольку нажатие клавиш Shift и Caps Lock мы не обрабатываем, поэтому регистр букв у нас различаться не будет. В нашем обработчике нам нужно проверить, нажата ли клавиша (Её скан-код равен еденице). Если это действительно так, то вызовем процедуру переключения в реальный режим и генерации там прерывания 10h для очистки экрана. Если это не так, отобразим символ на экране. В любом случае нужно перед сбросом заявки на прерывание в контроллере прерываний послать подтверздение обработки прерывания контроллеру клавиатуры в порт 061h - необходимо установить и сразу сбросить 7 бит этого порта. После чего необходимо сбросить заявку на прерывание и вернуть управление:

```
;
; IRQ 1 обработчик - клавиатура
;

int9_handler:
    push ax
    push edi
    xor  ax, ax

    ; запрашиваем позиционный код клавиши
    in   al, 060h

    dec  al   ; Нажат ли
 ? (его сканкод = 1)
    jnz _continue_handling

    ; Esc нажат - пробуем переключиться в реальный режим, вызвать там
```

```
    ; прерывание 10h с кодом AH=3 (очистка экрана) и вернуться обратно

    mov  ax, 3
    push 10h
    call REAL_MODE_SWITCH_SERVICE

    jmp Ack

_continue_handling:
    ; отжатия не обрабатываем, только нажатия
    mov  ah, al
    and  ah, 80h
    jnz clear_request

    ; преобразуем позиционный код в ASCII по таблице
    and  al, 7Fh
    push edi
    mov  edi, ascii
    add  di, ax
    mov  al, [edi]
    pop  edi

    ; выводим символы на экран один за другим
    mov  edi, dword [cursor]
    shl  edi, 1
    mov  byte [es:edi], al
    inc  dword [cursor]

    ; посылка подтверждения обрабоки в порт клавиатуры
    ; (установка и сброс 7 бита порта 061h)
   Ack:
    in   al, 061h
    or   al, 80
    out  061h, al
    xor  al, 80
    out  061h, al

clear_request:
    pop  edi
    pop  ax
    jmp  int_EOI
```

Нам осталось определить функцию REAL_MODE_SWITCH_SERVICE, которой в стеке нужно передать номер прерывания и которая будет делать следующее:

• полноценный переход в 16битный реальный режим
• генерация прерывания с заданным номером с сохранением регистров
• переход обратно в 32битный защищенный режим и возврат управления.

Чтобы не портить контекст перед вызовом прерывания, нам нужно позаботиться о сохранении тех регистров, которые мы собираемся изменять в процессе перехода:

```
use32
old_ax dw ?
old_cl db ?
REAL_MODE_SWITCH_SERVICE:
    mov  [old_cl], cl
    mov  [old_ax], ax
```

Далее достаем из стека номер прерывания командой mov cl, [esp+4]. После этого отключаем аппаратные прерывания и NMI уже известным нам способом, предварительно запомнив командой pushfd старые флаги. После этого загружаем в IDTR регистр, описывающий таблицу векторов прерываний в реальном режиме:

```
    ; переключаемся обратно в реальный режим...
    lidt fword [REAL_IDTR]
```

Теперь нам нужно передать управление в 16битный сегмент с лимитом 64К. Это нужно обязательно сделать перед (!) переключением в реальный режим. Иначе мы получим не реальный режим, а так называемый Unreal Mode, который нас не интересует. Итак, перезагрузка регистра CS новым селектором:

```
    ; загружаем в CS селектор 16-битного сегмента с лимитом 64к
    jmp  00100000b:__CONT

use16
    __CONT:
```

Конечно, нужно поставить директиву use16, т.к. дальнейший код выполняется уже в 16битном сегменте. Теперь можно переключиться в реальный режим сбросом бита Protect Enable (PE) в управляющем регистре CR0:

```
    ; мы в 16битном сегменте. переключаемся в реальный режим.
    mov  eax, cr0
    and  al, 0FEh
    mov  cr0, eax
    jmp  0:REAL_ENTRY

;    Код реального режима
REAL_ENTRY:
```

Теперь мы находимся в реальном режиме и в теневой части регистра CS находится дескрикптор 16битного малого сегмента. Однако, регистры DS,SS,ES все еще хранят старые значения, для чего их сразу же нужно перезагрузить. Заодно разрешаем аппаратные прерывания и NMI.

```
REAL_ENTRY:
    mov  ax, cs
    mov  ds, ax
    mov  ss, ax
    mov  es, ax

    ; разрешаем аппаратные прерывания и NMI
    in   al, 70h
    and  al, 7Fh
    out  70h, al
    sti
```

Теперь динамически сформируем команду INT xx, потому что система команд архитектуры IA-32 поддерживает только команду INT с фиксированным номером прерывания. Нам придется сформировать ее динамически: опкод команды INT xx равен CD xx. Перед генерацией прерывания заодно восстановим регистры ax и cl, которые подвергались изменению в процессе переключения:

```
    mov  [int_no], cl

    mov  ax, [old_ax]
    mov  cl, [old_cl]

    db 0CDh       ; INT xx
    int_no db 0

    mov  [old_ax], ax
```

После возврата из прерывания запомним возвращенное значение ax, т.к. мы его поменяем в процессе обратного переключения. Далее все по-новой: запрет аппаратных прерываний и NMI, инициализация защищенного режима, прыжок в 32битный сегмент кода, перезагрузка селекторов DS,ES,SS.

Далее восстанавливаем AX, восстановление флагов командой popfd (нельзя тупо сделать sti, что казалось бы более очевидным, потому как перед во время вызова REAL_MODE_SWTICH_SERVICE прерывания отключены!) и делаем RET с выталкиванием четырех байт из стека (номер прерывания):

```
     mov  ax, [old_ax]
```

```
; разрешаем аппаратные прерывания и NMI
in   al, 70h
and  al, 7Fh
out  70h, al
popfd

ret  4
```

На этом мы, собственно, закончим наше и так затянувшееся повествование. Исходный код загрузчика и файл конфигурации Bochs (только в нем надо пути сменить на другие к ROM-BIOS) можно найти здесь: http://gr8.cih.ms/uploads/loader.rar

Пока!

(C) Great, 2007

В статье использованы материалы wasm.ru, dims.karelia.ru и книги А.Жуков, А.Авдюхин "Ассемблер", СПб.:"БХВ-Петербург", 2003.

ЗЫ. Как было справедливо замечено, вектора 0..1F заняты исключениями. Поэтому IRQ желательно перенсести, перепрограммировав контроллер следующей подпрограммой:

```
use32
redirect_IRQ:
; 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 )

        ; APIC Off
        mov     ecx,1bh
        rdmsr
        or      ah,1000b
        wrmsr

        mov     al,11h
        out     0a0h,al
        out     20h,al

        mov     al,bh
        out     0a1h,al
        mov     al,bl
        out     21h,al

        mov     al,02
        out     0a1h,al
        mov     al,04
        out     21h,al

        mov     al,01
        out     0a1h,al
        out     21h,al

        mov     al,dh
        out     0a1h,al
        mov     al,dl
        out     21h,al

        ; APIC On
        mov     ecx,1bh
        rdmsr
        and     ah,11110111b
        wrmsr

        ret
```

В код загрузки после инициализации протектед моде но перед включением прерываний придется добавить

```
mov  dx, 0FFFFh
mov  bx, 2820h
call redirect_IRQ
```

для переноса IRQ на 20..27 для ведущего и 28..2F для ведомого контроллеров.

Файлы (находится в архиве `wasm_files.zip`: `files/0452/loader.rar`) к статье.

# 20. Алгоритмы

# Сплайн. Аппроксимация данных

**Автор:** Chingachguk / HI-TECH

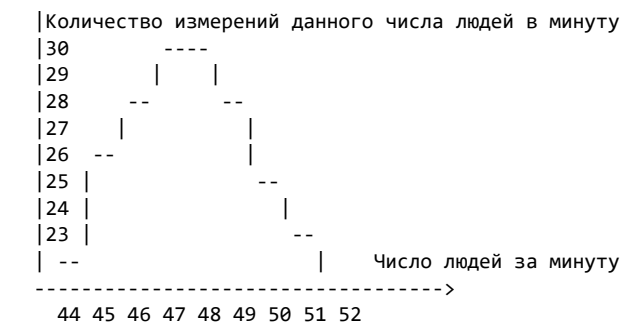
**Дата:** 2002-11-21

**Вводная часть.**

В данном документе речь пойдет об интерпретации экспериментальных данных.

Под экспериментальными данными понимается некий набор измерений, являющийся результатом опыта.

Так, например вы стоите на выходе из метро и считаете число людей, проходящих мимо вас за каждую минуту. Допустим, в результате у вас получились следующие измерения: 45,23,55,87,53,48,61... Обобщив измерения в виде гистограммы, вы получаете некоторое распределение:




Для экспериментатора может быть интересно понять, не может ли данное распределение быть описано некоторой аналитической формулой. Иначе говоря, определить природу процесса: с какой вероятностью можно ожидать выхода определенного числа людей в произвольном измерении, среднее число людей, выходящих из метро, наиболее вероятное и т.д.

У вас может возникнуть вопрос, зачем такие сложности, если все эти вопросы можно снять прямым измерением. Т.е. пусть перед нами стоит задача определить, сколько людей проходит за 8 часов через выход за рабочий день. Да, можно стоять все эти восемь часов и старательно записывать каждого отдельного человека. Именно так и поступает наше гос-во, когда хочет переписать всех мемберов нашей большой тусовки. При этом на исходе восьмого часа у вас уже слезятся глаза, у вас трижды проверяли документы и дважды подходили какие-то амбалы...

Однако анализируя результаты получасового опыта, в течении которого вы замеряли число человек за минуту, вы имеете другую возможность решить поставленную задачу. Допустим, вы предполагаете, что данное распределение (вероятность выхода определенного числа людей в минуту) описывается определенной формулой, проверяете свою гипотезу, убеждаетесь, что она более-менее достоверна. Формула дает Вам среднее значение распределения. Вам стало известно среднее число людей в минуту - достаточно умножить его на число минут в восьми часах для получения результата. Более того, вы также можете оценить ошибку ваших расчетов, поскольку подразумевается кратковременное (неполное) измерение, очевидно содержащее некоторую неопределенность результата в сравнении с абсолютно точным измерением, т.е. формула также может дать точность оценки, например:

```
Число человек в минуту = 47 ± 5;
```

где 5 - точность оценки за минуту, или примерно 10%. С такой же точностью можно будет определить число людей за восемь часов:

```
Число человек за 8 часов = (47 ± 5)×(8×60) = 22560  ± 2400;
```

**Пример аппроксимации данных опыта.**

Перейдем от абстрактного опыта к более жизненному. Предположим, вы анализируете свойства генератора 8-ми байтных ключей. Насколько случайными получаются ключи? Т.е. насколько близка псевдослучайная последовательность к истинно случайной.

Критерием качества псевдослучайной последовательности может служить число нулевых/единичных бит, пар бит 00-01-10-11, троек бит... Все они должны быть равновероятны, т.е. вероятность выпадения в каждом ключе определенного числа нулевых бит должно описывается так называемым биномиальным распределением:

$$f(k) = C_n^k / 2^n;$$

где n - полное число бит в ключе (в нашем случае 8×8 = 64), k - определенное число бит, Ckn - число вариантов размещения k нулевых бит по n свободным местам:

$$C_n^k = n! / k! \times (n-k)!;$$

2n - число всех битовых комбинаций в числе из n бит, $f$(k) же вероятность выпадения k бит в числе из n бит.

Таким образом, появляется возможность проверки генератора ключей на уровне числа битов, пар битов, троек... Для этого необходимо получить достаточное число ключей (провести опыт), получить данные опыта - число нулевых бит в каждом ключе и попытаться **аппроксимировать** , т.е. описать полученные данные некоторой зависимостью (формулой). Если распределение числа нулевых бит недостоверно описывается биномиальным распределением, что будет наблюдается, например, в случае следующего алгоритма генерации ключа:

```
LOOP INDEX=1 TO 8
  KEY[INDEX] = PASSWORD[INDEX] XOR MAGIC_NUMBER
END LOOP
```

то можно будет сделать определенные выводы о свойствах генератора.

В некоторых случаях аппроксимацию данных называют "сплайном". Данное наименование можно встретить в таких популярных программах, как Excel, в котором можно попытаться выполнить сглаживание (splain) зависимости, например полиномом степени от 2 до 6, правда весьма примитивно. Также можно упомянуть визуальное отображение звука в некоторых CD/mp3 плеерах в виде огибающей кривой. В качестве более-менее профессиональных средств анализа численных данных рекомендую Origin.

**Еще один пример аппроксимации (реальный опыт).**

Приведу также пример одного реального опыта, выполненного учеными во льдах Антарктики.

Как известно, полярная шапка нашей планеты представляет собой огромную толщу льда (до нескольких километров), лежащую на твердой (каменистой) поверхности Земли. Она образовывалась тысячи лет: вода замерзала и увеличивала эту толщу.

Ученые провели следующий опыт: они бурили небольшую скважину в толще льда и исследовали особенности намерзания (вспоминается файл "Лёд" из "Секретных материалов"). Оказалось, что намерзание происходило этапами: например, на некоторой глубине обнаружилось намерзание

толщиной 10 метров, потом 50, потом 30 метров - подобно годичным кольцам деревьев.

Как же ученые интерпретировали полученные ими результаты толщин слоев намерзшего льда? Они сделали следующее достаточно простое предположение: допустим, толщина намерзания льда в зависимости от времени описывается суммой гармоник (синусоид), т.е.:

```
Намерзло льда (t) = S(a
i
×s
i
n(w
i
×t + b
i
));
```

где S означает сумму синусоид, ai - амплитуда i-ой синусоиды, wi - ее период, bi - ее начальная фаза.

Проведя соответствующие расчеты (называемые разложением в ряд Фурье, когда некоторую зависимость $f(x)$ представляют как сумму синусоид с различными частотами и амплитудами), ученые определили, что процесс намерзания хорошо описывается суммой всего трех гармоник с тремя различными периодами ! Это означает, что на нашей планете происходили (и происходят) некие периодические процессы, влияющие в том числе и на намерзание льда на полярных шапках. Так, один из периодов составил около 10 000 лет - следовательно, с такой периодичностью на Земле наступает некоторый процесс похолодания.

**Приближение экспериментальных данных аналитической зависимостью.**

**Критерий Хи-квадрат.**

Как же осуществить проверку достоверности выбранной аппроксимирующей функции? Допустим, у вас есть следующая зависимость:

```
X  Y
1  2
2  4
3  6
4  7
5  10
6  13
 ...
```

Разумно полагая, что это что-то вроде y=ax, причем скорее всего a=1, только точки x=4 и x=6 не совсем вписываются в эту гипотезу, вы хотели бы проверить свою теорию и получить что-то вроде примерного значения "a" и некоторую оценку достоверности своего предположения.

Каким образом можно осуществить такую проверку? Сформулируем задачу в наиболее общем виде: обозначим аргументы (входные данные) за Xi, значения зависимости (выходные данные) за Yi, погрешность значений за Eyi. Аппроксимирующую функцию (допускаемый аналитический вид зависимости) за YiTeop. Тогда назовем **критерием** достоверности аппроксимации экспериментальной зависимости Yi = function (Xi) теоретической зависимостью YiTeop = function (Xi) функцию:

```
Cr = function(X
i
,(Y
i
,Ey
i
),Y
i
Teop);
```

значение которой говорит о том, насколько точно хорошо теоретическое распределение YiTeop = function (Xi) описывает реальное распределение Yi = function (Xi).

В нашем простом случае у входных данных (X=1,2,3,4...) нет никаких погрешностей, равно как нет их и у выходных данных (Y=2,4,6,7,10,13). Теоретическое распределение имеет простой вид YTeop = aX с одним неизвестным параметром (a). Предполагаем a=2 и получаем ряд значений теоретического распределения:

```
YTeop(X) = 2X = 2,4,6,8,10,12
```

Полученные значения YTeop необходимо подставить в критерий-функцию Cr и по ее значению оценить достоверность выбранной аппроксимации.

Какой же вид должна иметь критерий-функция Cr? Допустим, чем меньше ее значение, тем достовернее аппроксимация. Очевидно, она должна возрастать тем больше, чем больше получается отклонение экспериментальных значений Y от аппроксимирующих их YTeop, учитывая все точки распределения.

Таких критерий-функций существует несколько. Приведу пример функции Xq ("Хи-квадрат"). Вот ее вид:

```
Xq = S( (Y
i
-YTeopi)
2
 / Ey
i
2
 )i=1..N
```

где S означает суммирование квадратов разницы между экспериментальными и теоретическими значениями, деленными на квадраты погрешностей измерения.

Оценим поведение этой функции. Чем больше отклонение теоретического приближения от экспериментального, тем больше значение Xq, т.е. чем больше значение Xq, тем менее достоверным является приближение.

А вот как будет выглядеть запись Xq для нашего случая с линейной (y=ax) зависимостью:

```
X  Y  YTeop=ax YTeop-Y (Y-YTeop)
2

1  2     2       0        0
2  4     4       0        0
3  6     6       0        0
4  7     8       1        1
5  10    10      0        0
6  13    12      -1       1
 ...
```

Итого в этом случае Xq=1+1=2.

Тонкий момент. Обратите внимание на то, что каждый квадрат отклонения делится на квадрат погрешности измерения Y - Eyi. Смысл в том, что разные точки распределения могут иметь **различные** ошибки, другими словами - давать различное количество информации о характере искомого распределения: чем больше погрешность точки, тем меньше ее вклад в оценку достоверности приближения. В нашем случае Eyi отсутствуют у всех точек, что эквивалентно тому, что все точки вносят одинаковый вклад в искомый вид теоретического распределения.

Оценку достоверности аппроксимации при помощи Xq можно определить по так называемым "таблицам Стьюдента", в которых приводится максимально допустимое значение Xq для аппроксимаций распределений с различным числом входных данных и различным числом параметров аппроксимирующего распределения. Однако должно быть ясно, что чем больше

параметров имеет приближение и чем больше входных данных, тем большим будет минимально допустимое значение Xq, определяющее достоверность аппроксимации.

В нашем случае говорят, что зависимость Y(x) = 2,4,6,7,10,13.. аппроксимируется функцией y=ax,a=2 с точностью Xq = 2 для 6 точек, число параметров аппроксимирующей функции равно 1 (это a=2).

**Применение критерия Хи-квадрат для аппроксимации.** Разберем более детально пример с аппроксимацией распределения линейной функцией y=ax и покажем, как можно найти искомый параметр распределения "a" не предполагая его значение, а вычислив его с помощью функции Xq.
**Способ первый. Аналитическое решение.**

Замечаем, что в случае приближения y=ax Xq - это функция от одного аргумента:

```
Xq = function(a);
```

поскольку множества Y и X фиксированы на момент расчета, а YTeop вычисляется как функция от X. Тогда для того, чтобы найти минимум функции Xq достаточно решить уравнение:

```
 dXq
 ---  = S(2×(Y
i
-aX
i
)×(-X
i
) ) = 0;
 da
```

найдя неизвестное "a".

В более общем случае теоретическое распределение может быть записано в виде:

```
YTeop = function( a
0
 , a
1
 , a
2
... ),
```

где a0..am - параметры искомого теоретического приближения. Очевидно, уравнение нахождения неизвестных коэффициентов aj можно записать в виде:

```
 dXq
 --- = 0,
 da1
 dXq
 --- = 0,
 da2
 ...
 dXq
 --- = 0,
 dam
```

т.е. получается система из m уравнений, образованных записью частной производной по каждому из a1..am параметру. Решая эти уравнения совместно, можно получить неизвестные параметры a1..am приближающего распределения.

**Способ второй. Численные методы.**

Несмотря на то, что Способ 1 может дать однозначное и наиболее точное значение параметров приближающего распределения aj, существует и другой способ определить неизвестные aj. Он заключается в поиске на некотором поле значений ajтаких, при которых значение Xq минимально. Поясню мысль рисунком:

```
a1 minimum -- a1 minimum+step_a1 -- a1 minimum+step_a1×2 -- ... a1 max
```

```
a2 minimum -- a2 minimum+step_a2 -- a2 minimum+step_a2×2 -- ... a2 max
...
am minimum -- am minimum+step_am -- am minimum+step_am×2 -- ... am max
```

Т.е. вначале задаются некие граничные (максимальные и минимальные) значения каждого из неизвестных параметров aj и шаг разбиения интервала (aj max - aj min) - step_aj. Каждое значение aj пробегает значения от минимума до максимума. Для каждой комбинации параметров aj (очевидно, всего возможно ((число шагов разбиения))число параметров aj вариантов) рассчитывается значение Xq и ищется такая комбинация, которая отвечает минимальному Xq

Позволю себе назвать такой способ поиском "на решетке"(lattice) значений неизвестных параметров aj. Значение step_aj назовем "шагом решетки".

После того, как на данной "решетке" значений параметров будут найдены параметры, отвечающие минимальному Xq, необходимо будет выполнить модификацию решетки с целью локализации значения каждого искомого параметра в области истинного значения параметра:

- Если значение параметра, отвечающего минимальному Xq будет обнаружено внутри решетки (т.е. он будет между минимальным и максимальным значением для данного параметра), то новые границы будут определены следующим образом:

Новое значение минимума/максимума = Значение параметра ± Шаг решетки

- Если значение параметра, отвечающего минимальному Xq будет обнаружено на краю решетки (т.е. он будет равен либо минимальному либо максимальному значению для данного параметра), то новые границы будут определены следующим образом:

Новое значение минимума/максимума = Значение параметра ± ширина решетки.

Рассмотрим пример того, как это бы происходило в случае нашей простой линейной зависимости Y(x) = 2,4,6,7,10,13.. Мы предполагаем, что у аппроксимирующей зависимости один параметр: a. Найдем его. Сначала зададим максимальное и минимальное начальное значение a:

```
 a
max
 = 100, a
min
 = -100;
```

число разбиений решетки пусть будет равно 5, тогда в первом приближении (на первой решетке) будет проверены следующие значения a:

-100 -50 0 +50 +100,

при этом шаг решетки равен:

(amax - amin) / (5-1) = ( 100 - (-100) ) / 4 = 50,

и мы вычислим пять значений функции Xq для всех значений a:

```
a     -100       -50        0          +50        +100
Xq(a) 9.47E+0005 2.46E+0005 3.74E+0002 2.09E+0005 8.73E+0005
```

Минимум Xq в данном случае получается если a=0. Поскольку точка лежит внутри решетки (a <> -100 и a <> 100), то новые максимальные и минимальные значения "a" необходимо выбрать следующим образом:

```
 a
max
 = 0 + 50, a
min
 = 0 - 50;
```

и перейти к следующему расчету Xq на новом интервале.

Чем больше будет выполнено таких приближений, тем точнее будут найдены искомые параметры, но до известного предела, заложенного в самом критерии Xq.

Программная реализация именно такого способа будет рассмотрена ниже. В чем его достоинства/недостатки по сравнению с Способом 1? Достоинством является универсальность относительно вида приближающей функции YTeop: нет необходимости решать сложную систему уравнений в частных производных по aj, недостатком является некоторая неточность результатов определения параметров aj, что будет показано ниже.

**Реализация аппроксимации данных полиномом численными методами.**

Реализуем программно решение следующей задачи: пусть имеется набор значений X,Y, предположительно образованных полиномом вида:

```
Y(x) = a
n
x
n
 + a
n-1
x
(n-1)
 + ... a
0
```

Попытаемся построить алгоритм, способный аппроксимировать заданную зависимость Y(X) полиномом вида Y(x) = anxn + .. +a0 : найти неизвестные коэффициенты an,an-1,..,a0 и дать оценку достоверности приближения с помощью критерия Xq.

Реализация алгоритма будет проведена на win32-ассемблере с использованием сопроцессора, все особенности работы с которым будут по возможности прокомментированы.

Заголовок программы на MASM: стандартное подключение библиотек и т.д.

```
 .386
 .model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
```

Описываем данные и константы программы.

```
; Флаги вывода промежуточных результатов (Debug):
; выводить значение степени полинома из файла
ShowPolinomPower_NOT_DEFINED equ TRUE
; выводить значения X,Y из файла
ShowXYAtFileReading_NOT_DEFINED equ TRUE
FloatMask equ 14 ; Маска для вывода чисел с плавающей точкой:
                 ; 14 цифр после десятичной точки
```

MaxPower equ 10 ; Максимальная степень входного полинома:

```
                 ; y(x)=a
```

0
+a
1
×x+a
2
×x
2
+...a

MaxPower
×x
MaxPower

; Размер решетки для вычислений:
LatticeSize equ 4 ; (число разбиений интервала поиска)
ImpLattices equ 500 ; Максимальное число итераций
MaxPoints equ 40 ; Максимальное число входных точек
.data
; Максимальное (по модулю) начальное значение коэффициентов
MaxStartCoffValue dd 1.0E+10
BreakValueOfXq dd 0.1 ; Величина, контролирующая момент, когда
                    ; приближение считается достоверным

coff struc ; Параметры полинома: его коэффициенты
Cofficient dd (MaxPower+1) dup (?)
coff ends
Point struc ; Описание одной точки: пара (X,Y)
X dd?
Y dd?
Point ends
.data?
Power dd? ; Предполагаемая степень из файла
EntryData db MaxPoints×sizeof(Point) dup(?) ; Входные точки (X,Y)
NumberDataPoints dd? ; Число точек в файле
CurrLattice dd? ; Текущий номер решетки
TempReal dd? ; Временная переменная
                    ; Массивы коэффициентов полинома:

amin coff <> ; Набор минимальных коэффициентов
amax coff <> ; Набор максимальных коэффициентов
awork coff <> ; Набор промежуточных коэффициентов
afoundmin coff <> ; Набор коэффициентов, отвечающих минимальному Xq
Xq dd? ; Значение Xq
Xqmin dd? ; значение минимального Xq на очередном шаге вычислений
; Набор индексов для прохождения узлов решетки
LatticeIndexes dd (MaxPower+1) dup(?)
; Индексы, отвечающие меньшему Xq
NewIndexes dd (MaxPower+1) dup(?)

```
  Константы ShowPolinomPower и ShowXYAtFileReading определяются для вывода отладочных сообщений при
      чтении параметров из файла. Они начнут работать, если удалить "_NOT_DEFINED".

  FloatMask управляет выводом чисел с плавающей точкой: в данном случае эта величина определяет вывод
      14 знаков после десятичной точки.

  MaxPower определяет максимально допустимую степень аппроксимирующего полинома, поддерживаемого
      программой - 10. MaxPoints определяет максимально возможный размер входных данных (точек x,y)

  LatticeSize является количеством разбиений интервала [максимальное значение \- минимальное
      значение] каждого коэффициента полинома. ImpLattices определяет число итераций в численном
      поиске минимального Xq (число анализируемых решеток). MaxStartCoffValue определяет начальные
      значения всех коэффициентов полинома: вначале полагаем макисмально/минимально возможными
      значения коэффициентов равными ± MaxStartCoffValue. BreakValueOfXq определяет точность
      определения коэффициентов полинома: чем больше этот порог, тем хуже будет определены
      коэффициенты полинома: на самом деле эта величина должна быть согласована с таблицей Стьюдента.

   .data
  ErrTitle  db "Error? ;)",0 ; Заголовок сообщения отладки/ошибки
```

```
 .code
start:
 .data
  AppName db "Splain test program",0
  Welcome db "Example of Approximate Data by Polinom. Coded by Chingachguk, 2002.",10,10
          db "Entry Data(must be in file 'splain.dat'):",10
          db "<polinom power>",10
          db "<X1>,<Y1>",10
          db "...",10
          db "<Xn>,<Yn>",0
 .code
  invoke MessageBox,NULL,addr Welcome,addr AppName,MB_OK+MB_ICONINFORMATION
 .data
IniFileName db "splain.dat",0 ; Имя файла данных
 .data?
IniDescr dd?           ; Дескриптор файла
 .code
; ×××××××××××××× Read Entry File ××××××××××××××××××××××××××××
; Open entry file (not Create !)
  invoke CreateFile,offset IniFileName,GENERIC_WRITE or GENERIC_READ,\
    FILE_SHARE_WRITE or FILE_SHARE_READ,0,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0
  mov  IniDescr,eax     ; Save file descriptor
  cmp  eax,0ffffffffh
  jz   FileNotOpen
 .data?
CurrString dd?
FileStringSize equ 80
FileString db FileStringSize dup(?)
 .code
  ; Ведем счет строкам входного файла
  mov  dword ptr CurrString,1
  ; Читаем предполагаемую степень полинома (max: MaxPower)
  push FileStringSize   ; размер приемника
  push offset FileString ; Адрес приемника строки
  push IniDescr         ; Дескриптор файла
  call ReadFileString
  ; Переводим строку со степенью в число (real)
  push offset Power
  push offset FileString
  call StringToVal
  jc   InvalidNumericFormat
  ; Переводим степень из real в int
  fld  dword ptr Power
  ; Задаем режим округления (биты 11 - отбросить дробную часть)
  call Set_FPUTruncType
  frndint
  fistp dword ptr Power
  ; Проверяем степень на валидность
  cmp  dword ptr Power,0
  jl   InvalidPowerValue
  cmp  dword ptr Power,MaxPower
  ja   InvalidPowerValue
; ××× Отладочное сообщение: значение предполагаемой степени полинома ×××
ifdef ShowPolinomPower
 .data
PolinomPowerName db "Power: "
 .code
  mov  edi,offset FileString
  mov  esi,offset PolinomPowerName
  mov  ecx,sizeof PolinomPowerName
  cld
  rep  movsb
  mov  eax,Power
  mov  ebx,10000000
  call DecChar
  mov  byte ptr [edi+8],0
  invoke MessageBox,0,offset FileString,offset ErrTitle,MB_OK
endif
  ; Читаем пары точек (X,Y) из файла
  mov  dword ptr NumberDataPoints,0
  ; Стартовое значение базы массива EntryData (ebp)
```

```
  xor  ebp,ebp
@@ReadPoints:
  ; Ведем счет строкам входного файла
  inc  dword ptr CurrString
  ; Читаем X
  push FileStringSize   ; размер приемника
  push offset FileString ; Адрес приемника строки
  push IniDescr         ; Дескриптор файла
  call ReadFileString
  test eax,eax
  jz   @@ReadPointsDone
  ; Переводим строку в число (real)
  lea  eax,EntryData.Point[ebp].X
  push eax
  push offset FileString ; Адрес строки
  call StringToVal
  jc   InvalidNumericFormat
  ; Читаем Y
  push FileStringSize   ; размер приемника
  push offset FileString
  push IniDescr
  call ReadFileString
  test eax,eax
  jz   @@ReadPointsDone
  ; Переводим строку в число (real)
  lea  eax,EntryData.Point[ebp].Y
  push eax
  push offset FileString
  call StringToVal
  jc   InvalidNumericFormat
; ××× Отладочное сообщение: значения X,Y ×××
ifdef ShowXYAtFileReading
  cld
  mov  edi,offset FileString
  mov  eax,' :X '
  stosd
  push FloatMask  ; Маска вывода
  lea  eax,EntryData.Point[ebp].X
  push eax   ; Число
  push edi   ; Изображение числа
  call ValToString
  push FloatMask
  lea  eax,EntryData.Point[ebp].Y
  push eax ; Число
  mov  esi,offset FileString
  call Str_Len
  lea  edi,[esi+ecx]
  mov  eax,' :Y '
  stosd
  push edi   ; Изображение числа
  call ValToString
  invoke MessageBox,0,offset FileString,offset ErrTitle,MB_OK
endif
  ; Максимальное число входных точек не должно быть больше MaxPoints
  inc  dword ptr NumberDataPoints
  cmp  dword ptr NumberDataPoints,MaxPoints
  ja   InvalidPointsValue
  ; Индексируем следующие точки в массиве
  add  ebp,sizeof Point
  jmp  @@ReadPoints
@@ReadPointsDone:
; ×××××××××× approximate Data ××××××××××××
; Прочитав все точки из файла, задаем начальные значения коэффициентов
; set initial coefficient's values (равные ±MaxStartCoffValue)
  mov  ecx,Power
  inc  ecx
  xor  esi,esi
@@SetStartCoffs:
  fld  MaxStartCoffValue
  fchs                  ; Сделать отрицательным
  fstp amin.Cofficient[esi]
```

```
  fld  MaxStartCoffValue
  fstp amax.Cofficient[esi]
  ; Следующий коэффициент
  add  esi,sizeof coff / (MaxPower+1)
  loop @@SetStartCoffs
; ××××××× Основной цикл: поиск коэффициентов, отвечающих минимальному Xq ×××××××
; В этом цикле мы вычисляем Xq на всех узлах решетки,
; находим минимальный Xq и запоминаем отвечающие ему
; значения коэффициентов решетки,
; затем сжимаем/сдвигаем решетку минимального Xq в зависимости от того,
; лежат ли коэффициенты на краю (внутри) решетки.
; Если значение Xq показывает, что приближение достоверно,
; мы заканчиваем основной цикл поиска
  ; Номер решетки приближения
  mov  dword ptr CurrLattice,1
NextLattice:
  ; find Xq minimum at the current Lattice
  ; Получить стартовое значение Xq
  ; (Получить значение полинома с минимальными коэффициентами (массив amin))
  fldz
  fstp Xqmin
  xor  esi,esi          ; Индекс - esi - номер точки из файла
  mov  ecx,NumberDataPoints ; Число точек в файле
@@GetStartXq:
  push dword ptr EntryData.Point[esi].X ; значение аргумента (X)
  push Power            ; размер (степень) полинома
  push offset amin      ; адрес массива коэффициентов полинома
  call polinom
  ; Результат: значение полинома в ST(0)
  ; вычислить разницу -(Yi-polinom(Xi))
  fsub dword ptr EntryData.Point[esi].Y
  ; Возвести в квадрат разницу -(Yi-polinom(Xi)), она в ST(0)
  mov  eax,2                        ; степень - 2
  call get_pow          ; вернется в ST(0) квадрат
  fadd dword ptr Xqmin  ; Прибавить к накопителю Xqmin
  fstp dword ptr Xqmin  ; Запомнить накопитель
  add  esi,sizeof Point
  loop @@GetStartXq
  ; Обнуляем стартовые коэффициенты для прохода по значениям коэффициентов
  mov  ecx,Power
  inc  ecx
  mov  edi,offset LatticeIndexes
  cld
  xor  eax,eax
  rep  stosd
  ; выполняем поиск Xq на текущих значениях коэффициентов
FindXqMinimum:
  ; Определяем коэффициенты текущего приближенного полинома (awork)
  xor  esi,esi          ; Индексы - esi, edi
  xor  edi,edi
  mov  ecx,Power        ; По всем степеням
  inc  ecx
@@GetCurrentCoffs:
  ; вычисляем разницу между максимальным и минимальным значением коэффициента
  fld  amax.Cofficient[esi]
  fsub amin.Cofficient[esi]
  ; Делим разницу на число (узлов решетки-1), получая шаг решетки
  mov  TempReal,LatticeSize-1
  fidiv dword ptr TempReal
  ; Умножаем на номер узла текущего коэффициента
  fimul dword ptr LatticeIndexes[edi]
  ; Получаем значение коэффициента в узле, добавляем минимальное значение
  fadd dword ptr amin.Cofficient[esi]
  ; вычисляем текущие коэффициенты полинома (awork)
  fstp dword ptr awork.Cofficient[esi]
  ; Следующий коэффициент
  add  esi,sizeof coff / (MaxPower+1)
  add  edi,4            ; Следующий номер узла
  loop @@GetCurrentCoffs
  ; вычисляем Xq для промежуточных значений коэффициентов (awork)
  fldz
```

```
  fstp Xq
  xor  esi,esi          ; Индекс - esi - номер точки из файла
  mov  ecx,NumberDataPoints ; Число точек в файле
@@GetCurrentXq:
  push dword ptr EntryData.Point[esi].X
  push Power            ; размер (степень) полинома
  push offset awork     ; адрес массива коэффициентов полинома
  call polinom
  ; Результат: значение полинома в ST(0)
  ; вычислить разницу -(Yi-polinom(Xi))
  fsub dword ptr EntryData.Point[esi].Y
  ; Возвести в квадрат разницу -(Yi-polinom(Xi)), она в ST(0)
  mov  eax,2            ; степень - 2
  call get_pow          ; вернется в ST(0) квадрат
  fadd dword ptr Xq     ; Прибавить к накопителю Xqmin
  fstp dword ptr Xq
  add  esi,sizeof Point
  loop @@GetCurrentXq
  ; Сравниваем Xq и Xqmin
  fld  Xq
  fcomp dword ptr Xqmin ; Сравнить Xq с Xqmin и вытолкнуть из стека ST(0)
  fstsw ax              ; Перенести флаги в регистр ax
  sahf                  ; Дублировать их в регистре флагов
  jnc  @@CurrXqNoLow
  ; Если Xq < Xqmin, то присвоить новое значение Xqmin
  ; Т.е. мы нашли коэффициенты с меньшим Xq, запомним их
  fld  dword ptr Xq
  fstp dword ptr Xqmin
  ; Также запомнить текущие коэффициенты в afoundmin
  mov  ecx,Power
  inc  ecx
  mov  esi,offset awork
  mov  edi,offset afoundmin
  cld
  push ecx
  rep  movsd
  pop  ecx
  ; Запомнить текущие номера узлов решетки
  mov  esi,offset LatticeIndexes
  mov  edi,offset NewIndexes
  rep  movsd
@@CurrXqNoLow:
  ; Инкрементировать номера узлов решетки,
  ; перебирая все узлы решетки
  xor  esi,esi
@@NextApprox:
  inc  dword ptr LatticeIndexes[esi*4]
  cmp  dword ptr LatticeIndexes[esi*4],LatticeSize
  jb   FindXqMinimum    ; Следующая решетка
  mov  dword ptr LatticeIndexes[esi*4],0
  inc  esi
  cmp  esi,dword ptr Power
  jbe  @@NextApprox
  ; Мы прошли по всем узлам решетки
  ; Анализируем, где был найден минимум Xq
  mov  ecx,Power
  inc  ecx
  xor  esi,esi
@@AnaliseLattice:
  mov  eax,dword ptr NewIndexes[esi]
  test eax,eax
  jz   @@XqMinLieOnRange
  cmp  eax,LatticeSize-1
  jnz  @@XqMinNoLieOnRange
@@XqMinLieOnRange:
  ; point lie on the rage of lattice: no change size of lattice !
  ; Строим решетку симметрично относительно найденного узла
  ; Новый минимум/максимум=найденный узел±ширина решетки
  ; Получаем в ST(0) новое значение минимального коэффициента
  fld  dword ptr afoundmin.Cofficient[esi]
  fsub dword ptr amax.Cofficient[esi]
```

```
  fadd dword ptr amin.Cofficient[esi]
  ; Получаем в ST(0) новое значение максимального коэффициента, мин.->ST(1)
  fld  dword ptr afoundmin.Cofficient[esi]
  fadd dword ptr amax.Cofficient[esi]
  fsub dword ptr amin.Cofficient[esi]
  jmp  @@NextLatticePoint
@@XqMinNoLieOnRange:
  ; point lie inside of lattice: minimize size of lattice
  ; Строим решетку симметрично относительно найденного узла
  ; Новый минимум/максимум=найденный узел±шаг решетки
  ; Получаем в ST(0) новое значение минимального коэффициента
  fld  dword ptr amin.Cofficient[esi]
  fsub dword ptr amax.Cofficient[esi]
  ; Делим разницу на число (узлов решетки-1), получая шаг решетки
  mov  TempReal,LatticeSize-1
  fidiv dword ptr TempReal
  fadd dword ptr afoundmin.Cofficient[esi]
  ; Получаем в ST(1) новое значение максимального коэффициента, ST(0)->ST(1)
  fld  dword ptr amax.Cofficient[esi]
  fsub dword ptr amin.Cofficient[esi]
  ; Делим разницу на число (узлов решетки-1), получая шаг решетки
  fidiv dword ptr TempReal
  fadd dword ptr afoundmin.Cofficient[esi]
@@NextLatticePoint:
  ; ST(0) - новый максимум, ST(1) - новый минимум
  fstp dword ptr amax[esi] ; <-ST(0)
  fstp dword ptr amin[esi] ; <-ST(0)
  add  esi,4
  loop @@AnaliseLattice
  ; Переходим к следующей решетке, выполняя не более ImpLattices приближений
  inc  dword ptr CurrLattice
  ; Проверяем величину текущего минимума Xq
  ; Возможно, мы уже нашли коэффициенты
  fld  dword ptr BreakValueOfXq
  fmul dword ptr Power
  fadd dword ptr BreakValueOfXq
  fimul dword ptr NumberDataPoints
  ; ST(0)=NumberDataPoints × BreakValueOfXq × (Power+1)
  ; Если величина Xqmin =< NumberDataPoints×BreakValueOfXq×(Power+1),
  ; то приближение считается достоверным
  ; Сравнить NumberDataPoints×BreakValueOfXq×(Power+1) с Xqmin и вытолкнуть из стека ST(0)
  fcomp dword ptr Xqmin
  fstsw ax              ; Перенести флаги в регистр ax
  sahf                  ; Дублировать их в регистре флагов
  jnc  @@ApproximateValid
  cmp  dword ptr CurrLattice,ImpLattices
  jbe  NextLattice
@@ApproximateValid:
; ×××××××××× Цикл поиска коэффициентов закончен ××××××××××××××××
; выводим полученные коэффициенты, значение Xq для анализа достоверности
; нашего приближения данных из файла полиномом
 .data
ResultTitle db "Calculations resume",0
ResultHeader0 db "Assume that you data was approximated by polinom with power????.",10,10
ResultHeaderD db "a[?]=?.??????????????????  "
ResultHeaderE db "Xq =?.???????????????????? for????? points.",10
 .data?
ResultHeader db 1000 dup(?)
 .code
  cld
  mov  edi,offset ResultHeader
  ; выводим степень полинома
  push edi
  mov  eax,Power
  mov  ebx,1000
  mov  edi,offset ResultHeader0+60
  call DecChar
  pop  edi
  mov  esi,offset ResultHeader0
  mov  ecx,sizeof ResultHeader0
  rep  movsb
```

```
  ; выводим полученные коэффициенты
  mov  ecx,Power        ; По всем степеням
  inc  ecx
  xor  esi,esi
@@OutCoffs:
  pushad
  mov  eax,Power
  inc  eax
  sub  eax,ecx
  mov  ebx,1
  mov  edi,offset ResultHeaderD+2
  call DecChar
  push FloatMask        ; Маска вывода
  lea  eax,afoundmin.Cofficient[esi]
  push eax
  push offset ResultHeaderD+6 ; Изображение числа
  call ValToString
  popad
  push esi
  push ecx
  mov  esi,offset ResultHeaderD
  mov  ecx,sizeof ResultHeaderD
@@CopyCoff:
  lodsb
  test al,al
  jnz  @@UsalSym
  mov  al,10
  mov  ecx,1
@@UsalSym:
  stosb
  loop @@CopyCoff
  pop  ecx
  pop  esi
  ; Следующий коэффициент
  add  esi,sizeof coff / (MaxPower+1)
  loop @@OutCoffs
  mov  al,10
  stosb
  ; выводим Xqmin
  push edi
  push FloatMask ; Маска вывода
  push offset Xqmin ; Число
  push offset ResultHeaderE+5 ; Изображение числа
  call ValToString
  mov  byte ptr ResultHeaderE+27,' '
  mov  eax,NumberDataPoints
  mov  ebx,10000
  mov  edi,offset ResultHeaderE+32
  call DecChar
  pop  edi
  mov  esi,offset ResultHeaderE
  mov  ecx,sizeof ResultHeaderE
  rep  movsb
  xor  al,al
  stosb
  invoke MessageBox,0,offset ResultHeader,offset ResultTitle,MB_OK+MB_ICONINFORMATION

Exit:
  invoke CloseHandle,IniDescr
  invoke ExitProcess,NULL
;×××××××××××× Подпрограмма вычисления значения полинома ××××××××××××××××××××××
; Get value of LocCoff[i]×x
i
, i=0..power
; [ebp+8] - адрес массива коэффициентов полинома
; [ebp+0Ch] - размер (степень) полинома
; [ebp+10h] - значение аргумента (X)
; Результат: значение полинома в ST(0)
polinom proc
  push ebp
  mov  ebp,esp
```

```
  ; result: [ebp-4], локальная степень полинома: [ebp-8]
  sub  esp,2*4
  push esi
  ; Определяем начальный результат
  fldz
  fstp dword ptr [ebp-4]
  ; Загружаем адрес коэффициентов полинома
  mov  esi,[ebp+8]
  ; Степень полинома
  fldz
  fstp dword ptr [ebp-8]
@@polinom:
  ; Получить X
Текущая степень (0..[ebp+0Ch])

  fld  dword ptr [ebp+10h] ; Загрузить X
  mov  eax,dword ptr [ebp-8] ; степень
  call get_pow          ; вернется в ST(0)
  ; Умножить на текущий коэффициент
  fmul dword ptr [esi]
  ; Добавить сумматор
  fadd dword ptr [ebp-4]
  ; Запомнит новый результат
  fstp dword ptr [ebp-4]
  add  esi,sizeof coff / (MaxPower+1) ; Следующий коэффициент
  inc  dword ptr [ebp-8]
  mov  eax,dword ptr [ebp-8]
  cmp  eax,[ebp+0Ch]
  jbe  @@polinom
  ; Возвращаем в ST(0) результат
  fld  dword ptr [ebp-4]
  pop  esi
  leave
  ret  3*4
polinom endp
;×××××××××××××××××××××××××××××× Сообщения об ошибках ××××××××××××××××××××××
FileNotOpen:
 .data
IniOpOK   db "Entry Data File (splain.dat) not open (not exist?) !",0
 .code
  invoke MessageBox,0,offset IniOpOK,offset ErrTitle,MB_OK or MB_ICONERROR
  jmp  Exit
InvalidNumericFormat:
 .data
InvalidNumericFormatMsg db "Invalid numeric format in string "
ErrStrNum               db "00000000",0
 .code
  mov  eax,CurrString
  mov  ebx,10000000
  mov  edi,offset ErrStrNum
  call DecChar
  invoke MessageBox,0,offset InvalidNumericFormatMsg,offset ErrTitle,MB_OK or MB_ICONERROR
  jmp  Exit
InvalidPowerValue:
 .data
InvalidPowerValueMsg db "Invalid Power Value",0
 .code
  invoke MessageBox,0,offset InvalidPowerValueMsg,offset ErrTitle,MB_OK or MB_ICONERROR
  jmp  Exit
InvalidPointsValue:
 .data
InvalidPointsValueMsg db "Too much data points in file !",0
 .code
  invoke MessageBox,0,offset InvalidPointsValueMsg,offset ErrTitle,MB_OK or MB_ICONERROR
  jmp  Exit
```

Кратко о вспомогательных программах. Для реализации алгоритма нам потребуется несколько подпрограмм:

• **семейство Set_FPU..Type.** Подпрограммы для задания режима округления сопроцессора при операциях конвертирования чисел с плавающей точкой в числа типа inteher.
• **get_pow**. Подпрограмма возведения в целую степень числа с плавающей точкой. Реализована примитивно умножением аргумента самого на себя степень раз.
• **StringToVal** : Подпрограмма деформатирования строки в число типа Real. Позволяет читать из файла числа с плавающей запятой в виде строки и преобразовывать их в формат real (4 байта). Реализована следующим образом: накопитель инициализируется числом 0, затем читается цифра из строки, добавляется к приемнику, приемник предварительно доумножается на 10. Десятичная точка учитывается подсчетом числа цифр после нее и финальным делением на 10 в степени "число цифр после десятичной точки".
• **ValToString** : Подпрограмма форматирования числа типа Real в строку. Позволяет выполнять форматный вывод чисел с плавающей точкой с заданным числом знаков после десятичной точки. Алгоритм вывода довольно прост и, возможно, допускает некоторую погрешность: определяется максимальная степень десяти в числе при помощи команды fyl2x сопроцессора: фактически вычисляется max_power = int( log10(x) ). Далее последовательным делением на 10 в степени max_power, max_power-1, ... определяются коэффициенты разложения числа по степеням десяти.
• **ReadFileString** : парсер входного файла. Позволяет разделять строки с числовыми данными из входного файла, отделенные стандартными разделителями (10 или 13+10 или пробелами) и читать их в заданный буфер.

```
; ××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××
; Вспомогательные подпрограммы
; ××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××××
 .data
  DecNumber dd 10.0
; ×××××××××××× Set_FPU...Type: Задать режим округления сопроцессора ××××××××××××
; Set_FPURoundType: округление до ближайшего целого (00)
; Set_FPUTruncType: отбрасывание дробной части (11)
 .data?
control dw?
 .code
Set_FPURoundType proc
  fstcw word ptr control
  and word ptr control,0F3FFh
  fldcw word ptr control
  ret
Set_FPURoundType endp
Set_FPUTruncType proc
  fstcw word ptr control
  or  word ptr control,0C00h
  fldcw word ptr control
  ret
Set_FPUTruncType endp
; ×××××××××××× get_pow: Подпрограмма возведения в целую степень числа ××××××××××
; ST(0) - аргумент, eax - степень
; result: ST(0)
; Для возведения в произвольную стпень числа следует воспользоваться
; более сложным алгоритмом с использованием fyl2x:
; см., например, П.И. Рудаков, К.Г. Финогенов
; "Программируем на языке ассемблера IBM PC"
 .code
get_pow proc
  test eax,eax
  jnz  @@NoZeroPower
  fstp ST(0)           ; вытолкнуть аргумент из стека
  fld1                  ; Загрузить 1.0
  ret                   ; Вернуть 1.0
@@NoZeroPower:
  push ecx
  mov  ecx,eax
  ; Сделать степень положительной
@@NegLoop:
  neg  ecx
  js   @@NegLoop
```

```
  fld1                  ; Загрузить 1.0, ST(0) - накопитель
@@get_pow:
  fmul ST(0),ST(1)      ; Накопитель=Накопитель x arg
  loop @@get_pow
  fxch ST(1)            ; Поменять накопитель и аргумент
  fstp ST(0)            ; вытолкнуть аргумент из стека
  ; Если степень отрицательна, необходимо Накопитель=1/Накопитель
  cmp  eax,0
  jge  @@NoNegPow
  fld1                  ; Загрузить 1
  fxch ST(1)            ; Обменять ST(1) и ST(0)
  fdivp ST(1),ST(0)     ; Разделить 1 на Накопитель
@@NoNegPow:
  pop  ecx
  ret
get_pow endp
; ×××× StringToVal: Подпрограмма деформатирования строки в число типа Real ×××××
; [ebp+8] - Адрес строки(Zero-terminated)
; [ebp+0Ch] - Адрес приемника (real или dd)
; Десятичным разделителем считается символ "."
; Пробелы перед цифрами пропускаются
; Ошибка - флаг CF установлен
 .code
StringToVal proc
  push ebp
  mov  ebp,esp
  sub  esp,1*4          ; Число из строки [ebp-4]
  mov  ebx,[ebp+0Ch]    ; Адрес приемника храним в ebx
  fldz                  ; Обнуляем приемник
  fstp dword ptr [ebx]
  cld
  ; Запоминаем знак числа
  xor  edi,edi
  mov  esi,dword ptr [ebp+8]
  lodsb
  dec  esi
  cmp  al,'-'
  jnz  @@NoSignBefore
  ; Устанавливаем 31-ый бит знака числа
  inc  edi
  ror  edi,1
  inc  esi
@@NoSignBefore:
  xor  edx,edx          ; Флаг десятичной точки и число знаков после нее
@@ParseString:
  xor  eax,eax
  lodsb
  test al,al
  jz   @@EndParse
  cmp  al,' '           ; Пропускаем пробелы слева
  jz   @@ParseString
  cmp  al,'.'
  jnz  @@NoDecPoint
  test dh,dh
  jnz  @@ErrorSign
  mov  dx,256
  jmp  @@ParseString
@@NoDecPoint:
  sub  al,'0'
  js   @@ErrorSign
  cmp  al,9
  ja   @@ErrorSign
  mov  dword ptr [ebp-4],eax
  fld  dword ptr [ebx]  ; Загружаем текущее значение результата
  fmul DecNumber        ; Умножаем результат на 10
  fiadd dword ptr [ebp-4] ; Добавляем к результату текущее число
  fstp dword ptr [ebx]  ; Запоминаем получившееся значение
  ; Учитываем число знаков после десятичной точки
  inc  dl
  jmp  @@ParseString
@@EndParse:
```

```
  ; Учесть десятичную точку
  test dh,dh
  jz   @@ThereAintPoint
  mov  dh,0
  ; Разделить результат на 10
edx

  fld  dword ptr DecNumber ; Загрузить аргумент (10)
  mov  eax,edx          ; Целая степень
  call get_pow          ; 10
edx
 вернется в ST(0)
  fld  dword ptr [ebx]  ; Загрузить результат без точки
  fxch ST(1)            ; Обменять ST(1) и ST(0)
  fdivp ST(1),ST(0)
  fstp dword ptr [ebx]  ; Res'=Res/10
edx

@@ThereAintPoint:
  ; Учесть знак числа
  or   dword ptr [ebx],edi
  clc
  leave
  ret  2*4
@@ErrorSign:
  stc
  leave
  ret  2*4
StringToVal endp
; ××××× ValToString: Подпрограмма форматирования числа типа Real в строку ××××××
; [ebp+8] - Адрес строки(будет Zero-terminated) в виде 12345.89
; [ebp+0Ch] - Адрес числа типа real (или dd)
; [ebp+10h] - Маска вывода числа (byte)
ValToString proc
 .code
  push ebp
  mov  ebp,esp
  ; Локальные переменные:
  ; [ebp-4] - модуль числа
  ; [ebp-08] - максимальная степень 10-ти в числе
  ; [ebp-0Ch] - временная
  sub  esp,4*3
  cld
  mov  edi,dword ptr [ebp+8] ; Адрес выходной строки
  mov  ebx,[ebp+0Ch]    ; Загружаем адрес числа в ebx
  mov  edx,[ebx]        ; Читаем его в edx
  ; Определяем знак, удаляя его одновременно (31 бит)
  shl  edx,1
  jnc  @@NoSign
  mov  al,'-'
  stosb
@@NoSign:
  shr  edx,1
  mov  [ebp-4],edx      ; Запоминаем модуль числа в локальной переменной
  ; Проверка на ноль
  test edx,edx
  jnz  @@NonZero
  mov  al,'0'
  stosb
  jmp  @@DoneValToStr
@@NonZero:
  ; Определяем старшую степень десяти - Log10(x) = Log2(x)/Log2(10)
  ; выполняется в два приема, поскольку нет операции Log10 в FPU,
  ; а есть y×Log2(x)
  ; Получаем 1/Log2(10)
  fld1                  ; Загрузить 1.0
  fldl2t                ; Загрузить log2(10)
  fdivp ST(1),ST(0)     ; Разделить 1/log2(10), результат в ST(0)
  fld  dword ptr [ebp-4] ; Загрузить наше число, 1/log2(10) протолкнуть в ST(1)
  fyl2x ; ST(0)=ST(1)×log2(ST(0))
  ; Задаем режим округления (биты 11 - отбросить дробную часть)
```

```
  call Set_FPUTruncType
  frndint
  fistp dword ptr [ebp-8] ; выгрузить степень как целое со знаком
  ; Задаем режим округления (биты 00 - округлить до ближайшего целого)
  call Set_FPURoundType
  ; Сохранить степень числа
  push dword ptr [ebp-8]
  ; вывести число, деля его на 10
...

  ; вывести целую часть числа, точку, дробную часть
  mov  ecx,dword ptr [ebp+10h] ; Сколько цифр после десятичной точки выводить
  inc  ecx              ; Цифра перед десятичной точкой
  xor  edx,edx          ; Флаг вывода точки
@@OutNumbers:
  fld  dword ptr DecNumber ; Загрузить 10 - основание
  mov  eax,dword ptr [ebp-8] ; Степень
  call get_pow          ; ST(0) = 10
Степень

  fld  dword ptr [ebp-4] ; Загрузить число
  ; ST(0)=Число, ST(1) = 10
Степень

  fdiv ST(0),ST(1)      ; ST(0)=ST(0)/ST(1)
  ; Задаем режим округления (биты 11 - отбросить дробную часть)
  call Set_FPUTruncType
  frndint
  fistp dword ptr [ebp-0Ch] ; выгружаем как целое без округления
  ; Задаем режим округления (биты 00 - округлить до ближайшего целого)
  call Set_FPURoundType
  ; выводим цифру
  mov  eax,dword ptr [ebp-0Ch]
  mov  ebx,1
  call DecChar          ; В формате - @n1
  inc  edi
  ; выводим точку, если это необходимо
  test edx,edx
  jnz  @@PointAlreadyOut
  mov  al,'.'
  stosb
  inc  edx
@@PointAlreadyOut:
  ; вычесть из числа 10
Степень(сейчас в ST(0))
 x Текущая цифра
  fimul dword ptr [ebp-0Ch]
  fld  dword ptr [ebp-4] ; Загрузить число
  fsub ST(0),ST(1)
  fstp dword ptr [ebp-4] ; выгрузить число с вычетом
  fstp ST(0)             ; Очистить стек
  ; Уменьшить степень на 1
  dec  dword ptr [ebp-8]
  loop @@OutNumbers
  ; Восстановить степень числа
  pop  dword ptr [ebp-8]
  ; вывести степень 10-ти
  mov  ax,'+E'
  mov  edx,[ebp-8]
  cmp  edx,0
  jge  @@NoNegPower
  neg  edx
  mov  ah,'-'
@@NoNegPower:
  stosw
  mov  eax,edx
  mov  ebx,1000
  call DecChar           ; В формате @n4
  add  edi,4
@@DoneValToStr:
  mov  byte ptr [edi],0
```

```
  leave
  ret  3*4
ValToString endp
; ××××××××××××× ReadFileString: Подпрограмма чтения строки из файла ××××××××××××
; [ebp+8] - Дескриптор файла
; [ebp+0Ch] - Адрес приемника строки
; [ebp+10h] - размер приемника
; -> eax реальный размер строки в приемнике и ноль в конце приемника
; Разделителями считаются символы 0A или 0D,0A, пробел.
; Пробелы слева читаются до любого другого символа,
; затем считываются символы числа, пробел служит знаком конца строки
ReadFileString proc
 .code
  push ebp
  mov  ebp,esp
  ; Переменная для байта из файла размером в 1 байт [ebp-8]
  ; Число байт, прочтенных ReadFile [ebp-4]
  ; Число байт в строке [ebp-0Ch]
  sub  esp,3*4
  cld
  mov  edi,[ebp+0Ch]
  mov  dword ptr [ebp-0Ch],0
  ; Флаг чтения любого символа, кроме пробела
  xor  esi,esi
@@NextByte:
  push NULL
  lea  eax,[ebp-4]
  push eax
  push 1
  lea  eax,[ebp-8]
  push eax
  push dword ptr [ebp+8]
  call ReadFile
  cmp  dword ptr [ebp-4],0
  jz   @@EndRead
  mov  al,byte ptr [ebp-8]
  cmp  al,0Ah
  jz   @@EndRead
  cmp  al,0Dh
  jz   @@NextByte        ; Пропускаем байт 0Ah
  cmp  al,' '            ; Пробел?
  jnz  @@NoFiller
  test esi,esi
  jnz  @@EndRead         ; Пробел после строки - break
  jmp  @@AfterFiller     ; Пробелы перед строкой заносим в буффер
@@NoFiller:
  inc  esi               ; Установим флаг, что был символ <> пробел
@@AfterFiller:
  mov  ebx,dword ptr [ebp-0Ch]
  inc  ebx
  cmp  ebx,dword ptr [ebp+10h]
  jae  @@EndRead
  inc  dword ptr [ebp-0Ch]
  stosb
  jmp  @@NextByte
@@EndRead:
  mov  byte ptr [edi],0
  mov  eax,dword ptr [ebp-0Ch]
  leave
  ret  3*4
ReadFileString endp
; ××××××××××××× DecChar: Подпрограмма форматирования строки из числа ××××××××××
; eax=number to digit, edi=offset result string in format 00000000(@n[ebx])
; ebx=начальный делитель
DecChar proc
  pushad
  pushfd
  cld
@GetDec:
  xor  edx,edx
  div  ebx
```

```
  add  al,'0'
  stosb
  push edx
  mov  eax,ebx
  xor  edx,edx
  mov  ebx,10
  div  ebx
  mov  ebx,eax
  pop  eax
  test ebx,ebx
  jnz  @GetDec
  popfd
  popad
  ret
DecChar endp
; xxxxxxxxxxxxxxxxxxxxx Str_Len: Get lenght of string xxxxxxxxxxxxxxxxxxxxxxx
; esi=addr of string; result: ecx=length
Str_Len proc
  xor  ecx,ecx
  push eax
  push esi
@@GetStrLen:
  lodsb
  test al,al
  jz   @@ExitStrLen
  inc  ecx
  jmp  @@GetStrLen
@@ExitStrLen:
  pop  esi
  pop  eax
  ret
Str_Len endp
end start
```

Оценим работоспособность алгоритма на его способности аппроксимировать Y(X) полиномом вида Y(x) = a2x2 + a1x+a0 (параболой): найти неизвестные коэффициенты a2, a1, a0.

Для примера рассмотрим простую параболу:

```
Y(x) = 3x
2
 - 5x -8;
```

на интервале значений x={1..10}:

```
  X     Y
 1.0 -10.00
 2.0  -6.00
 3.0   4.00
 4.0  20.00
 5.0  42.00
 6.0  70.00
 7.0 104.00
 8.0 144.00
 9.0 190.00
10.0 242.00
```

Поместим эту зависимость Y(x) в файл входных данных программы (splain.dat), указав предполагаемую степень полинома первой строкой:

```
Содержимое входного файла splain.dat:
2
 1.0 -10.00
 2.0  -6.00
 ...
10.0 242.00
```

В результате программа должна выдать следующий результат (содержимое MessageBox'a):

```
Assume that you data was approximated by polinom with power 0002
a[0]= -7.282...
```

```
a[1]= -5.288...
a[2]= 3.0247...

Xq = 0.413... for 00010 points.
```

Как видно, программа достаточно точно смогла определить коэффициенты параболы. Заметим, однако, что несмотря на то, что входные данные являлись абсолютно точно параболой с опредленными коэффициентами, они не были определены на 100%. Почему так происходит? Дело в том, что критерий Xq выполняет подбор коэффициентов по величине квадратов отклонения точек экспериментальных данных от теоретических. Очевидно, что для небольших значений Y приближение будет менее точным, нежели чем для больших: см. например, то, что значение a[2] определено гораздо точнее, чем a[1] или тем более a[0].

**Оценка предложенного метода численной аппроксимации с помощью Xq.**

В результате проведенного теста оказывается, что предложенный способ достаточно точно позволяет определять параметры аппроксимирующей зависимости, по крайней мере позволяет выполнять предварительные оценки свойств экспериментальных зависимостей. Вместе с этим он претендует на некоторую универсальность алгоритма относительно выбора вида и параметров аппроксимирующей зависимости: она может быть практически сколь угодно сложной и иметь широкий спектр параметров; алгоритм не требует сложных математических вычислений.

# Стеганография - искусство пряток

**Автор:** GRRL / SET#26, пер. Aquila

**Дата:** 2003-07-03

*[ Примечание переводчика: надо сказать, что между оригиналом и переводом существует немалое количество расхождений, так как в процессе перевода оказалось, что многие языковые структуры, которые употребляет автор, не входят в арсенал переводчика, поэтому кое-где последний вставал в тупик и долго думал, как из него выбраться. Однако бросать наполовину переведенный текст было переводчик не решился, поэтому он сделал всё, от него зависящее, чтобы создать перевод, более или менее передающий суть оригинала. ]*

В этой статье мы поговорим о действительно интересной теме, которой во многих случаях не уделяется достаточно внимания, хотя сам предмет разговора имеет непосредственное отношение ко всему информационному сообществу.

Нет! Стеганография - это не наука, изучающая стегозавров, это наука скрывания сообщений, но не путем их превращения в нечто нераспознаваемое, как это происходит в криптографии, а путём того, чтобы они оставались незамеченными.

Перед тем как перейти непосредственно к сути дела, необходимо изложить некоторые определения того, что такое стеганография с точки зрения этимологии, с точки зрения информатики и с точки зрения науки.

Этимологически, термин "стеганография" происходит от греческого слова "stegos", которое означает крышу. Таким образом, значение слова "стеганография" - это "письменная крыша".

С научной точки зрения стеганография - это набор методов и техник, камуфлирующих сообщение так, чтобы оно прошло незамеченным.

С технической точки зрения электронная стеганография заключается в том, чтобы включить сообщение в другой файл, так чтобы... ла ла ла.

Большая часть этого документа будет посвящена тому, чтобы показать, что стеганография - это гораздо больше, чем включение одного файла в другой, как это утверждается на многих сайтах.

**[ Практики включения ака: как это сделать? ]**

Существует тысячи способов включить сообщение, звук или изображение в другой файл, но главное отличие этих методов состоит в том, какой тип файла используется в качестве "крыши". На сегодняшний день существуют алгоритмы для скрытия сообщения в gif, bmp, jpeg, mp3, wav, mpeg... и многих других форматах. Все они используют более или менее схожие принципы, и у всех есть значительные ограничения. Не думайте, что сможете спрятать всю Библию в фотографию своей семьи...

Разумеется, целью данной статьи не является, как включать сообщения, ни оценивать соответствующие алгоритмы, ни тестировать призванные для их осуществления программы, поскольку это всего лишь введение в эту прекрасную тему.

Рассмотрим простой пример включения сообщения в файл BMP формата, для чего нам понадобится фотография нашей тёщи (используйте фотографию СВОЕЙ тёщи! Если её у вас нет, используйте фотографию другой не менее важной персоны) и сообщение. Предполагаем, что фотография имеет размер 200x400 пикселей.

В этом формате каждый пиксель представляется байтом, который, как известно, состоит из восьми битов (например, 00110101), изображение воссоздаётся из матрицы, содержащей все эти пиксели (рекомендуется, чтобы не использовалось RLE-сжатие).

Мы можем представить, что часть матрицы выглядить следующим образом:

```
...
00010101 10100101 01010101 00110101 01110101 01000010 01010011 01101010
00001011 01010101 10100101 01010111 11010111 10000101 01010010 01010010
10101001 10101011 00001001 10100100 00010001 10100101 00010101 10100101
...
```

Каждый байт означает цвет, и в чём же трюк?... Мы знаем, что если мы чуть-чуть подправим цвета, то в результате изменение изображения не будет заметно. Также мы знаем, что изменение самого младшего бита почти не скажется на получившемся в результате изображении, поэтому именно его мы и будем использовать...

Хотя об этом не было сказано ранее, сообщение, которое мы будем включать в изображение - это "SET" (название ezine, в котором был опубликован оригинал данной статьи - прим. пер), поэтому длл того, чтобы сделать задуманное нами, необходимо три октета, по одному на каждую букву. В шестнадцатиричной системе последние выглядят так - 73 65 74, а в двоичной так - 01001001 01000001 01001010.

Таким образом, изображение меняется следующим образом (изменяется последний бит каждого байта):

```
00010100 10100101 01010100 00110100 01110101 01000010 01010010 01101011
       -        -        -        -        -        -        -        -
00001010 01010101 10100100 01010110 11010110 10000100 01010010 01010011
       -        -        -        -        -        -        -        -
10101000 10101011 00001000 10100100 00010001 10100100 00010101 10100100
       -        -        -        -        -        -        -        -
...
```

Как можно видеть, несмотря на данные изменения, это не оказало значительного влияния на изображение. Занимательно, правда?

Разумеется, рассмотренный выше пример очень базов, усложнения этого алгоритма призваны преодолеть неизбежные ограничения. А теперь поговорим вот о чём. Ранее я написал, что есть множество методов помещать сообщения в файл. Подумайте, что если в предыдущем примере нам надо было бы поместить не три буквы, а шесть?

Легко! Можно использовать два самых младших бита, хотя это приведёт к тому, что в нашем изображении будет больше искажений. Также можно было бы поместить сообщение в заголовок файла или после финальной отметки... Всё зависит от того, какое изображение используется, есть много способов вставки сообщений, и принцип, использованный в примере выше можно использовать для многих других форматов файлов.

**[ Ограничения стеганографии ака до чего можно дойти? ]**

Самым приятным может показаться, что мы можем скрывать информацию, когда общаемся с кем-нибудь, и нас может радовать мысль, что даже если кто-то следить за нашей машиной, ничего важного он не встретит... Но на самом деле стеганография - это техника, которая ещё мало развита, мы пока находимся в каменном веке стеганографии.

В действительности возможность эффективного использования стеганографии зависит от размера сообщения, изображения, палитры цветов, и от того, что на мой взгляд является самым главным, от человеческого фактора: мало смысла скрывать что-либо где-то, если это где-то само по себе вызывает подозрения. Например, не стоит класть фотографию с пейзажем посреди директории, в которой находятся файлы совершенно другого типа. Файл, скрывающий сообщение, не должен привлекать к себе внимания.

Как это уже упоминалось ранее, сравнение различных стеганографических приложений не является целью данной статьи. По моему личному мнению все стеганографические приложения страдают значительными ограничениями, многие работают только с некоторыми типами файлами

или очень требовательны к типу изображения, не говоря о том, что практически все только с изображениями и работают... Наконец в стеганографии не существует программы, подобной PGP в криптологии, хотя во всех этих приложений содержится большой спектр возможностей, поэтому необходимо иметь кучу таких программ, чтобы иметь выбор в том как и где скрыть ваше сообщение.

Также необходимо принимать в расчёт, что многие стеганографические алгоритмы не "переживают" модификацию файла, содержащего сообщение.

**[ Приложения в реальной жизни aka когда и где они используются? ]** Вы уже могли подумать, что всё вышеизложенное парни из SET увидели в каком-то фильме, не имеющего ничего общего с реальностью. Нет, стеганография используется в реальной жизни с более или менее положительными успехами. Далее мы рассмотрим некоторые области применения: водяные знаки, передача террористических сообщений, обнаружение авторских прав и некоторые другие.
**Водяные знаки**

Как известно, корпорации вроде RIAA или SGAE (в Испании) употребляют слово "стеганография" как синоним "водяных знаков".

В реальности водяные знаки и стеганография - это не совсем одинаковые вещи, к тому же намерения пользователей стеганографии и пользователей водяных знаков также совпадают далеко не всегда.

Правообладатели включают водяные знаки во все свои файлы, чтобы спустить в дальнейшем своих "пауков" (специальные программы) с привязи, дабы те искали mp3 с этим водяными знаками. Когда они находят такие, то держателю ресурса, где были обнаружены пиратские mp3, приходят сообщения, подписанные адвокатами правообладателей.

Некоторое время назад прочитал (верю, что на www.barrapunto.com), что RIAA утверждает, что владеет технологией новых водяных марок, позволяющих использовать файл-носитель только ограниченное количество раз, после чего он становится нерабочим. Лично мне кажется, что подобные меры должны идти рука об руку с такими: запретить RIAA употреблять алкоголь, иначе она получит овердоз... какая ещё водяная марка, не дающая использовать файл? сколько литров алкоголя нужно выпить, чтобы ляпнуть подобную глупость?

Я ходил на уроки информатики и помню, что согласно теории mp3 подаётся на вход программе и может быть на её выходе и сам по себе не может напрямую влиять на ход выполнения программы. Поэтому, чтобы воплотить то, о чём говорит RIAA, нужна инфраструктура, которую легко спроектировать в теории, но нелегко ввести в употребление.

• Нужна программа, которая бы считывала бы количество раз запусков mp3 и прибавляла бы к этому счётчику единицу каждый раз, когда кто-то запускает mp3.
• Когда количество запусков достигает максимума, необходимо уничтожать архив.

Можно предположить следующие: меняем открытый формат mp3 на формат, в котором используется шифрование, где базой для последнего является количетство запусков... хотя, конечно, для этого нужна монополия на программы, проигрывающие mp3... Если такое случится, то пираты будут просто грабить на лету то, что должно будет идти на колонки, но это уже другая история...

**Изначальное предназначение: спрятанные сообщения**

Общеизвестно, что вокруг террористического акта 11 сентября витает множество мифов, одним из которых является система, использовавшаяся Бин Ладеном и его друзьями для общения друг с другом, чтобы обойти Эшелон и подобные ему системы слежения. Думаю, что знаю ответ: стеганография.

Если слухи, которые ходят по Интернету, верны, то способ, которым использовалась стеганография Аль-Каедой, наводит на мысль, что в этой организации далеко не все являются новичками в компьютерах, увидевшими его в первый раз в жизни и до этого всю жизнь читавшими Коран.

Несомненно, они умеют обращаться с техникой. Например, каким образом могли передаваться сообщения?

Был необходим сайт, через который можно было бы передавать достаточное количество сообщений, но вместе с тем, он не должен был быть подозрительным. Например, человек Аль-Каеды мог работать администратором на каком-нибудь сайте о распродажах одежды из вторых рук, сайтов вроде ebay (по крайней мере, о нём ходят такие слухи) или segundoamano.com.

**Обнаружение авторских прав**

Это очень похоже на то, что мы уже рассматривали - на водяные знаки. Многие софтверные фирмы (не буду перечислять по именам... ладно, например Микрософт) прячут подобным образом номер лицензии, а также персональные данные человека, пользующегося программой.

Это применение настолько очевидно, что я не буду слишком много об этом говорить, понятно, что индустрия может и должна себя защитить, хотя стоит также напомнить, что это абсолютное нарушение частной жизни, когда данные об абсолютно невинном пользователе прячутся таким образом, чтобы их мог прочитать какой-то другой человек.

Существую также другие применения, но они встречаются гораздо реже и не будут упомянуты в этой статье.

**[ Ручная стеганография ]**

Я уже говорил, что с долей скептицизма отношусь к современным стеганографическим приложением, так как их возможности весьма ограниченны.

Но без сомнения, я верю в другой тип стеганографии, который может назвать "ручной стеганографией", стеганографией, которая делается своими руками. Существует множество систем скрывания информацияю, и не обязательно прятать в изображениях или музыке, вы можете спрятать сообщения или шифр в других документах, таких как .exe, .xls, .txt и многих других, главное знать формат файла. Чтобы эффективно использовать эту технику, нужно с воображением подходить к делу и использовать наименее подозрительные места.

Также думаю, что не хватает гения, который добьется того, чтобы можно было включать файлы-носители друг в друга.

Одной формой стеганографии, хотя и довольно тупой является интерпретация слов. Вот пример.

"Я тебя хочу, ты меня ненавидишь, ты меня хочешь, я тебя ненавижу. На самом деле ты не понимаешь, что я чувствую, и думаю, что я чувствую боль из-за того, что чувствуешь ты".

Как видите, абсолютно нормальный текст - любовное послание. Сейчас извлечём из текста все "я" и "ты":

```
я ты ты я ты я я ты
```

И если мы будем считать "я" равным 1, а "ты" равным 0, то у нас получится:

```
1 0 0 1 0 1 1 0 0 1
```

Это частично зашифрованное сообщение. Конечно, если письмо будет написано точно так же, как это сделано выше, то перехвативший его человек может подумать две вещи: не являются ли отправитель сообщения идиотом, либо не скрывается ли здесь что-то серьёзное. Поэтому текст должен быть гораздо больше и более сложным по контексту, чтобы не привлекать внимания.

Заканчивая с данным разделом хочу отметить, что если кто-то думаешь, что стеганография - это сравнительно недавнее изобретение, то он заблуждается. Она использовалась древними греками, в средние века, а также немцами во время Второй мировой войны.

**[ Хорошие способы попрактиковаться в использовании стеганографии ]**

Если кто-то решил спрятать пару-другую своих файлов, вот несколько советов, которые вам следует принять во внимание:

• Убедитесь, что ваш файл-новитель соответствует контексту. Например, изображение должно находиться там, где оно не вызовет никаких подозрений, скажем в директории с другими картинками.
• Электронная стеганография во многом предназначена для того, чтобы прятать ваши сообщения от взглядов тех, кому оно не предназначается. Учтите, вы рискуете тем, что вас примут за подразделение Аль-Каеды :-).
• Выбирайте правильную стеганографическую систему, избегайте слишком известных программ. Никогда не знает, по каким критериям осуществляется поиск сообщений, никто не знает, какими инструментами пользуются те, кто контролируют Эшелон, и насколько эффективно они это делают, никто не знает, какое программное обеспечение используется для шпионских сетей.
• Помните, что в случае, если это окажется нелегальным (хотя во многих странах не существует каких-либо законодательных ограничений относительно стеганографии), вы всегда сможете сослаться на случайность и незнание :-).

**[ Некоторые ссылки ]**

Для тех, кому захотелось продолжить исследования в данной области, я привожу несколько ссылок по этой теме, хотя я уже предупредил, что информация очень разбросана и почти вся на английском.

• http://www.jjtc.com/Steganography/ --> довольно хороший сайт.
• http://www.StegoArchive.com --> довольно полный сайт.
• http://steganography.com/ --> для похода за покупками.
• http://www.privacyexposed.com/resources/steganog.htm --> много ссылок.

Также можно попробовать воспользоваться ответами, которые даёт по этой теме Google.

# Циклический инкремент пароля

**Автор:** PolASoft

**Дата:** 2004-05-27

Автор: (c) PolASoft

В данной статье мы рассмотрим один аспект перебора паролей, на котором обычно не заостряют внимание при написании программ-брутфорсеров (программ для подбора паролей), а именно - непосредственно сам инкремент пароля, т.е. формирование следующего пароля для его анализа.

Конечно, если проверка пароля (т.е. "тело" программы перебора) выполняется тысячи, десятки тысяч или же миллионы тактов процессора, то можно использовать любой способ инкремента пароля - заметного изменения скорости не будет. Но когда на проверку пароля уходит всего 150...300 тактов процессора и нам нужно быстро сформировать новый пароль, то, конечно же, вопрос оптимального инкремента пароля становится очень и очень актуальным.

Сразу оговорюсь, что рассматриваться будет именно вариант полного перебора всех паролей, без семантического анализа рядом стоящих букв, не используя перебор по маске, гибридный перебор и т.д. Т.е., мы собираемся последовательно перебирать комбинации "A", "B", ... "Z", затем - "AA" и т.д.

И еще одно условие - изменяемым символом в наших комбинациях будет первый символ пароля, а не последний, т.е. комбинации будут вида:

```
AAA
BAA
CAA
...
```

а не

```
AAA
AAB
AAC
...
```

Это существенно упрощает код инкремента пароля, да и, в принципе, нет особой разницы в том - слева или справа менять символы. Нам ведь главное - сделать это максимально быстро, не так ли? А, желательно, еще и компактно. :)

Способов инкремента пароля существует немало. Самым распространенным является "индексный" метод инкремента, когда пароль представлен в виде индексов используемых символов. Т.е. при алфавите "ABCDEFGHIJKLMNOPQRSTUVWXYZ" пароль ADMIN будет представлять собой массив значений {1, 4, 13, 9, 14, 0}, где ноль - конец пароля. Или же массив вида {0, 3, 12, 8, 13}, если для хранения длины пароля используется дополнительная переменная.

Данный метод, конечно же, является неплохим, но автор предлагает рассмотреть другой метод, называемый **"циклическим"** (и ниже мы рассмотрим, почему он так назван).

Этот метод, помимо того, что является крайне быстрым, является еще и очень компактным инкрементом, что с успехом позволяет использовать его как в крупных проектах, так и миниатюрных программках, где необходим перебор паролей. Также в этом варианте "накладные расходы" на изменение длины пароля и изменение некрайнего символа также минимальны.

Данный метод с успехом применяется во всех программах перебора паролей на сайте www.InsidePro.com.

Примеры кода будут даны в синтаксисе языка Ассемблер, встроенного в Visual C++ 6.0 и, соответственно, самого языка C++ 6.0.

Итак, что мы имеем:

```
static char szPassword[256]; // Буфер для хранения текущего пароля
static char szAlphabet[256]; // Алфавит, т.е. набор символов для перебора
...
strcpy(szAlphabet, "ABC");   // Для примера возьмем короткий алфавит "ABC"
ZeroMemory(szPassword, sizeof(szPassword)); // Начинаем перебирать с пустого пароля
...
```

Нам нужно:

• получить символ "A";
• записать его в ячейку szPassword[0];
• затем получить следующий символ - "B";
• также записать его вместо "A";
• получить символ "C";
• записать его вместо "B";
• определить, что мы дошли до конца алфавита;
• взять заново первый символ "A"
• записать его в ячейку szPassword[0];
• проверить ячейку szPassword[1];
• если там ноль, то изменилась длина пароля, поэтому пишем туда символ "A";
• если там НЕ ноль, то инкремент этого символа;
• и так далее...

Идея нашего инкремента следующая - попробуем-ка мы получать следующий символ ("B", к примеру) автоматически на основе того, что мы знаем предыдущий символ - "A".

Для этого нам нужно, чтобы символ "A" был как-то связан с символов "B". Может, в виде таблицы? Или же ... да-да, нам нужна именно таблица, в которой по адресу 65 (код символа "A") будет расположен сам символ "B"!

Поэтому строим таблицу (массив bAlphabet) размером 256 байт следующего вида:

```
0-й байт = код символа "A", т.е. 65
65-й байт = код символа "B", т.е. 66
66-й байт = код символа "C", т.е. 67
67-й байт = ноль.
(все остальные байты не важны, т.к. к ним обращений не будет)
```

Тогда перебор символов (в развернутом виде) будет происходить следующим образом:

```
mov ebx,offset bAlphabet     // Адрес нашей таблицы
xor eax,eax                  // В AL - ноль
movzx eax,byte ptr [ebx+eax] // В AL - символ "A"
movzx eax,byte ptr [ebx+eax] // В AL - символ "B"
movzx eax,byte ptr [ebx+eax] // В AL - символ "C"
movzx eax,byte ptr [ebx+eax] // В AL - ноль
movzx eax,byte ptr [ebx+eax] // В AL - снова символ "A"
...                          // И так далее
```

(Примечание: использование команд "xor eax,eax" и "movxz eax,..." вместо "mov al,0" и "mov al,[ebx+eax] позволяет избежать больших штрафов на процессорах Intel, возникающих при чтении 32-битного регистра после модификации его 8-битной части).

Или же, используя однобайтовую команду XLAT, наш пример будет выглядеть так:

```
mov ebx,offset bAlphabet
xor eax,eax
xlat                         // В AL - символ "A"
xlat                         // В AL - символ "B"
xlat                         // В AL - символ "C"
xlat                         // В AL - ноль
xlat                         // В AL - снова символ "A"
...                          // И так далее
```

Таким образом, мы видим, что без лишних команд сравнения (хотя они, конечно, потом понадобятся, но их будет всего две) и лишнего кода мы осуществляем **циклический**"обход" символов нашего алфавита, причем переход от последнего символа к первому происходит **автоматически**!

Код для инициализации нашей таблицы может быть, к примеру, таким:

```
static unsigned char bAlphabet[256];
...
int i = 0, k = 0;
ZeroMemory(bAlphabet, sizeof(bAlphabet));
while (TRUE)
{
        bAlphabet[k] = (unsigned char)szAlphabet[i];
        if (!szAlphabet[i])
                break;
        k = (unsigned char)szAlphabet[i];
        i++;
}
```

Также нам необходимо проверить алфавит, задаваемый пользователем, на предмет одинаковых символов, т.к. при построении такой таблицы с алфавитом "ABCADEF", к примеру, будет происходить следующее - после символа "A" получим символ "B", затем - "C", после него - "A" и вместо ожидаемого "D" мы снова выйдем на символ "B" и т.д. Таким образом, часть алфавита после второго символа "A" будет просто проигнорирована и нам необходимо перед перебором проанализировать алфавит и просто удалить повторные вхождения одинаковых символов.

Теперь же приведем полный код циклического инкремента с подробными комментариями:

```
__asm
{
        L0:
            ...
            // Проверка пароля, т.е. тело программы-брутфорсера
            ...

            mov edi,offset szPassword // Адрес строки с паролем
            mov ebx,offset bAlphabet  // Адрес таблицы
        L1: movzx eax,byte ptr [edi]  // Текущий символ пароля
            cmp al,0                  // Не изменилась ли длина пароля?
            je L4
        L2: xlatb                    // Следующий символ пароля
            cmp al,0                  // Не последний ли символ алфавита?
            je L3
            mov [edi],al              // Новый символ пароля
            ...
            // Инкремент кол-ва обработанных паролей и другой завершающий код
            ...
            jmp L0                    // На начало глобального цикла

        L3: xlatb                    // AL = снова первый символ алфавита
            stosb                     // Его сохранение в [EDI], инкремент EDI
            jmp L1

        L4: inc dword ptr [nLen]     // Увеличение длины пароля
            jmp L2
}
```

Если же нам проверка на изменение длины пароля не нужна (может быть, мы перебираем до нажатия пользователем Ctrl+C, или же до установки какого-либо флажка, или же просто останавливаем наш поток по внешнему воздействию и пр.), то наш код можно упростить:

```
            ...
            mov edi,offset szPassword
            mov ebx,offset bAlphabet
        L1: movzx eax,byte ptr [edi]
            xlat
```

```
            cmp al,0
            je L3
            mov [edi],al
            ...
            // Инкремент кол-ва обработанных паролей и другой завершающий код
            ...
            jmp L0

        L3: xlatb
            stosb
            jmp L1
```

Ну как? ;)

Затрачено каких-то **25 байт** и у нас полноценный перебор паролей из **любых** 8-битных символов (в том числе и служебных 0x01...0x1F) и с **любого** начального пароля (при условии, что он состоит из символов нашего алфавита). Добавляем еще несколько байт и мы можем отслеживать длину текущего пароля!

Теперь посмотрим на время, затраченное на инкремент пароля.

Но сразу оговоримся, что если время исполнения нам более важно, чем объем кода, то сделаем следующую замену на более быстрый код: команды "mov ebx,offset bAlphabet" и "xlat" заменим на команду "movzx eax,byte ptr [bAlphabet+eax]".

Тогда переход от комбинации "A" к "B" будет таким (по командам):

```
            ...
            mov edi,offset szPassword         // Адрес пароля
        L1: movzx eax,byte ptr [edi]          // Текущий символ пароля - "A"
            movzx eax,byte ptr [bAlphabet+eax] // В AL - следующий символ пароля, т.е. "B"
            cmp al,0                          // У нас НЕ равно нулю
            je L3                             // Переход не происходит
            mov [edi],al                      // Сохраняем новый символ "B" вместо "A"
            ...
```

Смотрите - всего 6 команд процессора.
На большинстве процессоров такой код исполнится за ... **4-5 тактов**. :)
Вы можете измерить это время и сами, вставив до и после этого кода команды RDTSC.
А вот почему это происходит:

1. Нет ни одного ветвления, а значит нет и лишних штрафов на ошибочное предсказание ветвления, на повторное заполнение конвейера и т.д.
2. Все приведенные команды - очень простые и большинство процессоров декодируют их в одну микрокоманду, что позволит одновременно декодировать до нескольких таких команд, и, как следствие, очень эффективно их исполнять.

Более того - в реальном коде декодирование (и исполнение) подобных простых команд тесно связано с последующим и предыдущим кодом, что может снизить среднее время выполнения этих 6 команд до **3-4 тактов**.

Итак, 3-4 такта на замену одного пароля другим - совсем даже неплохой результат.

Но, конечно же, можно развивать эту тему и дальше - если основной код перебора (т.е. непосредственной проверки пароля - верный он или нет), написанный на Ассемблере, построить так, что текущий символ пароля (который оставался в регистре AL) **сохранять** на протяжении всего цикла перебора (конечно же, для этого можно использовать и другой регистр), тогда нам не придется его же считывать из памяти заново, что даст совокупную экономию еще в **один такт**!

Теперь поговорим о "накладных расходах", возникающих при переходе на соседний символ и при изменении длины пароля. Это тоже немаловажный вопрос, т.к. если на эти переходы будет затрачено 20-30 тактов (к примеру), то при длине алфавита в 26 символов среднее время инкремента пароля увеличится на 1 такт. А если больше 20-30 тактов?

Здесь нужен более сложный подход, чем при простом переходе от "A" к "B", но все-таки мы дадим несколько рекомендаций для снижения временных затрат.

• Вместо второй команды "xlat" (более компактной) применим команду "movzx eax,byte ptr [bAlphabet+eax]".
• Вместо команды "stosb" (тоже более компактной) применим пару команд "mov [edi],al" и "inc edi".
• Адрес метки L1 должен быть кратен 32 байтам.
• Или же вместо выравнивания метки L1 "развернуть" этот мини-цикл и не делать перехода назад, а использовать "копию" кода, находящегося по адресу L1.
• И т.п.

И в итоге - немного арифметики.

Пусть среднее время цикла проверки одного пароля - 300 тактов процессора.
Тогда средняя скорость перебора на процессоре частотой 1 ГГц составит 3,33 миллиона паролей/сек.
При уменьшении времени цикла на 10 тактов (к примеру, за счет наших ухищрений) мы получим 290 тактов и среднюю скорость около 3,45 миллионов п/с.
Разница: **~110 тысяч паролей в секунду**!

Видите - снизив время, затраченное на работу с одним паролем, всего лишь на несколько тактов, ваша программа сможет перебирать быстрее на сотню (или даже сотни) тысяч паролей в секунду!

И еще один совет, касающийся подсчета количества обработанных паролей.

Да-да, читатель может сказать, что тут-то и так все ясно. Конечно, ясно, но и здесь можно выиграть пару-тройку тактов (если мы стремимся максимально оптимизировать процесс перебора).

Для накопления количества перебранных паролей обычно используется две переменных типа DWORD, т.к. при больших скоростях (миллионы паролей/сек) 4 миллиарда паролей (размер типа данных DWORD) будут обработаны за несколько десятков минут, что, конечно же, явно недостаточно.

Итак, обычно счетчик паролей увеличивают так:

```
DWORD dwLow, dwHigh;
...
dwLow++;
if (dwLow == 0)
        dwHigh++;
```

Или, на Ассемблере:

```
        inc dword ptr [dwLow]
        jnz L1
        inc dword ptr [dwHigh]
L1:     ...
```

Что мы видим - 3 команды, из которых одна - команда проверки, причем практически всегда результат проверки будет TRUE и процессору придется "перепрыгивать" на команду по адресу L1, что влечет за собой штрафы и перезаполнение конвейера.

Либо же сделать проверку не JNZ, а JZ, перенести по адресу L1 команду "inc dword ptr [dwHigh]" и добавить еще одну команду - JMP обратно...

Но можно сделать такой трюк, используя следующую пару команд:

```
        ...
        add dword ptr [dwLow],1
        adc dword ptr [dwHigh],0
        ...
```

В первой команде мы инкрементируем младший DWORD счетчика, а во второй - увеличиваем старший DWORD на ... ноль? Да, на ноль, но только если не было переполнения младшего DWORD'а счетчика. Если же оно было, то включится флаг переноса "C" и команда ADC прибавит в значению dwHigh ноль и значение этого флага, т.е. единицу. Что нам и нужно.

И декодируется/выполняется наша пара команд за ... **1 такт**. :)

Кстати, хитрые парни из Microsoft тоже не зря получают свою зарплату. :)
Если в Visual C++ 6.0 написать следующий код:

```
...
unsigned __int64 qwCount; // Сразу используем переменную типа QWORD
...
qwCount++;
...
```

и сделать компиляцию с оптимизацией (не забудьте включить создание ASM-листинга), то вы можете увидеть, что команда "qwCount++;" откомпилируется в ту же пару команд ADD/ADC. Причем, это явно не "могучий интеллект" компилятора, а одна из "изюминок" оптимизации, заранее заложенных в компилятор его разработчиками.

Автор будет рад, если приведенные выше примеры позволят хоть на чуть-чуть, но увеличить скорость работы ваших программ, а также, если у вас получится и дальше развить показанные выше идеи оптимизации перебора паролей.

Удачи всем!

**Листинг 1.** Программа генерации паролей.
Для ее использования откройте пустой проект (Win32 Console Application) в пакете Visual Studio 6.0, создайте новый CPP-файл и скопируйте в него следующий код:
(Примечание: если вы проинициализируете строку szPassword своим паролем, состоящим из символов алфавита szAlphabet, то программа начнет инкрементировать пароль именно с него).

```
#include "stdio.h"
#include "windows.h"

int main(int argc, char* argv[])
{
        static char szPassword[256];
        ZeroMemory(szPassword, sizeof(szPassword));

        static char szAlphabet[256];
        strcpy(szAlphabet, "ABC");

        static unsigned char bAlphabet[256];
        ZeroMemory(bAlphabet, sizeof(bAlphabet));

        int i = 0, k = 0;
        while (TRUE)
        {
                bAlphabet[k] = (unsigned char)szAlphabet[i];
                if (!szAlphabet[i])
                        break;
                k = (unsigned char)szAlphabet[i];
                i++;
        }

        while (TRUE)
        {
                __asm
                {
                    pushad
                    mov edi,offset szPassword
                    mov ebx,offset bAlphabet
                L1: movzx eax,byte ptr [edi]
                    xlat
```

```
                cmp al,0
                je L3
                mov [edi],al
                jmp L5

            L3: xlat
                stosb
                jmp L1

            L5: popad
            }
            printf("%s\n", szPassword);
        }
        return 0;
}
```

# Алгоритм шифрования ГОСТ 28147-89. Метод простой замены.

**Автор:** Evil`s Interrupt.

**Дата:** 2004-10-17

_«Пока ты жив, не умирай, на этот мир взгляни.
У многих здесь душа мертва – они мертвы внутри.
Но ходят и смеются, не зная, что их нет,
Не торопи свой смертный час» – она сказала мне.

Ария, «Там высоко»_

**Содержание:**



**Введение.**

Данный документ является моей попыткой описать метод простой замены алгоритма шифрования ГОСТ 28147-89 наиболее простым, но, тем не менее, технически-грамотным языком. О том, насколько получилось ли это у меня, читатель скажет свое мнение, после того как прочтет первые шесть пунктов.

Для того, что бы мой труд дал больше пользы рекомендую вооружиться трудами авторов указанных в списке используемой литературы. Рекомендуется также калькулятор, чтобы в нем были функция по расчету операции **XOR** , т.к. прочтение статьи предполагает, что читающий вознамерился изучить данный алгоритм шифрования. Хотя в качестве справочного пособия она тоже подойдет, но я писал эту статью именно, как обучающую.

**Предварительные сведения о блочных шифрах.**

Прежде чем мы начнем рассматривать алгоритм, нам необходимо ознакомиться с историей создания такого рода шифров. Алгоритм относится к разряду блочных шифров, в архитектуре которых информация разбивается на конечное количество блоков, конечный естественно может быть не полным. Процесс шифрования происходит именно над полными блоками, которые и образуют шифрограмму. Конечный блок, если он неполный дополняется чем либо (о нюансах по его дополнению я скажу ниже) и шифруется так же как и полные блоки. Под шифрограммой я

понимаю – результат действия функции шифрования над некоторым количеством данных, которые пользователь подал для шифрования. Другими словами шифрограмма – это конечный результат шифрования.

История развития блочных шифров ассоциируется с началом 70х годов, когда компания IBM осознав необходимость защиты информации при передаче данных по каналам связи ЭВМ, приступила к выполнению собственной программы научных исследований, посвященных защите информации в электронных сетях, в том числе и криптографии.

Группу исследователей – разработчиков фирмы IBM, приступившей к исследованию систем шифрования с симметричной схемой использования ключей, возглавил доктор **Хорст Файстель**.

**2.1 Сети Файстеля**

Предложенная Файстелем архитектура нового метода шифрования в классической литературе получила название «Архитектура Файстеля», но на данный момент в русской и зарубежной литературе используется более устоявшийся термин – "сеть Файстеля" или Feistel`s NetWork. В последствии по данной архитектуре был построен шифр «Люцифер» - который позднее был опубликован и вызвал новую волну интереса к криптографии в целом.

Идея архитектуры "сети Файстеля" заключается в следующем: входной поток информации разбивается на блоки размером в n битов, где n четное число. Каждый блок делится на две части – L и R, далее эти части подаются в итеративный блочный шифр, в котором результат j-го этапа определяется результатом предыдущего этапа j-1! Сказанное можно проиллюстрировать на примере:

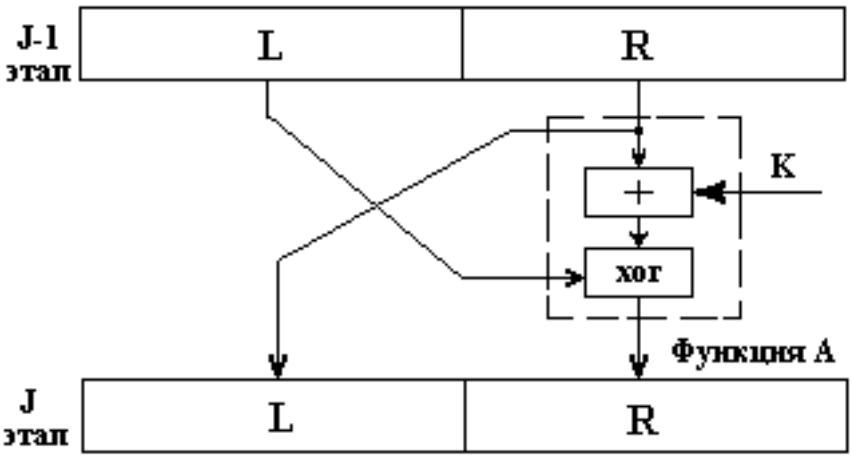



**Рис. 1**

Где, функция A – это основное действие блочного шифра. Может быть простым действием, таким как операция XOR, а может иметь более сложный вид быть последовательностью ряда простых действий – сложение по модулю, сдвиг влево, замена элементов и т.д., в совокупности эти простые действия образуют так называемый – основной шаг криптопреобразования.

Следует заметить, что ключевыми элементами работы функции является подача элементов ключей и операция XOR и от того насколько хорошо продуманы работа этих операций, говорит о криптостойкости шифра в целом.

Для того чтобы идея сетей Файстеля была окончательна ясна, рассмотрим простейший случай изображенный на **рис. 1** , где в функции A – выступит операции “mod 2” (“xor”), но это **простейший** случай, в более серьезной ситуации, например сокрытие информации государственной важности функция A может быть более сложной (сколько я видел функция A действительно бывает очень сложной):

Исходные данные:

L = 1110b, R = 0101, K = 1111b

Цель:

Получить шифрограмму

Решение:

1. (R + K) mod 24 = Smod, Smod = 0100b

2. (Smod + L) mod 2 = Sxor, Sxor = 1010b

3. L = R, R = Sxor

Итог:

L = 0101b, R = 1010b

Поясним наши действия:

1. Эта операция сложение по mod 24. На практике такая операция сводится к простому сложению, где мы должны сложить два числа и проигнорировать перенос в 5й разряд. Так как, если проставить над разрядами двоичного представления числа проставить показатели степени, над пятым разрядом как раз будет показатель четыре, взглянем на рисунок ниже, где изображены действия нашей операции:


**Рис. 2**

Здесь я стрелкой указал на показатели степени, как видно, результат должен был получиться 10100, но так как при операции mod 24 игнорируется перенос, мы получаем 0100.

2. Эта операция в литературе называется mod 2, на языке ассемблера реализуется командой **XOR**. Но ее более правильное название mod 21. Без этой уникальной операции вряд ли можно построить быстрый, легко реализуемый алгоритм шифрования и при этом, чтобы он был еще довольно криптостойким. Уникальность этой операции заключается в том, что она сама себе обратная! К примеру, если число А поXORить с числом Б, в результате получим В, в дальнейшем достаточно переXORить числа Б и В между собой, чтобы получить прежнее значение А!

В этой операции мы получили 1010 имея числа 1110 и 0100, чтобы получить обратно 1110, достаточно переXORрить между собой числа 0100 и 1010! Более подробно об этой операции можно почитать в статье, которая вложена на сайте **www.wasm.ru** , «**Элементарное руководство по CRC_алгоритмам обнаружения ошибок** » автор, которой **Ross N. Williams**. В этом труде есть пункт - «**5. Двоичная арифметика без учета переносов** ». Вот именно в этой статье и описана операция **xor!** Я восклицаю потому что в этой статье эта операция так расписана, что читатель не просто понимает как работает эта операция, он даже начинает ее **видеть, слышать и чувствовать!**

3. Это действие необходимо, чтобы при расшифровывании из шифрограммы можно было получить исходные значения.

### 2.2 Блочный шифр ГОСТ 28147-89

Алгоритм шифрования ГОСТ 28147 – 89 относится к разряду блочных шифров работающих по архитектуре сбалансированных сетей Файстеля, где две части выбранного блока информации имеют равный размер. Алгоритм был разработан в недрах восьмого отдела КГБ преобразованного ныне в ФАПСИ и был закреплен, как стандарт шифрования Российской Федерации еще в 1989 году при СССР.

Для работы данного метода алгоритма необходимо разбить информацию на блоки размером в 64 бита. Сгенерировать или ввести в систему шифрования, следующую ключевую информацию: ключ и таблицу замен. К выбору ключа и таблицы замен при шифровании следует отнестись очень серьезно, т.к. именно это фундамент безопасности вашей информации. О том, какие требования налагаются на ключ, и таблицу замен смотри пункт «Требования к ключевой информации».

При рассмотрении метода мы не будем заострять на этом внимания, т.к. эта статья, как я уже говорил выше, написана с целью, научить читающего, шифровать данные по методу простой замены данного алгоритма шифрования, но мы обязательно коснемся этого вопроса в конце статьи.

Теоретический минимум.

### 3.1 Ключевая информация

Как я уже говорил выше, в шифровании данных активное участие принимают:

3.1.1. Ключ – это последовательность восьми элементов размером в 32 бита каждый. Далее будем обозначать символом К, а элементы из которых он состоит – k1,k2,k3,k4,k5,k6,k7,k8.

3.1.2 Таблица замен – матрица из восьми строк и шестнадцати столбцов, в дальнейшем – Hij. Каждый элемент на пересечении строки i и столбца j занимает 4 бита.

---

### 3.2 Основной шаг криптопреобразования

Основным действием в процессе шифрования является – основной шаг криптопреобразования. Это ничто иное, как действие по шифрованию данных по определенному алгоритму, только название разработчики ввели уж больно громоздкое :).

Прежде чем начать шифровать, блок разбивают на две части L и R, по 32 бита каждая. Выбирают элемент ключа и только потом подают эти две части блока, элемент ключа таблицу замен в функцию основного шага, результат основного шага это одна итерация базового цикла, о котором речь пойдет в следующем пункте. Основной шаг состоит из следующих действий:

1. Сложение часть блока R суммируется с элементом ключа K по mod 232. О подобной операции я описал выше, здесь тоже самое только показатель степени не «4», а «32» - результат этой операции в дальнейшем буду обозначать Smod.
2. Полученный ранее результат Smod делим на четырех битные элементы s7,s6,s5,s4,s3,s2,s1,s0 и подаем в функцию замены. Замена происходит следующим образом: выбирается элемент Smod - si, с начала начинаем с младшего элемента, и заменяем значением из таблицы замен по i - той строке и столбцу, на который указывает значение элемента si. Переходим к si+1 элементу и поступаем аналогичным образом и продолжаем так, пока не заменим значение последнего элемента Smod – результат этой операции будем обозначать как, Ssimple.
3. В этой операции значение Ssimple сдвигаем циклически влево на 11 бит и получаем Srol.
4. Выбираем вторую часть блока L и складываем по mod 2 с Srol, в итоге имеем Sxor.
5. На этой стадии часть блока L становится равным значению части R, а часть R в свою очередь инициализируется результатом Sxor и на этом функция основного шага завершена!

### 3.3 Базовые циклы: “32-З”, “32-Р”.

Для того чтобы зашифровать информацию надо разбить ее на блоки размером в 64 бита, естественно последний блок может быть меньше 64 битов. Этот факт является ахиллесовой пятой данного метода «простая замена». Так как его дополнение до 64 бит является очень важной задачей по увеличению криптостойкости шифрограммы и к этому чувствительному месту, если оно присутствует в массиве информации, а его может и не быть (к примеру, файл размером в 512 байт!), следует отнестись с большой ответственностью!

После того как вы разбили информацию на блоки, следует разбить ключ на элементы:

K = k1,k2,k3,k4,k5,k6,k7,k8

Само шифрование заключается в использовании, так называемых – базовых циклов. Которые в свою очередь включают в себя n – ое количество основных шагов криптопреобразования.

Базовые циклы имеют, как бы это сказать, маркировку: n – m. Где n – количество основных шагов криптопреобразования в базовом цикле, а m – это «тип» базового цикла, т.е. о чем идет речь, о «З» ашифровывании или «Р» асшифровывании данных.

Базовый цикл шифрования 32–З состоит из 32-х основных шагов криптопреобразования. В функцию реализующую действия шага подают блок N и элемент ключа К причем, первый шаг происходит с к1, второй над полученным результатом с элементом к2 и т.д. по следующей схеме:

k1,k2,k3,k4,k5,k6,k7,k8,k1,k2,k3,k4,k5,k6,k7,k8,k1,k2,k3,k4,k5,k6,k7,k8k8,k7,k6,k5,k4,k3,k2,k1

Процесс расшифровывания 32–Р происходит аналогичным образом, но элементы ключа подаются в обратной последовательности:

k1,k2,k3,k4,k5,k6,k7,k8,k8,k7,k6,k5,k4,k3,k2,k1,k8,k7,k6,k5,k4,k3,k2,k1,k8,k7,k6,k5,k4,k3,k2,k1

Практика.

**4.1 Реализация основного шага криптопреобразования**

После того как мы познакомились с теорией о том, как шифровать информацию настало посмотреть, как же происходит шифрование на практике.

Исходные данные:

Возьмем блок информации N = 0102030405060708h, здесь части L и R равны:

L = 01020304h, R =05060708h, возьмем ключ:

K = ‘**as28** zw37 **q839** 7342 **ui23** 8e2t **wqm2** ewp1’ (это ASCII – коды, для того, чтобы посмотреть шестнадцатеричное представление, можно открыть этот файл в режим просмотра в Total Commander нажав на клавишу «**F3** » и далее клавишу «**3** »). В этом ключе значения элементов будут:

k1 = ‘as28’, k2 = ‘zw37’, k3 = ‘q839’, k4 = ‘7342’

k5 = ‘ui23’, k6 = ‘8e2t’, k7 = ‘wqm2’, k8 = ‘ewp1’

Также возьмем следующую таблицу замен:

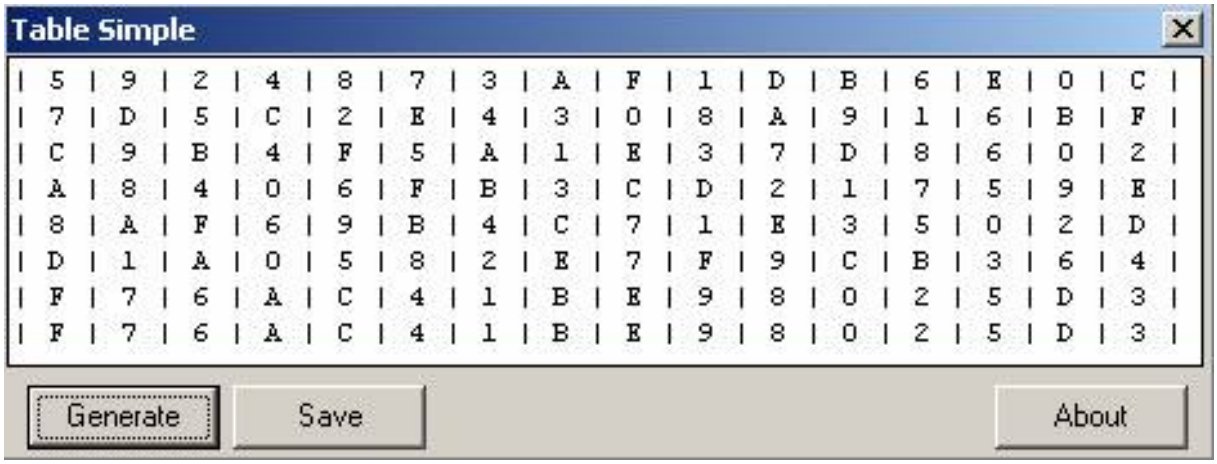



**Рис. 3**

Здесь строки нумеруются от 0 до 7, столбцы от 0 до F.

**Предупреждение:** Вся информация, в том числе и ключ с таблицей замен взята в качестве примера для рассмотрения алгоритма!

Задача:

Используя «Исходные данные», необходимо получить результат действия основного шага криптопреобразования.

Решение:

1. Выбираем часть R = 05060708h и элемент ключа k1 = ‘as28’, в шестнадцатеричном виде элемент ключа будет выглядеть так: 61733238h. Теперь же делаем операцию суммирования по mod 232:


**Рис. 4**

Как видно на рисунке у нас не произошло переноса в 33 бит помеченный красным цветом и с показателем степени «**32** ». А если бы у нас были бы другие значения R и элемента ключа – это вполне могло бы произойти, и тогда бы мы его проигнорировали, и в дальнейшем использовали только биты, помеченные желтым цветом.

Такую операцию я выполняю командой ассемблера **add** :

; eax = R, ebx = ‘as28’

add eax,ebx

; eax = Smod

Результат этой операции Smod = 66793940h

2. Теперь самая заковыристая операция, но если присмотреться по внимательней, то она уже не такая страшная, как кажется в первое время. Представим Smod в следующем виде:

6 6 7 9 3 9 4 0
i -> 7 6 5 4 3 2 1 0

**Рис. 5**

Я постарался наглядно представить элементы Smod на рисунке, но все равно поясню:

s0 = 0, s1 = 4, s2 = 9 и т.д.

Теперь начиная с младшего элемента s0, производим замену. Вспоминая пункт «**3.2 Основной шаг криптопреобразования** » i – строка, si – столбец, ищем в нулевой строке и нулевом столбце значение:

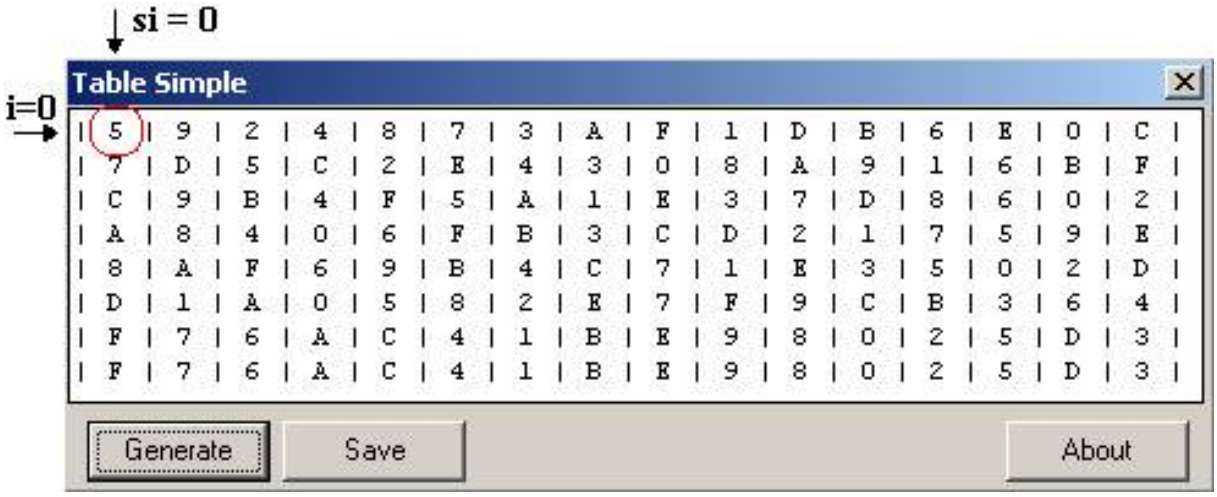



**Рис.6**

Таким образом, текущее значение Smod, не 6679394 **0** h, а 6679394 **5** h.

Приступаем заменять s1, т.е. четверку. Используя первую строку и четвертый столбец (s1= 4!). Глядим на рисунок:

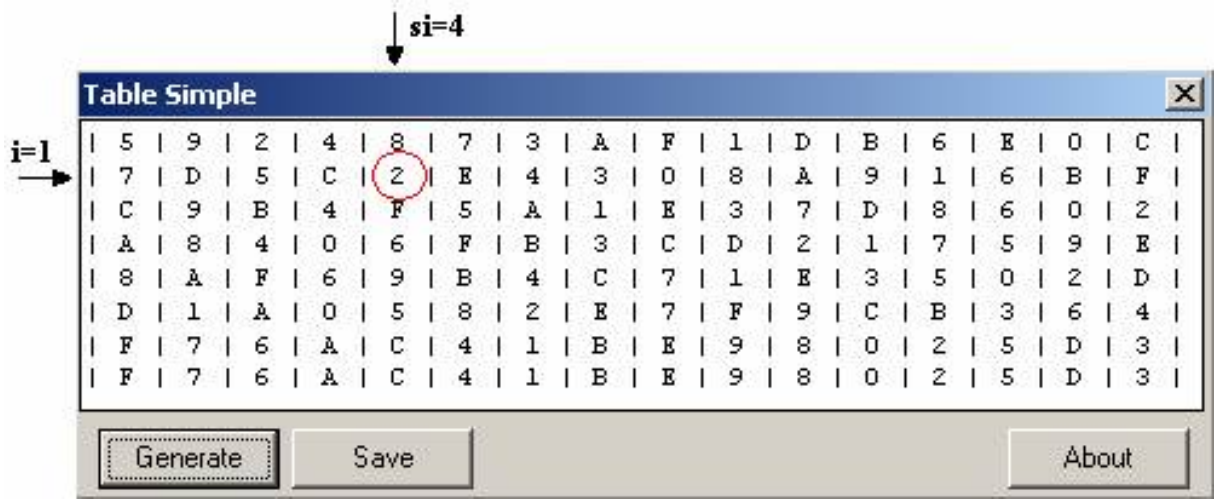



**Рис. 7**

Теперь уже значение Smod, не 667939 **4** 5h, 667939 **2** 5h. Я предполагаю, что теперь алгоритм замены читателю понятен, и я могу сказать, что после конечный результат Ssimple будет иметь следующее значение – 11e10325h.

О том, как это проще всего реализовать в виде команд ассемблера я расскажу позже в следующем пункте, после того, как расскажу о расширенной таблице.

2. Полученное значение Ssimple мы должны сдвинуть на 11 бит влево.

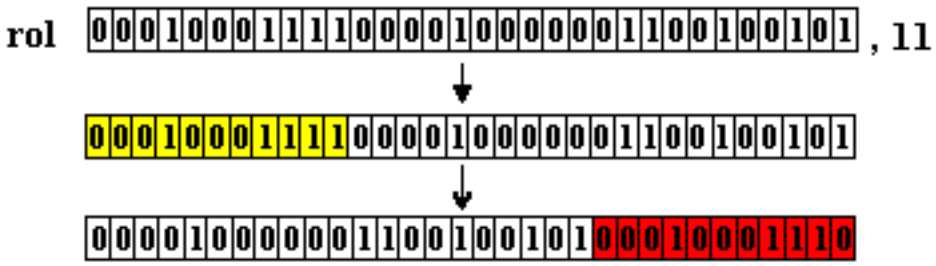



**Рис. 8**

Как видно это действие довольно простое, и реализуется одной командой языка ассемблера – **rol** и результат этой операции Srol равен 0819288Fh.

4. Теперь же остается часть L нашего блока информации поXORить со значением Srol. Я беру калькулятор от w2k sp4 и получаю Sxor = 091b2b8bh.

5. Это действие итоговое и мы просто присваиваем, чисти R значение части L, а часть L инициализируем значением Sxor.

Конечный результат:

L = 091b2b8bh, R = 01020304h

**4.2 Увеличения быстродействия алгоритма**

Теперь же поговорим об оптимизации алгоритма по скорости. При процессе реализации, какого либо проекта, приходится учитывать, что программа, которая работает с регистрами чаще, чем с памятью работает наиболее быстрее и здесь это суждение тоже очень важно, т.к. над одним блоком информации целых 32 действия шифрации!

Когда я реализовывал алгоритм шифрования в своей программе, я поступил следующим образом:

1. Выбрал часть блока L в регистр eax, а R в edx.

2. В регистр esi инициализировал адресом расширенного ключа, об этом ниже.

3. В регистр ebx присваивал значение адреса расширенной таблицы замен, об этом тоже ниже

4. Передавал информацию пунктов 1,2, 3 в функцию базового цикла 32 – 3 или 32 – P, в зависимости от ситуации.

Изучая алгоритм шифрования, а, далее реализовывая, его я столкнулся с трудностью подачи элементов ключа в функцию базового цикла. Но я решил проблему с помощью преобразования ключа до расширенного.

Если посмотреть на схему подачи элементов ключа в пункте «**Базовые циклы: “32-3”, “32-P”** », то наш ключ для базового цикла 32 – 3 можно представить в следующем:

K32-3 = ‘as28’,‘zw37’,‘q839’,‘7342’,‘ui23’,‘8e2t’,‘wqm2’,‘ewp1’,

‘as28’,‘zw37’,‘q839’,‘7342’,‘ui23’,‘8e2t’,‘wqm2’,‘ewp1’,

‘as28’,‘zw37’,‘q839’,‘7342’,‘ui23’,‘8e2t’,‘wqm2’,‘ewp1’,

**‘ewp1’,‘wqm2’,‘8e2t’,‘ui23’,‘7342’,‘q839’,‘zw37’,‘as28’**

Т.е. с начала идут k1,k2,k3,k4,k5,k6,k7,k8 - **‘** as28’, ‘**zw37’, ‘** q839’, ‘7342’, ‘**ui23’, ‘8** e2 **t’, ‘** wqm2’, ‘**ewp1’** три раза эта последовательность повторяется. Затем элементы идут в обратном порядке, т.е.: k8,k7,k6,k5,k4,k3,k2,k1 - **‘ewp1’, ‘wqm2’, ‘8e2t’,‘ui23’,‘7342’,‘q839’,‘zw37’,‘as28’**.

Я заранее расположил в массиве элементы в том порядке, как они должны подаваться в 32 – 3. Тем самым я увеличил память, требуемую под ключ, но избавил себя от некоторых процессов мышления, которые мне были не нужны, и увеличил скорость работы алгоритма, за счет

уменьшения времени обращения к памяти! Здесь я описал только ключ для 32 – 3, для цикла 32 – P я поступил аналогично, но используя другую схему подачи элементов, которую я тоже описывал в пункте «**Базовые циклы: “32-3”, “32-P** ».

Настало время описать реализацию работы функции замен, как я обещал выше. Я не мог описать ранее, т.к. это требует ввода нового понятия – расширенная таблица замен. Я не смогу вам объяснить, что это такое. Вместо этого я вам покажу ее, а вы уж сами сформулируйте для себя, что же это такое – расширенная таблица замен?

Итак, для того чтобы разобраться, что такое расширенная таблица замен нам понадобится таблица замен, для примера возьму ту, что изображена на рис. 3.

К примеру, нам потребовалось заменить, число 66793940h. Представлю его в следующем виде:

```
      6 6 7 9 3 9 4 0
i -> 7 6 5 4 3 2 1 0
```

**Рис. 9**

Теперь если взять элементы s1,s0, т.е. младший байт, то результат функции замены будет равен 25h! Почитав статью Андрея Винокурова, которую я привел в пункте «**Список используемой литературу** », вы действительно обнаружите, что если взять две строки можно получить массив, позволяющий быстро находить элементы замены с помощью команды ассемблера **xlat.** Говорят можно и другим способом более быстрым, но Андрей Винокуров потратил на исследование быстрых алгоритмов для реализации ГОСТа около четырех лет! Думаю, не стоит изобретать велосипед, когда он уже есть.

Итак, о массиве:

Возьмем две первые строки нулевую и первую, создадим массив на 256 байт. Теперь наблюдаем одну особенность, что если надо преобразовать 00h, то результат будет 75h (опираемся на рис.3) – кладем это значение в массив на смещение 00h. Берем значение 01h, результат функции замен 79h, кладем его в массив на смещение 01 и так далее до 0FFh, которое нам даст 0FCh, которое мы положим в массив по смещение 0FFh. Вот мы и получили расширенную таблицу замен для первой группы строк: первой и нулевой. Но еще есть три группы: вторая стр.2, стр.3, третья стр.4, стр. 5, четвертая стр.6, стр.7. С этим тремя группами поступаем тем же способом, что и с первой. Результат – расширенная таблица замен!

Теперь можно реализовать алгоритм, который будет производить замену. Для этого берем исходные коды, которые выложил Андрей Винокуров на своей страничке, смотри «**Список используемой литературы** ».

```
lea ebx,extented_table_simple

mov eax,[положить число которое нужно заменить]

REPT 3

xlat

ror eax,8d

add ebx,100h ;переход к двум следующим узлам

ENDM

xlat

sub ebx,300h ; чтобы в дальнейшем ebx показывал на таблицу
```

Теперь еще одна особенность, предыдущими действиями мы не только заменили, но и сдвинули число на 8 бит влево! Нам остается только сдвинуть число еще на 3 бита влево:

rol eax,3h

и мы получаем результат операции rol eax,11!

Больше я ничего не могу добавить по оптимизации, единственное, что могу подчеркнуть то, что я говорил выше – используйте регистры чаще, чем обращение к памяти. Думаю эти слова только для новичков, опытные и без моих слов это прекрасно понимают :).

**Требования к ключевой информации.**

Как сказано в статье Андрея Винокурова ключ выбирают по двум критериям:

- критерий равновероятного распределения битов между значениями 1 и 0. Обычно в качестве критерия равновероятного распределения битов – выступает критерий Пирсона («хи-квадрат»).

Это значит ключом, в принципе может любое число. То есть при формировании очередного бита ключа вероятность его инициализации единицей или нулем 50/50!

Прошу заметить, что ключ из восьми элементов, каждый по 32 бита, таким образом всего в ключе 32*8 = 256 битов и количество возможных ключей 2256! Тебя это не поражает? :)

- критерий серий.

Если мы посмотрим на наш ключ, который я привел в пункте «**4.1 Реализация основного шага криптопреобразования** », то вы заметите, что справедлива следующая запись:

$$\mathbf{k_i} \underset{i \neq j}{\neq} \mathbf{k_j}$$

**Рис. 10**

Одной фразой значение k1 не должно повториться не в k2, не в каком либо другом элементе ключа.

То есть ключ, который мы выбрали в качестве рассмотрения алгоритма шифрования, вполне соответствует двум приведенным выше критериям.

Теперь про выбор таблицы замен:

Теперь же поговорим о том, как правильно выбрать таблицу замен. Основное требование к выбору таблиц замен – это явление «неповторяемости» элементов, каждый из которых размером в 4 бита. Как вы уже видели выше, каждая строка таблицы замен состоит из значений 0h, 1h, 2h, 3h, …, 0fh. Так вот основное требование гласит о том, что в каждой строке есть значения 0h, 1h, 2h, … , 0fh и каждое такое значение в одном экземпляре. К примеру, последовательность:

1 2 3 4 5 6 7 8 9 A B C D E F

Вполне соответствует этому требованию, но все же! Такую последовательность в качестве строки выбирать не рекомендуется. Так как если вы подадите значение на вход функции, которая опирается на такую строку, то на выходе вы получите такое же значение! Не верите? Тогда возьмите число 332DA43Fh и восемь таких строк, в качестве таблицы замен. Проведите операцию замены, и уверяю вас, на выходе вы получите число 332DA43Fh! То есть такое же, что вы подали на вход операции! А это не является признаком хорошего тона при шифровании, да и являлось ли? :)

Это было одно требование, следующий критерий говорит о том, что – каждый бит выходного блока должен быть статистически независим от каждого бита входного блока!

Как это выглядит проще? А вот как, к примеру, мы выбрали из приведенного выше числа элемент s0 = 0Fh, 01111b. Вероятность того, что мы сейчас заменим первый бит единицей или нулем равна

0,5! Вероятность замены второго, третьего и четвертого бита, каждый бит, рассматриваем по отдельности, единицами или нулями тоже равна 0, 5. При выборе s1 = 0Eh, вероятность того, что мы нулевой бит, а это «0», заменим нулем или единицей тоже равна – 0,5! Таким образом, согласно этому критерию между заменой нулевых битов элементов s0, s1 нет никакой закономерности! Да, вы могли заменить единицами, но вы также могли поставить и нули. :)

Для оценки таблицы по этому критерию можно построить таблицу коэффициентов корреляции, рассчитанные по формуле:

$$p_{ij} = 1 - \frac{\sum_k k_i \oplus s(k)_j}{2^{N-1}}$$

где

$0 \le i \le 3$

$0 \le j \le 3$

$N=4$

- если p = 1, то значение бита j на выходе равно значению бита i на входе при любых комбинациях бит на входе;

- если p = -1, то значение бита j на выходе всегда является инверсией входного бита i;

- если p = 0, то выходной бит j с равной вероятностью принимает значения 0 и 1 при любом фиксированном значении входного бита i.

Возьмем пример одной строки:

**D** — B — 4 — 1 — 3 — F — 5 — 9 — 0 — A — E — 7 — 6 — 8 — 2 — C

Разложим на «составляющие»:

Вход

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
| 1101 | 1011 | 0100 | 0001 | 0011 | 1111 | 0101 | 1001 | 0000 | 1010 | 1110 | 0111 | 0110 | 1000 | 0010 | 1100 |
| D | B | 4 | 1 | 3 | F | 5 | 9 | 0 | A | E | 7 | 6 | 8 | 2 | C |

Выход

Рассчитаем один коэффициент по формуле приведенной выше. Чтобы проще было понять, как это делается, поясню более подробно:

- берем 0-й бит 0-ого числа (0) на входе и 0-й бит 0-ого числа на выходе (1) проводим операцию 0 XOR 1 = 1.

- берем 0-й бит 1-ого числа (1) на входе и 0-й бит 1-ого числа на выходе (1) проводим операцию 1 XOR 1 = 0.

- берем 0-й бит 2-ого числа (0) на входе и 0-й бит 2-ого числа на выходе (0) проводим операцию 0 XOR 0 = 0.

- берем 0-й бит 3-ого числа (1) на входе и 0-й бит 3-ого числа на выходе (1) проводим операцию 1 XOR 1 = 0.

…….

Проведя последовательно операции XOR в такой последовательности, подсчитываем количество всех ненулевых значений, получаем значение 6. Отсюда P00 = 1-(6/24-1) = 0,25. Итак, выяснилось, что значение бита 0 на выходе равно значению бита 0 на входе в 4-х случаях из 16-ти;

Итоговая таблица коэффициентов:

Вход

**Выход** — 0 — 1 — 2 — 3

**0** — 0,25 — 0,00 — 0,00 — -0,75

**1** — 0,00 — -0,25 — 0,00 — 0,25

**2** — -0,25 — 0,25 — 0,00 — 0,00

**3** — 0,50 — -0,25 — 0,00 — 0,00

Как видно из таблицы корреляционных коэффициентов бит 3 на входе инвертирован относительно бита 0 на выходе в 14 случаях из 16, что составляет 87.5 % Вот это уже не допустимо для нормальных систем шифрования. Для разнообразия возьмем еще примерчик:

**9** — 8 — 3 — A — C — D — 7 — E — 0 — 1 — B — 2 — 4 — 5 — F — 6

Таблица коэффициентов будет следующая (кому не лениво может пересчитать)

Вход

**Выход** — 0 — 1 — 2 — 3

**0** — -0,25 — 0,00 — 0,00 — 0,00

**1** — 0,00 — 1,00 — 0,00 — 0,00

**2** — 0,00 — 0,00 — 1,00 — 0,00

**3** — 0,00 — 0,00 — 0,00 — -0,50

Ну, в этой таблице дела обстоят еще хуже – биты 1 и 2 группы остаются неизменными! Криптоаналитику есть, где развернуться :) С учетом всех этих требований простым перебором («в лоб») были найдены таблицы перестановки соответствующие указанной теории (на сегодняшний день – 1276 сочетаний) Вот некоторые из них:

09 0D 03 0E-06 02 05 08-0A 07 00 04-0C 01 0F 0B

00 05 0A 07-03 08 0F 0C-0E 0B 04 09-0D 06 01 02

06 0B 0F 00-0C 01 02 0D-08 07 09 04-05 0A 03 0E

04 0E 00 09-0B 01 0F 06-03 0D 07 0A-0C 02 08 05

04 02 08 0E-05 0F 03 09-0B 01 0D 07-0A 0C 06 00

07 03 09 0C-08 00 06 0F-0E 04 01 0A-0D 0B 02 05

06 0F 03 08-0D 04 0A 01-09 02 05 0C-00 0B 0E 07

0C 06 08 01-03 09 07 0E-0B 05 0F 02-04 0A 00 0D

04 0B 09 06-0E 01 00 0F-0A 05 03 0C-0D 02 07 08

00 0E 0F 01-07 08 09 06-04 0B 0A 05-03 0D 0C 02

0F 09 01 07-04 0A 08 06-0E 00 02 0C-05 03 0B 0D

0A 03 04 01-05 0C 0B 0E-08 06 0F 0D-07 09 00 02

0B 06 0F 01-04 0A 08 05-00 0D 0C 02-07 09 03 0E

0C 03 02 08-0D 06 0B 05-07 09 04 0F-0A 00 01 0E

02 0B 0F 04-09 00 06 0D-05 0E 01 08-0C 07 0A 03

**Список использованной литературы.**

1. Статья Андрея Винокурова:

**Алгоритм шифрования ГОСТ 28147-89, его использование и реализация**

**для компьютеров платформы Intel x86.**

(можно найти по адресу: http://www.enlight.ru/crypto/frame.htm).

Тут же и исходные коды, по реализации алгоритма шифрования.

2. Статья Хорста Файстеля:

**Криптография и Компьютерная безопасность.**

(можно найти по тому же адресу что и предыдущую статью)

3. Ross N. Williams:

**Элементарное руководство по CRC алгоритмам обнаружения ошибок**

Выложена на сайте **www.** wasm. **ru**.

**Благодарности.**

Хотелось бы высказать благодарность всем посетителям форума www.wasm.ru. Но особо бы хотелось бы поблагодарить ChS, который в настоящий момент известен, как SteelRat, он помог мне понять такие вещи, чего я бы, наверное, никогда бы не понял, а так же помощь при написании пункта: «**Требования к ключевой информации** », основной часть данного пункта была написана им. Также глубоко признателен сотруднику КГТУ им. А.Н. Туполева Аникину Игорю Вячеславовичу и грех было бы не отметить Криса Касперски, за то, что он есть и Volodya / wasm.ru за его наставления. Ох, и достается мне от него :). Так же хочу отметить Sega-Zero / Callipso зато, что донес до моего разума некоторые математические дебри.

Это, пожалуй, все, что я хотел бы сказать вам.

Буду, признателен за критику или вопросы, связанные с этой статьей или просто советы. Мои контактные данные: int20h@yandex.ru, ICQ – 337310594.

С уважением Evil`s Interrupt.

+ P.S.: Этой статьей я не старался кого-то перещеголять. Она была написана с умыслом, облегчить изучение ГОСТа и если у вас получились трудности, то это не значит, что я повинен в этом. Будь разумны, и наберитесь терпения, всего вам доброго!

# 21. Консоли и КПК

# GBA ASM - День 1: Постигаем ассемблер

**Автор:** Mike H, пер. Aquila

**Дата:** 2003-07-13

Вы можете спросить, будем ли мы использовать тот же ассемблер, который идёт вместе с GCC. Короткий ответ: НЕТ. Развёрнутый ответ: мы будем использовать ассемблер Goldroad, который можно найти на его сайте или скачать здесь (находится в архиве `wasm_files.zip`: `files/recovered/gbaasm01/goldroad.zip`). Я использую версию 1.6F. Вам следует скачать самую последнюю версию ассемблера и распаковать её куда-нибудь, так чтобы в пути не было пробелов. Если в пути нет пробелов - это всегда существенно облегчает жизнь.

Оки, вы распаковали ассемблер туда, куда хотели. У меня есть файл, который может вам изрядно помочь. Это screen.h (находится в архиве `wasm_files.zip`: `files/recovered/gbaasm01/screen.h`) из туториала Pern'a, который я вручную сконвертировал в синтаксис ассемблера Goldroad. Скачайте его здесь и поместите в ту же директорию, где находится Goldroad.exe.

Теперь вы наверняка захотите проверить, как он работает. Но для начала вам нужен эмулятор. Идите и найдите VisualboyAdvance, так как я на 100% уверен, что он понимает командную строку. Чтобы проверить, как он работает, создайте файл first.asm, скопируйте в него нижеследующий исходник и поместите в ту же директорию, где находится ассемблер.

```
; начинайте копировать отсюда

@include screen.h
@textarea

ldr r1,=REG_DISPCNT
ldr r2,=(BG2_ENABLE|MODE_3)
str r2,[r1]
ldr r1,=0x0FF
ldr r2,=vram+2410
str r1,[r2]
label1
B label1

@pool
@endarea

; заканчивайте копировать здесь
```

Альтернативный код на C будет примерно следующим:

```
int main()
{
SetMode(MODE_3|BG2_ENABLE);
vram[2410] = 0x0FF;
while(1)
{}
}
```

Тем не менее, как вы вероятно знаете, скомпилированный C-код будет ОГРОМНЫМ. Только ассемблер может дать вам маленькие бинарники. Давайте проанализируем исходник строка за строкой.

```
@include screen.h
```

Эта строка делает то же самое, что и выражение #include в C.

```
@textarea
```

Эта строка просто определяет, где начинается код.

```
ldr r1,=REG_DISPCNT
```

О, вот и первая настоящее ассемблерное выражение. Здесь в дело вступаю я.

В ARM'е (процессор GBA) есть 16 регистров r0-r15. r15 - это вроде IP (Instruction Pointer) в процессорах Intel. ОСТАВЬТЕ ЕГО В ПОКОЕ! Я стараюсь работать только с первыми 10 - r0-r9. Если вы знаете Intel- или x86-ассемблер, тогда вы ПОЧТИ знаете инструкции MOV и LDR. Эти инструкции позволяют вам перемещать 32-х битные числа. Второй операнд (число/регистр) перемещается в первый (число/регистр). Инструкция MOV используется только в особых случаях. Мы пока будем использовать LDR, которая расшифровывается как LoaD Register (загрузить в регистр). Она загружает в указанный регистр значение другого регистра, либо ячейки памяти, либо заданного числа. Давай вернёмся к нашей строке кода:

```
ldr r1,=REG_DISPCNT

 * Это инструкция LDR.

        * Первым параметром в нашем случае является регистр r1.

        * REG_DISPCNT задан в screen.h и определяет местонахождение памяти
          управления экраном.
```

Используйте звёздочки, чтобы определить к чему относятся комментарии.

```
ldr r2,=(BG2_ENABLE|MODE_3)
```

Здесь то же самое, что и в предыдщущей строке, только на этот раз вторым операндом являются два cORенных значения (оператор '|'). BG2_ENABLE и MODE_3 заданы в screen.h и будучи cORенными дают значение 0x0403. Это значение помещается в регистр r2.

```
str r2,[r1]
```

Вау, наша вторая инструкция, инструкция STR помещает значение первого регистра в ячейку памяти, задаваемую вторым операндом. STR расшифровывается как STore Register (сохранить регистр). В данном случае квадратные скобки [] говорят, что регистр используется как указатель. Знаете что? Эти три инструкции делают то, что в C для нас делает SetMode(MODE_3|BG2_ENABLE). Это переводит экран в битмэп-режим номер три (3).

```
ldr r1,=0x0FF
```

Сейчас вы должны это понять - это загружает в r1 0x0FF (255).

```
ldr r2,=vram+2410
```

Вы должны понять и этом, за исключением того, что я сейчас объясню. Для начала, просто чтобы вы знали, vram также задаётся в screen.h, и это память, в которую вам понадобится писать, чтобы вывести что-нибудь на экран. vram определяется как 0x06000000.

Теперь наша инструкция прибавляет 2410 к значению vram, загружая значение в r2. Почему? Чтобы поместить значение цвета вам нужно задать x и y, верно? Поэтому вот формула для 3-его режима:

```
 y * 240 + x
```

Соответственно наша инструкция задаёт позицию x=10, y=10. Более того, 0x0FF, который мы ранее поместили в r1, будет означать ярко-красный цвет, когда мы поместитм точку на экран.

```
str r1,[r2]
```

Снова инструкция str, только в этом раз мы помещаем содержимое регистра r1 в ячейку памяти, на которую указывает r2. В данном случае r2 указывает на экранную координату 10,10, а r1 - это ярко-красный цвет. Таким образом, сохранение ярко-красного цвета в экранную память в позиции

10,10... Позже я дам вам увидеть результат. Но пока ещё немного кода.

```
label1
```

Задаёт метку, к которой вы можете перейти подобно тому, как работает команда goto в BASIC.

```
b label1
```

Эквивалентно 'Goto label1' в BASIC'e. Эти две строки создают бесконечный цикл. Инструкция B работает подобно инструкции Jxxx в x86-ассемблере, она передаёт управление в зависимости от результата предыдущих операций, подробнее об этом будет написано дальше.

```
@pool
```

Даёт ассемблеру место, куда можно поместить некоторые из чисел, которые мы задавали и использовали. АБСОЛЮТНО НЕОБХОДИМО.

```
@endarea
```

Говорит ассемблеру, что секция кода закончена.

Эй, мы только что закончили с нашей первой программой под GBA на ассемблере!

**Компилирование нашего исходника**

Я предполагаю, что вы работаете в Windows, да и один совет - ПОСТАВЬТЕ XP! (Второй совет от переводчика - СНЕСИТЕ XP!)

Наберите 'goldroad first.asm' в директории, где находится Goldroad Assembler и нажмите ввод.

Если вы увидели какие-нибудь сообщения об ошибках, попробуйте снова, убедитесь, что задали команду верно.

Если вы увидели "Assembled successfully" или что-нибудь вроде этого, поздравляю!

Если всё прошло как надо, то в папке Goldroad'a должен появиться файл first.gba.

Теперь осталось запустить полученный файл в эмуляторе GBA (VisualBoy Advance). На экране должна появиться красная точка где-то в районе 10,10. Поздравляю! Только что закончился День 1.

# GBA ASM - День 2: Немного информации об ассемблере ARM

**Автор:** Mike H, пер. Aquila

**Дата:** 2003-07-14

ARM - это компания, которая делает процессор GBA. Процессоры ARM являются RISC-процессорами (в отличии от процессоров INTEL). RISC расшифровывается как Reduced Instruction Set Computers (CISC - Complex ...). Хотя в этих процессорах не так много инструкций (что хорошо), инструкции ARM (а может и других RISC-процессоров, я не знаю) имеют много различных назначений и комбинаций, что делает RISC-процессорами таким могущественными, каким они являются.

**Регистры**

Я не знаю о других процессорах ARM, но у того, который используется в GBA, 16 регистров и в отличии от процессоров Intel (и других), все регистры можно спокойно использовать (как правило). Регистры следующие:

r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15

Вау! Много! Объясняю по-порядку.

ro: Делайте, что хотите!

r2 to r12 : то же самое

r13 : На некоторых ARM-системах r13 - это указатель на стек (SP в процессорах INTEL). Я не уверен, играет ли r13 ту же роль в GBA, я могу только предупредить, чтобы вы были осторожны с ним, когда работаете со стеком.

r14 : Содержит адрес возврата для вызываемых процедур. Если вы их не используете, тогда можете делать с ним всё, что хотите.

r15 : Program Counter и флаги, так же как IP (Instruction Pointer в Intel). Он отличается от Intel'овского регистра IP тем, что у вас есть свободный доступ к нему, так же как и к любому другому регистру, но учтите, что его изменение приведет к тому, что управление будет передано на другой участок кода, а флаги изменятся.

Давайте проделаем небольшое математическое упражнение... 16 минус 3 (как правило) даёт нам 13 регистров. Не правда ли, это круто? Успокойтесь :).

Сейчас вы можете спросить, чем же на самом деле являются регистры. Регистры - это специальные участки памяти, которые являются частью процессора и не имеют действительного адреса, а известны только по своим именам. Регистры 32-х битны. Почти всё в любом ассемблерном языке использует регистры, поэтому вы должны знать их также хорошо, как своих родственников.

**Инструкции ассемблера ARM**

Во-первых, я хочу начать с того, что по моему мнению тот, кто придумал ассемблер ARM, является гением.

Во-вторых, я хочу представить моего хорошего друга CMP. Скажите ему привет, и возможно, только возможно, он станет и вашим другом. CMP расшифровывается как CoMPare (сравнить). Эта инструкция может сравнить регистр и число, регистр и регистр или регистр и ячейку памяти. Затем, после сравнивания CMP устанавливает флаги статуса, которые сообщают вам результат сравнения. Как вы можете вспомнить, регистр r15 содержит флаги. Они сообщают результат

сравнения. Инструкция CMP была специально создана для установки значения этих флагов и ни для чего более.

Флаги могут содержать следующие состояния:

```
EQ EQual / Равно
NE Not Equal / Не равно
VS oVerflow Set / Установка переполнения
VC oVerflow Clear / Очищение переполнения
HI HIgher / Выше
LS Lower or the Same / Ниже или то же самое
PL PLus / Плюс
MI MInus / Минус
CS Carry Set / Установка переноса
CC Carry Clear / Очищение переноса
GE Greater than or Equal / Больше или равно
GT Greater Than / Больше
LE Less than or Equal / Меньше или равно
LT Less Than / Меньше
Z is Zero / Ноль
NZ is not Zero / Не ноль
```

Эти состояния играют очень важную роль в ассемблере ARM.

ПРИМЕЧАНИЕ: флаги только сохраняют условия (Равно, Меньше и так далее). Больше они ничем не важны.

**Суффиксы условий**

Вы уже видели инструкцию B (Branch). Инструкция B делает то, что называется безусловным переходом (как GoTo в Basic или JMP в ассемблере INTEL). Но она может иметь суффикс (один из тех, что были перечислены выше), тогда она проверяет, соответствует ли ему состояние флагов. Если нет, то инструкция перехода просто не выполняется. Поэтому, если вы хотите проверить, равен ли регистр r0 регистру r4, а затем перейти к метке под названием label34, то вам необходимо написать следующий код:

```
CMP r0, r4     ; Комментарии в ассемблере идут после точки с запятой (;)
BEQ label34    ; B - инструкция перехода, а EQ - суффикс, означающий
               ; "Если Равно"
```

ПРИМЕЧАНИЕ : В Goldroad Assembler метки не нужно сопровождать (:), и кроме имени метки в строке ничего не должно быть.

ПРИМЕЧАНИЕ II: Писать CMP и BEQ заглавными буквами не обязательно, это просто для того, чтобы вам было более понятно.

Теперь вы знаете как делать переход в зависимости от состояния флагов, но что вы не знаете, так это то, что вы можете делать в зависимости от состояния флагов всё, что угодно, просто добавьте к любой инструкции нужный суффикс!

Также вам не обязательно использовать CMP для установления состояния флагов. Если вы хотите, чтобы, например, инструкция SUB (Subtract) устанавливала флаги, добавьте суффикс 'S' к инструкции (расшифровывается как 'Set flags'). Это полезно, если вы не хотите устанавливать состояние флагов лишней инструкцией CMP, поэтому сделать это и совершить переход, если результат был равен нулю, можно следующим образом:

```
SUBS r0,r1,0x0FF   ; Устанавливает флаги согласно результату выполнения
                   ; инструкции
ldrZ r0,=0x0FFFF   ; Загрузит в регистр r0 0x0FFFF только, если состояние
                     флагов равно Нулю.
```

**Обзор**

Сегодня мы изучили (ещё немного) о регистрах. Мы также узнали о гибкости инструкций ARM, которые могут выполняться (или не выполняться) в зависимости от состояния флагов. Мы узнали

много сегодня.

Завтра иы используем приобретённое сегодня знание ассемблера ARM для того, чтобы вывести картинку на экран GBA.

- Нечто невозможное является таковым только до той поры, пока оно не становится возможным /Jean-Luc Picard, Capt. , USS Enterprise/.

# GBA ASM - День 3: Больше ассемблера и помещение картинки на экран

**Автор:** Mike H, пер. Aquila

**Дата:** 2003-07-30

Во-первых, инструкции ADD и SUB расшифровываются как сложение (addition) и вычитание (substraction). Они обе могут иметь суффикс установки флагов, что особенно полезно, когда надо узнать, был ли результат операции равен нулю (0).

```
У обеих инструкций 3 операнда (параметра):

ADD r0, r1, 1

 r0 = r1 + 1
```

Инструкция ADD складывает последние два параметра и помещает их в первый.

ПРИМЕЧАНИЕ: я думаю, что первые два параметра должны быть регистрами, но я не уверен.

```
SUB r0, r1, 2

 r0 = r1 - 2
```

Инструкция SUB вычитает третий операнд из первого и помещает результат в первый.

Теперь, когда вы поняли, что такое ADD и SUB, давайте поместим на экран GBA картинку.

**Используемый инструмент**

Я использовал программу под названием Bimbo для экспорта картинки размером 240x160 в asm-формат. Скачайте её из раздела gfx-утилит на GBADev.Org. Теперь идите в Microsoft Paint (а лучше в какой-нибудь более приличный редактор - прим. пер.) и нарисуйте картинку размером 240 на 160 и сохраните её как 256-ти цветный битмап. Запустите Bimbo и нажмите кнопку "Открыть".

Нажмите кнопку "Export". При сохранение укажите "Assembler Source". Теперь начинается самое интересное. Bimbo не помещает '@' перед директивами в коде, который он генерирует, поэтому мы должны открыть наш любимый текстовый редактор и заменить DCW на @DCW. Чтобы сделать это как можно быстрее используйте функцию 'Replace All'.

Ок, теперь давайте напишем код для вывода нашей картинки на экран:

Замечание: переименуйте ваш asm-файл, в котором задана картинка, в pic.asm и откройте

```
Теперь идёт сам код:

; --Начинайте копировать здесь--

@include screen.h      ; подключаем screen.h
b start                ; перепрыгиваем через картинку (вы ведь не хотите,
                       ; чтобы GBA попытался её выполнить? :-) )
@include pic.asm       ; подключаем картинку
start                ; метка по имени start
@textarea        ; код начинается после этой директивы

; Так же как и в День 1 устанавливаем режим 3 с фоном 2

ldr r1,=REG_DISPCNT            ; также как и в День 1
ldr r0,=(MODE_3|BG2_ENABLE)
str r0,[r1]

; Приготовляемся к помещению картинки в VRAM (видеопамять)

ldr r0,=VRAM      ; делаем r0 указателем на VRAM (0x06000000).
```

```
ldr r1,=picBitmap  ; делаем r1 указателем на данные с вашей картинкой
ldr r3,=19200      ; мы хотим получить доступ к памяти 19200 раз
           ; цвет экрана в режиме 3 16-ти битен,
           ; 240x160 pixels = 38400
           ; 19200, так как за раз мы копируем 2 пиксела
           ; (инструкция сохранения 32-х битна)

label1           ; метка по имени label1
ldr r2,[r1]+4!   ; загружает в r2 следующие два (2) пикселя из данных с
                 ; с картинкой (на которые указывает r1).
str r2,[r0]+4!   ; сохраняет содержимое r2 (в нём 2 пикселя) в VRAM.
subs r3,r3,1     ; вычитает 1 количества раз, которое мы хотим получить
                 ; доступ к видеопамяти (VRAM).
                 ; обратите внимание на суффикс S в инструкции SUB - здесь
                 ; мы задаём состояние флагов.
bne label1       ; эта инструкция перехода проверяет, указывает ли
                 ; состояние флагов на нулевой результат. Если количество
                 ; раз, которое мы хотим получить доступ к VRAM, не равно
                 ; нулю (0), тогда продолжаем выполнение цикла.

infin         ; метка под названием infin
b infin       ; переход на infin

; предыдущие 2 строки являются бесконечным циклом. :-) while(1) {}

; сохраните этот файл как picscrn.asm
;--Заканчивайте копировать здесь--
```

Вы посмотрели код. Теперь давайте сассемблируем его!

Поместите pic.asm, screen.h и picscrn.asm в папку Goldroad. Сначала попытайтесь перетащить picscrn.asm в goldroad.exe, если он не создаст файл .GBA, тогда откройте DOS-окно, смените директорию на папку Goldroad'a и напечатайте:

Goldroad picscrn.asm и нажмите ввод

Эмулятор тоже должен быть в папке ассемблера, поэтому если это VisualboyAdvance, напечатайте:

```
Visualboyadvance picscrn.gba
```

Посмотрите на свою прелестную картинку на эмуляторе.

Поздра... Нет, есть ещё несколько вещей, которые требуют объяснений.

Вот эти 2 строки:

```
ldr r2,[r1]+4!
str r2,[r0]+4!
```

Вы, вероятно, интересуетесь, что такое +4!, правильно? Я тоже не знал, что это такое, пока не провёл некоторые исследования. '!' означает хитрую вещь под названием "write-back", поэтому строка 'ldr r2,[r1]+4!' на самом деле означает следующее:

Загрузить в регистр r2 содержимое того, на что указывает r1, а затем, после того как всё сделано, добавить 4 к r1. Чтобы добавить 4 к r1 до того, как произойдёт загрузка в r2, необходимо составить инструкцию следующим образом:

```
ldr r2,[r1+4]!

    * +4 внутри скобок '[]' означает "добавьте 4 к r1 ДО выполнения инструкции"

      * Восклицательный знак должен находится в конце всей инструкции и
        означает 'write-back'.
```

Причина для делания 'write-back' на +4 заключается в том, что мы загружаем 2 пикселя за раз, а каждый пиксель равен 2 байтам (2*2=4). И помните, что мы обрабатываем 2 пикселя за раз, потому что каждый пиксель занимает 2 байта, а инструкции загрузки и сохранения 32-х битны.

Ух! Это была тяжёлая работа! Поздравляю! Вы прошли безжалостный День 3.

Я не знаю ещё о чём будет День 4, либо об обработке нажатия на клавиатуру, либо о БИОСе GBA.

- Надеюсь, вам понравилось.

# GBA ASM - День 4: Обработка нажатий на клавиши

**Автор:** Mike H, пер. Aquila

**Дата:** 2003-08-02

**Некоторая информация об инструкциях MOV и LDR.**

Эта секция посвящена людям, которых всё ещё интересует разница между инструкциями MOV и LDR. У них у обоих одно и то же назначение - перемещать данные, правильно? Разница состоит в том, что инструкция MOV в основном используется для перемещения содержимого одного регистра в другой примерно вот таким образом:

```
mov r0,r1

  Если r1 = 100, то теперь r0 также = 100, поняли?
```

С помощью MOV вы можете перемещать в регистры числа, но они должны умещаться в 8 бит (1 байт). Просто используйте MOV для копирования регистров и LDR для загрузки в регистры значений примерно вот так:

```
ldr r0,=0x06000000

  LDR загружает регистр 32-х битным (4 байта) числом.
```

Чтобы подвести итоги этой маленькой секции, я скажу: "LDR используется для загрузки регистров значением, а MOV - для копирования регистра в регистр".

**О клавиатуре GBA.**

Ок, я допускаю, что настоящей клавиатуры в GBA нет!

У GBA есть 10 клавиш: ВВЕРХ, ВНИЗ, ВЛЕВО, ВПРАВО, A, B, START, SELECT, L и R. Когда вы нажимаете клавишу, очищается соответствующий бит. Регистр клавиатуры находится по адресу 0x04000130. В Keypad.h, который вы найдёте здесь (находится в архиве `wasm_files.zip`: `files/recovered/gbaasm04/keypad.h`). В нём определены следующие значения:

```
KEY_A 1
KEY_B 2
KEY_SELECT 4
KEY_START 8
KEY_RIGHT 16
KEY_LEFT 32
KEY_UP 64
KEY_DOWN 128
KEY_R 256
KEY_L 512
KEYS 0x04000130
```

Чтобы проверить, нажата ли клавиша, вы должны сделать следующее:

```
ldr r3,=KEYS     ; Эти две строки загружают в регистр r4
ldr r4,[r3]      ; содержимое ячейки памяти с состоянием клавиш
ands r4,r4,#KEY_A  ; Эта строка делает операцию AND над регистром клавиш и
                   ; кнопкой A. Обратите внимание, что у инструкции AND есть
                   ; суффик 'S', который уставновит флаги результата операции
```

Если клавиша A нажата, то следующая инструкция с суффиксом EQ выполнится ТОЛЬКО, если клавиша A нажата. Я думаю, что теперь очевидно, как протестировать другие клавиши.

Поэтому далее следует небольшая программа, иллюстрирующая обработку нажатия на клавиши.

```
;;--- НАЧАЛО КОДА ---;;
@include screen.h
@include keypad.h     ; убедитесь, что эти файлы находятся в папке с Goldroad
@textarea
```

```
ldr r4,=REG_DISPCNT
ldr r5,=(MODE_3|BG2_ENABLE)
str r5,[r4]

; Наша обычная процедура установки режима экрана 3.

ldr r7,=VRAM+2410

; r7 указывает на видеопамять в позиции 10,10.

start   ; метка под названием старт, вы уже должны отличать метки от других
        ; инструкций, так что это последний раз, когда объясняю подобные вещи.


ldr r8,=KEYS    ; загружает в r8 адрес ячейки памяти с состоянием клавиш
ldr r6,[r8]   ; получает её значение
ands r6,r6,#KEY_UP   ; бит и значение темперь должны быть равны (EQual),
                     ; если клавиша нажата и не равны, если клавиша не нажата.

ldr r6,=0x00FF00FF  ; загружает две (2) красные точки в r6
ldrne r6,=0x00000000  ; если результатом операции было неравенство, помещаем
                      ; две чёрные точки в r6

str r6,[r7]      ; помещаем пиксели, находящиеся в r6, в r7 (r7 указывает на
                 ; видеопамять 10,10)
b start     ; переходим обратно на начало
;;--- ЗАКАНЧИВАЕМ КОПИРОВАТЬ ---;;
```

Сассемблируйте этот файл!

ПРИМЕЧАНИЕ: чтобы получить результат от этой программы, вы должны держать клавишу нажатой.

Вы поняли, что проверка нажатий на клавиши очень важно?

День 5 будет посвящён спрайтам.

# GBA ASM - День 5: Помещение спрайта на экран

**Автор:** Mike H, пер. Aquila

**Дата:** 2003-08-14

**Введение** Этот день посвящён спрайтам, а если более точно тому, как вывести его на экран. Да, именно так - всё, что мы будем делать сегодня - это выведем на экран 16-ти цветный спрайт размером 32x32 на экран. **Конвертирование изображения**

Так же, как и когда мы помещали на экран изображение, нам необходимо сконвертировать его во что-то понимаемое Goldroad'ом. В этот раз мы будем использовать GIFs2SPRITEs.exe (находится в архиве `wasm_files.zip`: `files/recovered/gbaasm05/gifs2sprites.zip`), который может конвертировать GIF'ы в C-код. Теперь посмотрим, как Goldroad будет обрабатывать C-файл. Давай поиграем в короткий ответ/длинный ответ! Короткий ответ: он не умеет этого делать. Длинный ответ: мне пришлось сделать программу на VB, конвертирующую вывод GIFs2SPRITEs.exe в Goldroad'овские @DCW-выражения. Вы можете скачать GIFs2SPRITEs.exe здесь (он заархивирован). Теперь моя программа: CtoASM.exe (находится в архиве `wasm_files.zip`: `files/recovered/gbaasm05/ctoasm.zip`).

Теперь нарисуйте в MSPaint'e (лучше в Photoshop'e - прим.пер.) картинку размером 32x32, сохраните её как GIF-файл под названием sprit.gif в Goldroad'овской папке "tools". Теперь откройте DOS-подсказку и делайте следующее:

```
> cd Полный путь к папке, где находятся sprit.gif и gifs2sprites.exe

> gifs2sprites 16 sprit.h sprit.gif

> ctoasm
```

Обратите внимание, что это Windows-программа. Выберите sprit.h, в диалоге сохранения файла напишите "sprit.asm". Затем нажмите на кнопку GO и нажмите на OK, когда появится предпупреждение, что весь процесс может занять определённое количество времени.

Теперь, я надеюсь, у вас есть файл sprit.asm. Я знаю, что моя программа не эффективна, но тем не менее она работает.

**OAM (Object Attribute Memory)**

Если вы программировали под GBA на C раньше, то вы, возможно, знаете, что OAM - это где вы сообщаете GBA всё о вашем спрайте, например: размер, форма, цвета (16/256), X-позиция, Y-позиция, номер тайла (tile)/чара (char). OAM находится по адресу 0x7000000. OAM состоит из 128 элементов (0-127), т.е. GBA может отслеживать 128 спрайтов в OAM.

Помните, что вы также ограничены доступной памятью для изображений спрайтов. Каждый элемент OAM 8 байтов размеров, где 4 (0-3) аттрибута по два байта. Аттрибуты следующие:

*ПРИМЕЧАНИЕ: перечислены только важные части. Идите на The Pern Project за полным перечнем.

```
Аттрибут 0 :
Форма      |  Глубина цвета  | Y-позиция
Квадратный |   COLOR_16      | (позиция на экране по вертикали)
Высокий    |   COLOR_32      |
Широкий    |

Аттрибут 1 :
Size      |  X-позиция
Size_8    | (позиция на экране по горизонтали)
Size_16   |
Size_32   |
```

```
Аттрибут 2 :
Номер тайла размером 8x8 в памяти для спрайтов, с которого начинается данный
спрайт.

Аттрибут 3 :
Я думаю, что #3 предназначен для чего-то, связанного со вращением спрайтов.
Поэтому здесь о нём не будет рассказываться.
```

Давайте начнём писать код! Скачайте sprites.h (находится в архиве `wasm_files.zip`: `files/recovered/gbaasm05/sprites.h`).

```
;;-- Начало кода --;;

@include screen.h
@include sprites.h
b start              ; перепрыгиваем через данные
@include sprit.asm   ; это наша сконвертированная картинка
start:               ; я начал помещать двоеточия, когда понял, что это
                     ; лучше чем так, как это позволяет делать Goldroad
Init_Sprites         ; макрос в sprites.h, инициализирующий все x-позиции спрайтов
                     ; в значение 300

ldr r1,=REG_DISPCNT  ; делаем r1 указателем на регистр контроля экрана
ldr r2,=(MODE_0|BG2_ENABLE|OBJ_ENABLE|OBJ_MAP_1D)
str r2,[r1]          ; устанавливаем режим экрана 0 с бэкграундом 2,
                     ; включёнными спрайтами (OBJs) и мэппингом спрайтов 1d
                     ; (чтобы это ни было - прим. пер.) (кто захочет делать
                     ; вложенные циклы на ассемблере, если мы можем этого
                     ; избежать!)

;;-- Хватит копировать --!!
```

Я собираюсь объяснить эту часть, затем я покажу остальной код. Если вы думаете, что комментарии сами по себе всё хорошо объясняют, то всё равно прочитайте это :). Во-первых, мы подключаем все необходимые файлы, потом вызываем макрос (который я великодушно накодил для вас), чтобы инициализировать спрайты. Далее мы помещаем экран в режим 0, что вы можете найти довольно необычным, так как большинство программерских туториалов, которые я видел, используют режим 3 или 4. Я использовал режим 0, потому что это тайл-режим (tiled mode), а битмэп-режимы (3 или 4) разделят память под спрайты и используют половину для экрана или буфера.

Одновременнос с установлением режима 0 мы разрешаем использование спрайтов и бэкграунда. Также мы устанавливаем загрузку спрайтов в 1D. Если бы мы поставили 2D, то нам бы пришлось отслеживать x и y. Следующая часть кода:

```
;;-- Код продолжается --!!

ldr r0,=OAM        ; делаем r0 указателем на OAM (0x7000000)
ldr r1,=(SQUARE|COLOR_16|10)|((SIZE_32|10)<<16)
                   ; выше мы помещаем все аттрибуты 0 и 1 в r1, ниже будет
                   ; подробное объяснение
str r1,[r0]+4!     ; помещаем аттрибуты 0 и 1 в OAM и увеличиваем значение
                   ; указателя в OAM так, чтобы он указывал на аттрибут 2
ldr r1,=0          ; аттрибут 2 - это номер тайла, а наш спрайт начинается с первого: 0
str r1,[r0]        ; помещаем его в OAM. Если вы  хотели другой спрайт, вам
                   ; нужно было сделать "+4!"
                   ; to get the OAM pointer (r0) past Attribute 2 and past rotation stuff in
    Attribute 3 and to
                   ; the start of the next sprite's OAM entry
                   ; чтобы получить указатель OAM (r0), пропустите аттрибут 2,
                   ; то, что относится к вращению и перейдите к началу
                   ; следующего элемента OAM

;;-- Заканчивайте копировать здесь --!!
```

Теперь нужно сделать некоторое объяснение, в основном того, что касается второй строки:

```
ldr r1,=(SQUARE|COLOR_16|10)|((SIZE_32|10)<<16)
```

Обратите внимание, что так как каждый аттрибут длиной 2 байта, мы можем заполнить 32-х битный регистр двумя аттрибутами. Поэтому первые 3 вещи cORенные друг с другом - это аттрибут 0, а вторые две - это аттрибут 1. Аттрибут 1 сдвигается налево, чтобы быть в первых 16-ти битах регистра. Я понимаю, что внешне это выглядит наоборот, потому что кажется, что код помещает аттрибут 1 перед 0. Сначала я думал, что первые три cORенные вещи должны быть в первых 2-х байтах, это не заработало, поэтому я изменил код так, каким он является сейчас, и он работает.

"не чините то, что работает"

**Палитра (цвета спрайтов)**

Есть две палитры, одна для бэкграундов и одна для спрайтов (0x500200). Палитра - это куда помещаются цвета спрайтов. Также я должен сказать, что в C вы би использовали цикл FOR, который бы пробегал 256 раз, используя 16-ти битный указатель. В ассемблере все указатели ВСЕГДА 32-х битны, поэтому мы загружаем двойное количество данных за раз, обращаясь к памяти в два раза меньше (128 раз). Вот код, чтобы загрузить палитру из нашего сконвертированного GIF-файла.

```
;;-- Код продолжается --!!

ldr r1,=OBJPAL     ; r1 указывает на палитру спрайтов (0x500200)
ldr r2,=128        ; 128 - число раз, которое мы обращаемся к палитре
ldr r3,=pallete    ; палитра задана sprit.asm, который мы получили после
                   ; конвертирования

palloop:        ; этот цикл - типичный пример кода загрузки
ldr r7,[r3]+4!  ; он помещает 4 байта из палитры в r7, а потом
str r7,[r1]+4!  ; сохраняет 4 байт в палитру спрайтов. Каждый раз используется
                ; "обратная запись" каждого регистра на 4
subs r2,r2,1    ; эта строка вычитает из количества обращений к палитре 1
bne palloop     ; и делает так, чтобы флаги представляли состояние РАВНО

; if r2 = 0   (r2==0, это для любителей C). Это BNE-выражение выполнится если
; состояние флагов НЕ РАВНО, поэтому выполнение цикла остановится, когда мы
; закончим

;;-- Заканчивайте копировать здесь --!!
```

Я думаю, что в этот раз комментарии к коду действительно выполняют свое предназначение, поэтому переходим к следующей секции.

**CHARMEM (Sprite Picture Memory)**

SPM начинается с 0x6010000. Если бы мы были в битмапном режиме экрана (3 или 4), тогда бы нам пришлось начать с 512 тайла вместо 0. Вот почему мы в режиме 0.

```
;;-- Код продолжается --!!

ldr r1,=CHARMEM   ; смотрите описание ниже
ldr r2,=128
ldr r3,=obj0

sprloop:
ldr r7,[r3]+4!
str r7,[r1]+4!
subs r2,r2,1
bne sprloop

;;-- Заканчивайте копировать здесь --!!
```

Это тот же загружающий код, изменились только указатели (спрайт в CHARMEM), а obj0 - это наш сконвертированный спрайт, находящийся в sprit.asm.

**Конец кода и этого дня**

Конец кода делает бесконечный цикл и это всё.

```
;;-- CODE CONTINUE --!!

infiniteloop
b infiniteloop

;;-- END OF CODE --!!
```

*"Кому-то нужны объятия..."*
- Ryan Stiles, "Whose line is it anyway?"

# GBA ASM - День 6: Простые звуковые эффекты

**Автор:** Mike H, пер. Aquila

**Дата:** 2003-08-21

**О звуковых каналах GBA**

У GBA есть шесть звуковых каналов. Сегодня мы изучим только первые 4 (1-4), которые используются внутренним GBC GBA. Другие два канала - это прямые звуковые каналы, которые могут проигрывать сконвертированные звуковые файлы - о них мы сегодня говорить не будем. ПРИМЕЧАНИЕ: термин "direct sound" в данном контексте является не более чем полуофициальным названием 5 и 6 звуковых каналов GBA и никак не связан с DirectSound, который является частью мультимедийной библиотеки для Windows от Microsoft.

Чтобы мы могли воспользоваться определенными преимуществами, даруемые написанными мною самим макросами и константами, скачайте, пожалуйста, sound.h.

Сегодня мы узнаем, что нужно сделать, чтобы заставить выдать спикеры какой-нибудь звук. Чтобы заставить издать какой-нибудь звук каналы 1 или 2, нужно сделать следующее:

**Шаги генерации звука**

Включить поддержку звука (я предоставляю макрос SoundOn)

Включить требуемый звуковой канал (я предоставляю макрос Sx_Enable, где x - это нужный вам канал).

Загрузить REG_SOUNDxCNT_L правильным значением (вам придётся сделать это самим :-) ).

Загрузить REG_SOUNDxCNT_H правильным значением (вам придётся сделать это самим).

Загрузить REG_SOUNDxCNT_X правильным значением (сами).

Вот и всё, и поверьте мне, это не так уж много, но может немного смутить.

**Канал 1**

Во-первых, я дам вам рабочую программу, которая генерирует звук через 1 канал.

```
;-- Переписано на asm вручную с исходника на BELOGIC.COM с изменениями --;
@include screen.h
@include sound.h
@include keypad.h
@textarea

SoundOn      ; Шаг 1: мой макрос включает поддержку звука в REG_SOUNDCNT_X
             ; REG_SOUNDCNT_X контролирует всю звуковую систему.


S1_Enable    ; Шаг 2: мой макрос включает первый звуковой канал.
             ; S2_Enable включает канал... и так до 4.

ldr r1,=REG_SOUND1CNT_L   ; в этих двух строках я делаю шаги 3&4 за раз.
ldr r0,=0xf7800056
str r0,[r1]               ; сохраняем данные для шагов 3&4.

ldr r1,=REG_SOUND1CNT_X   ; Шаг 5: загружаем регистр.
ldr r0,=0x8400            ; Этот регистр ответственен за что-то, связанное с
str r0,[r1]               ; тем, что будет делать канал.

infin  ;  \
         ; - наш типичный бесконечный цикл.
b infin ; /
;-- Заканчивайте копировать здесь --;
```

Я посмотрел, что написано о звуковых регистрах на BeLogic.Com и понял не всё. Основываясь на их списке регистров, у звуковых канало 2 и 4 нет REG_SOUNDxCNT_X, а их регистры REG_SOUNDxCNT_H и REG_SOUNDxCNT_L также 16-ти битны, но разделены 2 байтами, поэтому мы можем писать в них более простым образом и не должны делать шаги 3&4 за раз, а можем разделить загрузки для регистров CNT_L и CNT_H.

Похоже, что канал 3 служит для проигрывания сэмплов, поэтому я о нём больше не буду говорить.

Канал 1 - это очень приятный амплитудный звукогенератор. Поскольку я не очень хорошо понимаю уравнения, используемые на BeLogic, я не могу их объяснить (вообще). Для того, чтобы получить хоть какой-то звук мне пришлось повозиться с используемыми значениям, пока я не получил достаточно хороший эффект.

**Канал 2**

Канал 2 - это то же самое, что и канал 1, но генерируются только квадратные (square) волны, поэтому нам не нужен регистр REG_SOUND2CNT_X. Вот тот же самый код, подправленный для использования канала 2:

```
;-- CONVERTED TO ASM BY HAND FROM BELOGIC.COM WITH MODIFICATIONS --;
@include screen.h  ;
@include sound.h   ;
@include keypad.h  ;
@textarea

SoundOn    ; Шаг 1: мой макрос, который включает поддержку звука.

S2_Enable  ; Шаг 2: мой макрос, включающий второй звуковой канал.

ldr r1,=REG_SOUND2CNT_L   ; Шаг 3: в случае со вторым каналом на нужно
                          ; объединять шаги 3 и 4
ldr r0,=0xff80            ; Регистры всё ещё 16-ти битны, но у нас в памяти
                          ; есть пространство для манёвра
str r0,[r1]               ; сохраняем данные

ldr r1,=REG_SOUND2CNT_H   ; Шаг 4:
ldr r0,=0xf400            ; я думаю, что для повторения звука на втором
str r0,[r1]               ; канале, вам необходимо инициализировать этот
                          ; регистр снова, а если это не будет работать, то
                          ; попробуйте проделать это с L и H.

; Шаг 5 не требуется для канала 2, так как у него нет регистра
; REG_SOUND2CNT_X. Канал 2 (и 4) не доставляют нам хлопоты, которые были у
; нас с каналом 1, так как дизайнеры системы памяти были так добры, что дали
; нам два лишних байта, дабы облегчить 32-х битное сохранение.

infin    ; \
         ;  - наш стандартный бесконечный цикл.
b infin  ; /
;-- STOP COPYING HERE --;
```

**Канал 4**

Теперь давайте получим что-нибудь от канала 4. Это псевдослучайный генератор шума. В коде мы просто заменить все '2' в именах регистра на '4' и вперёд:

```
;-- Сконвертировано из предыдущего примера --;
;-- Код должен работать, но заставить его это сделать у меня не получилось --;

@include screen.h
@include sound.h
@include keypad.h
@textarea

SoundOn

S4_Enable
ldr r1,=REG_SOUND4CNT_L
```

```
ldr r0,=0xf700
str r0,[r1]

ldr r1,=REG_SOUND4CNT_H
ldr r0,=0x8400
str r0,[r1]

infin

b infin
;-- Заканчивайте копировать здесь --;
```

Я не знаю, почему этот код не заработал. Я пытался скомпилировать C-код на BeLogic, но тот код тоже на работает.

ОБНОВЛЕНИЕ: в качестве подарка на День благодарения 2002, код заработал.

**Краткий итог**

Обратите внимание, что я не объяснял, за что отвечает каждый бит каждого регистра. Я уже говорил, что не понимаю уравнения и определённо не понимаю даже половину содержимого на BeLogic.com. Чтобы добиться больше эффектов, поиграйтесь со значениями регистров - это единственный путь, если не смотреть мануалы на BeLogic.com. Я уже рассказал вам всё (или даже больше), что знаю о звуке в GBA, теперь идите и напишите демку с подпрыгивающими шарами (вверх и вниз) :).

Спасибо,
- Mike H a.k.a GbaGuy

_- Нечто невозможное невозможно только до тех пор, пока оно не становится возможным.
- Jean-Luc Picard, Capt., USS Enterprise_

Я знаю, что уже использовал это изречение, но оно особенно подходит для программирования звука.

# GBA ASM - День 7: Всё о макросах

**Автор:** Mike H, пер. Aquila

**Дата:** 2003-09-23

**Зачем использовать макросы?**

Макросы - это алиасы для участков кода. Они делают вашу ассемблерную жизнь проще, так как вам не нужно запоминать куски кода (возможно, большие), которые вы часто употребляете. Чтобы вызвать макрос, напишите его имя, а через пробел параметры, разделённые запятыми. Когда ассемблер встречает макрос, он просто заменяет его вызов указанным в описании макроса кодом, подставляя, если это необходимо, параметры. Вот пример вызова макроса:

```
LoadRegRGB r0, 255, 0 , 0
```

Вот и всё! Макросы великолепно подходят для снижения количества печатаемого кода и повышения его читаемости.

**Простой макрос**

Макрос определяются примерно следующим образом: (что-то вроде шаблона макроса)

```
@macro macro_name arg1, arg2  ; макрос может иметь столько параметров,
                              ; сколько вам нужно, даже ни одного!

;;; здесь идёт код ;;;
@endm      ; конец макроса
```

Если вы не поняли, что это значит, вот код бесполезного макроса, загружающего 5 в r0:

```
@macro MakeR0Equal5
ldr r0,=5
@endm
```

Это макрос абсолютно бесполезен.

Теперь, я уверен, вы хотите узнать, как передавать аргументы (параметры) в макрос и использовать их в коде. Вот более полезная версия вышеприведённого макроса:

```
@macro LoadReg Register, Num
ldr Register,=Num
@endm
```

Если вы решите использовать этот макрос в реальном коде (не делайте этого, так как он абсолютно бесполезен!), вам следует поместить его либо в заголовочном файле, либо в начале кода. Я покажу вам, как выглядит вызов макроса, а затем объясню, как ассемблер обрабатывает его. Я вызываю этот макрос, чтобы загрузить 52 в r4. Смотрите:

```
LoadReg r4, 52
```

Вот как это выглядит в исходном коде. Когда ассемблер доходит до вызова этого макроса, он сначала определяет параметры, а затем производит замену:

• Register заменяется на r4
• Num заменяется на 52

Затем, после подстановки параметров, сам макрос заменяется на код в макросе. Поэтому после того, как вызов макроса обработан:

```
LoadReg r4, 52
```

Ассемблер удаляет его, а вместо него мы получаем:

```
ldr r4,=52
```

Обратите внимание, что параметры могут быть чем угодно, начиная от числа или регистра и кончая бессмысленным набором символов, которые могут привести к следующей ошибке:

Скажем, я попытался сделать следующее:

```
LoadReg $(#(*@*#_@, 52
```

Это переводится в:

```
ldr $(#(*@*#_@,=52
```

И я получаю несколько ошибок от Goldroad'a:

```
error : line 0 : syntax error
ldr $(
error : line 0 : syntax error
ldr $( (
error : line 0 : in expression - missing right parenthesis
ldr $( (*
error : line 0 : load out of range (max 0xffc b)
ldr $( (*
error : line 0 : syntax error
ldr $( (*
assembled with 5 errors
```

Теперь мы сделаем несколько более полезных макросов.

**Более полезные макросы**

Мне кажется, вам сейчас интересно, как задавать RGB-значения (на GBA это, фактически, BGR). Мы сделаем макрос, загружающий такое число в регистр. Итак, сколько аргуметов нам нужно. Давайте предположим... Так каким будет ваш ответ? Ок, я не знаю, что придумали вы, но я остановился на 4-ёх. 1 - это регистр, 2 - количество красного, 3 - количество зелёного, 4-ый - количество синего. Для тех, кто не знает, скажу, что на GBA RGBзначения задаются следующим образом:

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X B B B  B B G G  G G G R  R R R R

0-4 (R) = Красный
5-9 (G) = Зелёный
A-F (B) = Синий
```

Диаграмма выдрана из CowBite Spec

Обратите внимание, что каждое значение равно 5-ти битам, при том, что 16 бит неиспользовано. Поэтому на нужно сдвинуть биты заданных значений. Для этого я использовал содержимое типичного подобного макроса на C. Вот, что получилось:

```
@macro LoadRegRGB Register, Red, Green, Blue
ldr Register,=((Red)+(Green<<5)+(Blue<<10))
@endm
```

Разве не круто? Чтобы поместить красную точку, надо вызвать его следующим образом:

```
LoadRegRGB r3, 255, 0 , 0
```

И ассемблер воспримет это следующим образом:

```
ldr r3,=((255)+(0<<5)+(0<<10))
```

Обратите внимание, что вы уже должны понимать, на что ассемблер заменяет макрос, поэтому я больше не буду писать подобные строки (я голоден, мои пальцы болят и я замёрз).

Теперь давайте сделаем макрос для перемещения спрайтов, который мы использовали в Дне 5!

**Макрос, перемещающий спрайты**

Ок, нам нужен номер спрайта (0-127), если помните. А если нет, то не страшно :). Нам также нужна новая позиция по x и y. Это в сумме составляет 3 аргумента.

Номер спрайта мы умножим на 8, чтобы добраться до начала элементов спрайтов в OAM. 8 - это длина (в байтах) элементов спрайтов (Аттрибут0 - Аттрубит3). Затем загрузим в регистр содержимое аттрибутов 0&1 (аттрибуты 16-ти, а регистры - 32-х битны, поэтому у нас нет выбора). Затем мы сдвинем содержммое регистра так, чтобы в нём был только аттрибут 0 (в младших 16 битах). Далее мы выполним операцию побитового AND над регистром, используя значение 0xFF00, чтобы очистить старую y-позицию, а затем сделаем побитовый OR над нашей новой y-позцией. Так как наши операции загрузки и сохранения 32-х битны, мы должны загрузить два регистра аттрибутами 0 и 1. Мы не можем сделать это за один раз, потому что когда мы сохраняем аттрибут 0, мы обнуляем аттрибут 1. Поэтому мы также делаем побитовый AND над регистром, в котором находится аттрибут 1 (после некоторого сдлвига)

Ух, ты! Какой здоровенный параграф получился. Я надеюсь, что вы поняли его смысл. Может быть вам нужно поглядеть на конкретный код? Прекрасно. Вот моя реализация:

```
@macro MoveSprite Sprite, X, Y
ldr r2,=(OAM+(Sprite*8))  ; получаем начало аттрибутов спрайта в OAM
ldr r3,[r2]           ; загружаем в r3 аттрибуты 0 и 1
mov r3,r3, lsr#16     ; делаем сдвиг, чтобы в r3 был только аттрибут 0
ldr r4,[r2]           ; загружаем в r4 аттрибуты 0 и 1
mov r4,r4, lsl#16     ; делаем сдвиг, чтобы в r4 был только аттрибут 1
mov r4,r4, lsr#16     ; получаем значение r4 в младших битах
and r3,r3,#0xFF00     ; очищаем старое значение y
and r4,r4,#0xFE00    ; очищаем старое значение x
orr r3,r3,#X       ; устанавливаем новое значение y
orr r4,r4,#Y       ; устанавливаем новое значение x
mov r3,r3, lsl#16  ; готовимся добавить их
orr r4,r4,r3       ; добавляем их вместе, чтобы они оба были в r4
str r4,[r2]        ; сохраняем в OAV, всё! :)
@endm
```

LSR (Логический Сдвиг впРаво) и LSL (Логический Сдвиг вЛево) - это опции, а не инструкции. Это означает, что они должны употребляться только вместе с инструкциями. Они могут быть присоединены к концу инструкции MOV.

Вам понравился этот макрос?

**Подведение итогов**

Хотя иногда создание макроса может отнимать ощутимое количество времени б(создание макроса MoveSprite отняло у меня 4 часа), их использование всегда оправдывает затраченные усилия, потому что как только вы создали макрос, вы можете использовать его так часто, как это вам нужно. Что удобнее, писать каждый раз 13 строчек кода, когда вам нужно переместить спрайт, или просто написать "MoveSprite 0, 35, 50"?

# Программирование на ассемблере под PocketPC

**Автор:** Broken Sword

**Дата:** 2004-07-07

Для начала разберемся с основными понятиями и определениями: PocketPC и WindowsCE – это (с недавних пор) одна и та же операционная система реального времени (real-timeOS); предназначена она для т.н. встраиваемых (embedded) систем (КПК, банкоматы, бортовые компьютеры в автомобилях и т.п.).

Одной из главных характеристик систем реального времени является стабильность. Не беспокойтесь – Microsoft не отступила от своих традиций: WinCE подвисает довольно часто, причем даже от вполне безобидных ошибок в прикладных программах. Поэтому прежде чем запускать программу в КПК ее нужно тщательно обследовать на эмуляторе.

Еще одно преймущество WindowsCE– компактность. В ней реализовано в десять раз меньше API-функций, чем, к примеру, в WinNT. Для сравнения – главная системная библиотека в WindowsXP (kernel32.dll) весит 914 Кб, тогда как аналогичная библиотека из WindowsCE (coredll.dll) – всего 509 Кб. О чем говорят эти цифры? Да ни о чем J. Просто из coredll.dll выкинули много старых, ненужных для совместимости с Win9xфункций. Правда, добавили несколько новых. Вообще, некорректно сравнивать настольные ОС со встроенными, поэтому оставим эту тему.

Принцип модульности – вот основное преимущество этой, да и всех других встраиваемых ОС: WinCE состоит из множества модулей (около 220 «exe» и «dll» файлов, причем каждый модуль состоит из нескольких компонент («lib»), что делает эту ОС совсем пластилиновой), которые можно легко отключать и включать, т.е. WinCE – это конструктор, который может собрать любой под свои конкретные нужды с помощью PlatformBuilder-а, и если, к примеру, вашему холодильнику не нужны сокеты – вы просто не включаете их в конечный build. Что-то подобное можно наблюдать при инсталляции настольных версий Win – нам дают возможность избавиться от некоторых компонент, однако ядро и системные библиотеки остаются неизменными. Здесь же можно менять абсолютно все, вплоть до ядра. Таким образом, достигается минимальный размер и максимальное быстродействие. В принципе, по заверениям разработчиков, всю WinCE можно уместить в 400 Кб и она будет работать с 32 Кб оперативной памяти. С графическим интерфейсом (модуль GWES) она потянет уже на 4 Мб.

WinCE поддерживает множество процессоров (ARM, MIPS, x86), полный список можно посмотреть здесь:

http://www.microsoft.com/windowsce/Embedded/resources/processors.asp

Архитектура WinCE не является темой данной статьи, об этом вы можете почитать в MSDN. Поэтому перейдем непосредственно к главному.

Что необходимо для того, чтобы написать простейшую программку на асме для WinCE? Прежде всего – это … (нет, не больная психика, хотя не без этого) ассемблер для нужного проца, линкер и само устройство (или его эмулятор). Мой КПК Toshibae755 стоит на IntelPXA255 проце (который, в свою очередь, стоит на архитектуре ARM 5TE) . Поэтому в качестве ассемблера я использовал MSARMASM, который идет вместе с eVC (и который вы можете скачать по ссылке в конце статьи). Линкер – самый обычный MSlink.

Следует отметить, что ARMASM снабдили какой-то кой-как, на скору рученьку слепленной документацией (кое-что есть в MSDN, кое что есть в документации от eVC), хотя, очень возможно, что он и не содержит никаких неописанных возможностей.

В Интернете существует достаточное количество качественных ARM-ассемблеров, однако ни один из известных мне не знает ни про COFF, ни про PE (а именно он, кстати, и используется в WinCE).

Сразу признаюсь – на то чтобы простейшее «HelloWorld» приложение скомпилировалось и нормально запустилось я убил целый день. Однако результат оправдывает затраченные усилия – в итоге получился в три раза меньший по сравнению с C-шным exe-шник, а критерий размера ох как критичен для emb-систем, что бы там ни говорили настольные сотоварищи.

Предполагается, что вы более-менее знакомы с программированием под Win32 на ассемблере и с ARM-овским асмом.

Код стандартного «HelloWorld» выглядит следующим образом:

```
;--------------------cut here--------------------------------------------------------------------
    ---
  include  wince.inc

  IMPORT MessageBoxW
  IMPORT ExitThread

  EXPORT start

  AREA  .text, CODE
start
  eor  R0, R0, R0
  adr  R1, mestext
  adr  R2, mestit
  mov  R3, #MB_OK
  bl  MessageBoxW

  eor  R0, R0, R0
  bl  ExitThread

mestext dcb "H",0,"e",0,"l",0,"l",0,"o",0," ",0,"w",0,"o",0,"r",0,"l",0,"d",0,0,0
mestit  dcb "A",0,"S",0,"M",0,0,0

  END
;--------------------cut here--------------------------------------------------------------------
    ---
```

Пробежимся по каждой строчке кода.

```
include wince.inc
```

*wince.__inc – быстро слепленный файлик, пока что кроме определений для MessageBoxW ничего не содержит. Надеюсь, когда-нибудь в него добавятся новые строки... Кстати, наблюдательный читатель сразу обратит внимание на то, что численно все определения для WinCE совпадают с «настольными», однако их значительно меньше…*

```
IMPORT MessageBoxW
IMPORT ExitThread
```

*armasm не поддерживает директиву типа includelib, поэтому каждую импортируемую функцию нужно указывать вот таким вот образом.*

```
EXPORT start
```

*Дело в том, что без функции WinMainCRTStartup (которая автоматически создается в C) для платформы WindowsCE link.__exe не сможет создать exe-файл - не найдя этой функции в объектнике, созданном armasm-ом, он просто грязно выругается. Поэтому мало того, что метку точки входа необходимо указать в экспорте, так еще и в параметрах к link.__exe об этом нужно явно сказать при помощи опции /__entry. Помнится,__ml.__exeсам делал всю подобную грязную работу и автоматически «назначал» точкой входа метку, которая была указана после директивы END.__armasm настолько сырой (и, складывается впечатление – никому не нужный), что подобных наворотов просто не поддерживает.*

```
AREA  .text, CODE
```

*Здесь все понятно – это начало сегмента кода (.__text – это имя, и может быть любым)*

```
eor  R0, R0, R0
adr  R1, mestext
adr  R2, mestit
mov  R3, #MB_OK
bl  MessageBoxW
```

*В отличие от x86 (где параметры передаются в стеке) здесь параметры в функции передаются в регистрах (первый – в R0, второй – в R1 и т.д.). Результат возвращается в R0 (аналог EAX на x86). Здесь происходит вызов функции MessageBoxW – на экран выводится окошко с текстом. Почему W? Дело в том, что WinCE поддерживает только Unicode. MessageBoxA в coredll даже не пахнет. Хорошо это или плохо – программисту на асме вообщем-то как то все равно :), места больше в два раза жрет только…*

```
eor  R0, R0, R0
bl  ExitThread
```

*Почему ExitThread, а не ExitProcess? Все достаточно загадочно – дело в том, что coredll вообще не экспортирует ExitProcess, хотя в MSDN-е написано обратное. Близкий родственник – TerminateProcess есть, однако применять его крайне не рекомендуется. Еще больше смутила следующая строка, обнаруженная в одном из инклюдов к eVC:*

**_ #define ExitProcess(code) TerminateProcess (GetCurrentProcess (), (code)) _**

*Если у кого есть мысли по поводу того, зачем было избавляться от ExitProcess – прошу в комментарии. ExitThread из основного потока аналогичен ExitProcess, именно им я воспользовался (все таки, куда делся ExitProcess?)*

С mestext и mestit все и так понятно – такая убогость возникла из-за того, что dcw для ASCII-символов в armasm-е не поддерживается, а писать макрос как-то не было времени.

Несколько слов по поводу компиляции:

armasm.exe -32 -cpu xscale entry.s

**-32**– генерить 32-битный код. **xscale** – мой проц

link.exe coredll.lib /nologo /entry:"start" /subsystem:windowsce /machine:arm "entry.obj"

/machine:arm – без этого параметра PocketPC ругается и кричит, что формат файла неправильный. На самом деле все что он делает – меняет байт Machine в заголовке PE с Thumbна ARM, хотя почему там оказался 16-битный ARM вместо 32-х битного? Это значит лишь то, что параметр -32 в armasm просто не работает… L

Помните, в начале статьи я упоминал о глюкавости WinCE? Можете сами поэкспериментировать. Задайте, например, выравнивание секций на 4 байта. В итоге вы получите еще в три раза меньший файл… (716 байт) и безумное поведение WinCE при попытке его запустить.

Все необходимые файлы (кроме link.exe) можно скачать тут (находится в архиве `wasm_files.zip`: `files/recovered/pocketpc/asm_ce.zip`).

Это все. В следующей статье по WinCE попытаемся рассмотреть более насущные проблемы – например, написание драйвера под эту замечательную ОС.

# Кое-что об устойчивости и защищенности Windows CE на ARM-платформе

**Автор:** Broken Sword

**Дата:** 2004-07-21

Надеюсь, не найдется человека, который никогда не слышал о таких нелепых дырах в защите Win 9x, как незащищенные системные структуры и последствия к которым приводит такая халатность (создание своего callgate-а в таблице дескрипторов, или запись в незащищенные таблицы страниц). В данной статье будет проведено небольшое исследование, посвященное аналогичным проблемам WinCE на ARM v5 - платформе. Почему именно на ARM? Во-первых, потому что другой просто нет под рукой. Во-вторых, это чуть ли не самая распространенная на сегодняшний день платформа для КПК (последние процессоры Intel xScale и Samsung используют именно эту архитектуру).

Для начала придется рассмотреть основные защитные механизмы ARM, использующиеся на аппаратном уровне. По ходу дела будут приводиться аналогии (иногда не совсем удачные) архитектуры x86 (Intel).

Наверное, у всех слово «сопроцессор» так или иначе ассоциируется с обработкой чисел с плавающей запятой. Однако, на самом деле «сопроцессор» гораздо более обширное и глубокое понятие. Так, все управляющие структуры (аналоги регистров CRn в x86) в ARM расположены в регистрах сопроцессоров.

*Кстати,__ARM поддерживает до 16 сопроцессоров.*

Только два сопроцессора (CP0 и CP1) отвечают за обработку чисел с плавающей запятой.

Сопроцессор управления системой (System Control Coprocessor) **CP** 15 (можно назвать аналогом CR-регистров в x86) отвечает за управление внутренней архитектурой (MMU, кэш, буфера), в том числе и защитными механизмами. Состоит он из 15 регистров. Я не буду приводить здесь структуры этих регистров, т.к. по сути статья превратиться в мануал по ARM-у, а затрону только защитные аспекты.

Структуры, отвечающие за аппаратную защищенность в ARM v5:

**Запись вида** CPn_CRm[x:y] **означает «биты [x:y] регистра CRm в сопроцессоре CPn».**

| Название | Где |
|---|---|
| Mode bits in Current Program Status Register | CPSR [4:0] |
| ROM protection bit (R) | CP15_CR01 [9] |
| System protection bit (S) | CP15_CR01 [8] |
| Memory management unit enable/disable (M) | CP15_CR01 [0] |
| Domain Access Control Register | CP15_CR03 [31:0] |
| Coprocessor Access Register | CP15_CR15 [13:0] |
| Access Permissions | TTE |

**Mode bits in Current Program Status Register** – 5 бит в регистре CPSR (Current Program Status Register). Раньше эти биты (а также другие аналоги флагов из EFLAGS: ZF, CF и т.п.) находились в регистре R15 (PC) (аналог EIP), но с переходом на 32-битную адресацию гордо вынесены в отдельный регистр (R16 – CPSR). Вот молодцы все-таки ребята из Acorn – не пошли они по аналогиям Intel-овских Франкенштейнов.

Аналогом «битов режима» в x86 могут послужить биты CPL из селектора CS. Как видим, разнообразие режимов по сравнению с x86 довольно велико. Более того, мы можем добавлять свои собственные режимы! Так Intel и поступила в своем xScale, добавив новый отладочный (debug) режим. Видите, насколько более профессиональна и гибка ARM-архитектура по сравнению с x86?

**ROM protection bit (R)**

**System protection bit (S)**

Два этих бита рассматриваются только на пару с битами Access Permissions из записей таблиц трансляций (аналоги каталогов и таблиц страниц на x86). Вот нехитрая табличка:

| AP | S | R | Привилегированный режим | Режим пользователя |
|---|---|---|---|---|
| 00 | 0 | 0 | *нет доступа* | *нет доступа* |
| 00 | 1 | 0 | *только чтение* | *нет доступа* |
| 00 | 0 | 1 | *только чтение* | *только чтение* |
| 00 | 1 | 1 | - | - |
| 01 | x | x | *чтение/запись* | *нет доступа* |
| 10 | x | x | *чтение/запись* | *только чтение* |
| 11 | x | X | *чтение/запись* | *чтение/запись* |

**Memory management unit (M)** – бит отвечает за включение/выключение модуля управления памятью (MMU). Одной из важнейших функций MMU является преобразование логических адресов в физические.

*Аналогично в x 86 бит 31 в CR 0 отвечает за включение страничной адресации памяти.*

Однако, в ARM-архитектуре вообще нет понятия *сегментной* организации памяти – при отключении MMU логический адрес = физическому.

*Intel -у пришлось тянуть сегментные регистры из реального режима… бедняги.*

Пара слов о самом механизме преобразования виртуальных адресов в физические: точно так же, как и в x86 существует две таблицы трансляции адресов (первого и второго уровней).

*Кстати, таблицу трансляции первого уровня Intel -овцы у себя окрестили «Каталогом страниц».*

Физический адрес таблицы преобразования первого уровня хранится в регистре CP15_CR2.

*Аналог CP 15___CR 2 в x 86 – регистр CR 3.*

**Domain Access Control Register** – 32-разрядный регистр, содержащий права доступа к 16 возможным доменам (по два бита на домен). **Домен** (domain) – набор секций, больших и малых страниц памяти, права доступа к которым можно быстро изменить с помощью записи в этот регистр (CP15_CR03). То есть каждой странице (либо секции; см. далее, что такое секция) мы указываем, к какому домену она принадлежит в записях таблиц трансляций. Затем, при желании, изменяя всего два бита в регистре DACR, мы можем изменить права доступа хоть ко всей памяти. **Секция** (section) – область памяти размером 1 МБ, доступная через одноуровневый механизм преобразования адреса.

*Аналог x 86: адресация с использованием 4Мб страниц.*

*Страница*(page) – область памяти размером 1 (tiny), 4 (small) либо 64 (large) Кб. *Coprocessor Access Register*– 32-х разрядный регистр, содержащий права доступа к 16 возможным сопроцессорам

(точнее к его регистрам) - по одному биту на сопроцессор. Старшие 16 бит – зарезервированы. 16 младших и отвечают за 16 сопроцессоров. Хочется отметить, что биты 14 и 15 ВСЕГДА равны нулю. Что сие означает? А означает это то, что доступ к сопроцессорам 14 и 15 (CP14 и CP15) не может регулироваться этим регистром и возможен только в режиме с системными привилегиями. *Access Permissions*

См. табличку к битам R и S. Стоит только отметить, что AP при использовании секций задается в таблице трансляции первого уровня, а при использовании страниц – в таблице второго уровня, причем в этом случае задается сразу четыре разных значения AP – каждое отвечает за свою четверть области страницы.

Итак, приступим. Обязательно следует отметить, что эксперименты проводились на КПК Toshiba e755 с PocketPC 4.20.1081 (build 13100). Почему это так важно? Да потому, что вся манера поведения (в том числе и защитная) встроенных ОС целиком и полностью зависит от OEM (Original Equipment Manufacturer), т.е. в данном случае от программистов из компании Toshiba. Имеется ввиду следующее: тактику поведения ОС по отношению к чужеродным (user-level) программам определяют параметры, заданные при компиляции окончательного билда (я имею ввиду такие механизмы, как установки флагов ROMFLAGS в секции CONFIG, полноценное использование функции **OEMCertifyModule,** и т.п.).

*Несколько слов о механизме OEMCertifyModule. Дело в том, что изначально, при проектировании окончательного билда, __OEM может задать т.н. доверительные программы, расположив их в секции MODULES . Только эти программы могут использовать все API и имеют привилегии. Остальным недоступна целая группа API -функций (т.н.__Trusted ___API_ ), полный список которых приведен в MSDN -е. При загрузке любого процесса загрузчик передает управление на функцию OEMCertifyModule, в которой производится проверка загружаемого процесса на «доверительность» - по определенным сигнатурам в заголовке.*

Т.о. во всех защитных слабостях виноваты только программисты, компилирующие ядро под определенное устройство, и никто больше, т.к. ни ARM, ни WinCE (?) сами по себе не содержат уязвимостей по крайней мере на уровне OAL. Поэтому интересно было бы провести подобные исследования на КПК от других производителей.

Итак, попробуем почитать что-нибудь по адресу 0x80000000:

```
ldr  R0, =0x80000000
ldr  R1, [R0]
```

Хе… нет проблем! Никаких Access Violation! И вот что повергает в «легкий трепет» с самого начала: ВСЕ процессы WinCE запускает в привилегированном режиме (privileged-mode)! Где-то мы все это уже наблюдали…

*Для справки: в ARM -процессорах привилегированным режимом называется ЛЮБОЙ режим, отличный от пользовательского (__user -__mode ).*

Проверить сие очень просто: мы можем свободно пользоваться ЛЮБЫМИ привилегированными инструкциями, а также читать/писать в регистры CP15. Как думаете, что произойдет если процессу, к примеру, вздумается изменить себе адрес таблицы преобразования первого уровня? Или вовсе отключить MMU? Пожалуйста! Ничего ужасающего не произойдет (хотя бы потому, что ОС сидит в ROM-е и всегда на помощь придет Hard Reset) – просто все очень крепко зависнет.

*Виснет КПК довольно прозаично: никаких BSOD , просто на дисплее замирает последний кадр и подсветка остается в том состоянии, в котором была на момент ошибки – и все. Оживить его можно только утопленной сбоку кнопочкой Soft ___Reset_ -а.*

Проверим, с какими привилегиями работает наш процесс:

```
MRS  R0, CPRS
R0=0x1F
```

Хм, интересно… Смотрим табличку:

| M[4:0] | Режим | Описание |
|---|---|---|
| *10000* | User | Пользовательский режим (аналог процесса с CPL=3 в x86). |
| *10001* | FIQ | Режим «быстрой» обработки прерывания. Т.е. когда время на обработку прерывания сверхкритично! FIQ может «перебить» IRQ, но не наоборот. |
| *10010* | IRQ | Режим, в который процессор переключается после прихода сигнала прерываний (от клавиатуры, дисковода, таймера и т.п.). |
| *10011* | Supervisor | Режим супервизора (аналог процесса с CPL=0 в x86). |
| *10111* | Abort | В этот режим процессор входит после ошибок (что-то типа #GP). |
| *11011* | Undefined | Аналог #UD на x86 (Invalid Opcode Exception) |
| *11111* | System | Привилегированный режим пользователя |

*В xScale , как уже упоминалось выше, добавлен еще один – debug ___(0_ x 15) режим.*

У нас последний случай (0x15), т.е. процесс запускается в привилегированном режиме System. Что это за режим? System – это некий промежуточный уровень между Supervisor и User режимами. Т.е. мы уже не пользователь со связанными ногами, но еще и не супервизор. Кстати сказать, практически никакой разницы между System и Supervisor-ом нет – лишь в теории в будущих модификациях ARM-а System-режиму могут порезать привилегий.

Что ж, попробуем повысить себе уровень до супервизора:

```
MRS  R0,CPSR   ; CPSR -> R0
BIC  R0,R0,#0x1F  ; очистим биты режима
ORR  R0,R0,#0x13  ; режим Supervisor-а
MSR  CPSR_C,R0  ; R0 -> CPSR
```

Как думаете, что произошло после MSR CPSR_C,R0 ??? Мда… похоже WinCE по любому поводу впадает в глубокую кому. Не понимаю, как она вообще работать может…

*Не следует также забывать, что это именно та ОС, которая стоит в большинстве банкоматов и бортовых компах дорогих автомобилей (т.е. фактически отвечает за жизни миллионов людей !__J ).*

Поехали дальше. Команды CPS (изменить биты режима в CPRS) на ARM v5 еще не было, она появилась только на v6.

Есть еще один способ получить супервизора – через SWI. SWI (Software Interrupt) механизм вызова программных прерываний, что то типа INT на x86. При этом мы автоматически получаем супервизора. Проблема только в том, что мы не знаем какое именно прерывание даст нам супервизора. И вообще, не нужен нам никакой супервизор – мы и так обладаем всеми возможными привилегиями.

Попробуем посмотреть, что у нас с доменами.

```
mrc  P15, 0, R0, C3, C0, 0
```

Ну слава Богу! Хоть здесь нули! Кстати, ничего не мешает нам исправить их на единицы, что мы и проделаем:

```
mvn R0, #0
mcr P15, 0, R0, C3, C0, 0
```

Ну вот и все – теперь защита на уровне доменов отключена.

Вообще я в замешательстве – о чем думали производители? На кого может быть рассчитана система, где любой процесс может творить что ему угодно, причем обработкой исключений никто не хочет заниматься? Полнейшая анархия!

Позже, на XDA-developers.com я обнаружил старую статью об управлению памятью – там мужик поднимает точно такие-же проблемы.

Ладно, вернемся к программированию. Тут (находится в архиве `wasm_files.zip`: `files/recovered/securewincearm/v2p.zip`) вы можете скачать мою прогу по конвертации виртуального адреса в физический. Обратите внимание, что все секции (страницы) расположены в нулевом домене. Все секции имеют AP = 1, т.е. мы с нашими привилегиями можем гадить в любые адреса. Программа работает с одним ограничением – обрабатывает только секции. Обработать страницы мне не удалось по простой причине – VirtualCopy в большинстве случаев просто вешает систему, причем происходит это с каким-то мерзким пост-эффектом, когда система виснет на второй, а то и на третий VirtualCopy. Позже по данной теме в нете я обнаружил чела с теми же проблемами – его пост в форуме датирован 2002 годом, и там он утверждал, что на какой-то модели Casio VirtualCopy работало нормально. В любом случае, в исходнике есть строка, помеченная как “comment me”. Попробуйте запуститься, закомментировав эту строку и напишите о результатах в обсуждениях.

Хотелось бы поведать об одном кошмаре программиста на ассемблере под PocketPC – программа запускается и прекрасно работает на устройстве (!) под отладчиком, однако при запуске непосредственно на устройстве (без отладчика) либо, как вы уже догадались, все виснет, либо просто ничего не происходит. К обнаруженным мною подобным причудам можно отнести следующую:

- в маленьких прогах я не создаю секции данных, а располагаю их в секции кода, задав ей атрибуты RWE. Так вот я с самого начала забыл про эти атрибуты и что удивительно – при запуске под отладчиком ему плевать на атрибуты секции, он как то сам то ли по уму то ли по глупости разрешает запись в сегмент кода (при загрузке в устройство сам выставляет атрибуты). Естественно, загрузив прогу без отладчика мне долго пришлось выискивать проблему.

Также очень намучился со следующим - в исходнике есть код, по типу такого:

```
 …
BL PROC1
 …

PROC1
 …
 BL PROC2
 …
 mov PC,LR

PROC2
 …
 mov PC,LR
```

По привычке, руководствуясь интеловской психологией, я как-то совершенно не задумывался о выходе из вложенных процедур – у интела нужный адрес всегда на вершине стека и мы выходим по ret-у. Из-за того, что в ARM-е адрес возврата сохраняется в регистр R14 (LR) во вложенных процедурах в начале нужно сохранять старое значение LR. Опять же, пустяковая ошибка, но нервов попортила немало.

Ну что еще по мелочи – по аналогии с *nix если параметров слишком много (больше четырех), то часть из них передается в стеке.

Да, стоит также отметить, что в WinCE по GetLastError ничего кроме 0x57 (неверный параметр) вы не добьетесь.

**P. S.** Сколько же мата было в сторону пресловутого OEM за невозможность Hard Reset-нуть КПК без доставания из крэдла… (утопленная кнопка расположена сбоку и закрывается пластмассовой стенкой). После примерно полсотни зависаний процесс сброса КПК начал проводится с особым остервенением – честно говоря, не ожидал что КПК выживет к концу статьи. Один раз произошла вообще уникальная вещь – после очередного VirtualCopy КПК завис так, что даже Hard Reset не помогал – пришлось полностью убивать всю систему переключателем.

**P. P. S.** Пара слов о вирусах под WinCE. До сих пор якобы не зарегистрировано ни единого вируса для WinCE. Такая ситуация казалась мне довольно непонятной, т.к. как не может быть ни единого вируса в такой, во-первых, уязвимой системе как WinCE, а во вторых – хочешь не хочешь, а запиши свой код во все файлы в текущей директории и тебя окрестят вирусом. Только мне захотелось «прославиться» на этом поприще, как вдруг абсолютно случайно зайдя на сайт pocketnow.com обнаружилось, что сегодня (!) (19 июля 2004 года) был создан первый вирус под PocketPC (парнем из 29A). Тут же в Yahoo по запросу Windows CE virus первые пять ссылок указывают на этот замечательный случай (хотя еще вчера не было ни одной по теме). Я не удивлюсь, если о нем еще будут неделю трубить по всем новостям. Ну что ж, единственный шанс прославиться безвозвратно упущен… Вирус, кстати, довольно безобиден, и даже сам спрашивает у пользователя, можно ли ему нашалить в системе. Но ничего, думаю троян произведет гораздо больший эффект. Заодно можно будет узнать, сколько же людей в мире пользуются инетом через КПК. Кто хочет поучаствовать и у кого есть определенный опыт и интерес к удаленному администрированию – прошу в соавторы (хоть срок мотать вместе не скучно будет).

## Ссылки


• **ARM Architecture Reference Manual.pdf** — можно бесплатно заказать диск со всей документацией к ARM-у с сайта www.arm.com; в Украину первым классом летит три дня.
• **MSDN** — ну эта забесплатно и с места не сдвинется.

# GBA ASM - День 8: Загрузка данных с помощью DMA

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-07-07

**Зачем вообще использовать DMA?**

DMA (Direct Memory Access) - это относительно быстрый способ загрузки данных в память. До сих пор мы использовали обычный цикл загрузки данных. Преимущества использования DMA по сравнение с этим подходом следующие:

• DMA предположительно быстрее.
• В силу каких-то причин я получаю значительно меньший размер бинарника, когда использую DMA.
• Нам больше не нужно заморачиваться с циклом.
• Использование DMA для загрузки данных проще, чем использование цикла.
• Чтобы поместить необходимую информацию в три (3) DMA-регистра, потребуется использовать два из R0-R12 обычных регистров. 3 загрузки и сохрания займут 6 строк кода, что по размеру больше, чем загрузочный цикл, но это проще.
• Возможно, есть что-то ещё, но мне ничего не приходит в голову.

Поэтому, почему бы не использовать DMA? Для начала давайте посмотрим, какие возможности оно даёт и как их можно использовать.

**Об опциях DMA**

DMA имеет 4 канала (0-3). 0 - особый, пока его не используйте. 1-3 доступны для использования. Каналы 1 и 2 также также можно использовать для работы со звуком. В общем-то, между 1, 2 и 3 каналом не слишком много разницы, за исключением того, что они имеют разные приоритеты.

Приоритеты необходимы, так как DMA останавливает работу процессора, пока не закончит выполнение своей текущей задачи.

Набор опций DMA достаточно велик:

1. Загрузка в память 32-х битными или 16-ти битными блоками (16-ти битные по умолчанию).
2. Когда начинать DMA-загрузку (также называют "трансфером").
1. Прямо сейчас (DMA_TIMEING_IMMEDIATE)
2. При следующем HBlank'e (DMA_TIMEING_HBLANK)
3. При следующем VBlank'e (DMA_TIMEING_VBLANK)
4. Используется только для 3 канала, есть опция, позволяющая потоковое видео (согласно ThePernProject.Com).
3. Нужно ли увеличивать, понижать или держать фиксированным значение регистра адреса источника.
4. Нужно ли увеличивать, понижать или держать фиксированным значение регистра адреса назначения.

Много всего! Но пока что нам нужно только загружать данные, как-то: спрайты, палитры и картинки в режиме 3 на экран. Поэтому мы только начнём трансфер и укажем DMA увеличивать значение обоих регистров источника и назначения. Мы будем использовать трансфер 32-х битными блоками для спрайтов и палитр, но для картинок в режиме 3 мы будем использовать 16-ти битные блоки.

**Регистры DMA**

Есть 3 DMA-регистра для каждого канала (на самом деле 4, но 2 являются 16-ти битными):

• REG_DMAxSAD : Принимает адрес источника.
• REG_DMAxDAD : Принимает адрес назначения.
• REG_DMAxCNT : Принимает количество данных и опции, перечисленные в вышеприведённой секции.

Нам необходимо выполнить три шага, чтобы осуществить требуемый DMA-трансфер:

• Поместить адрес источника в REG_DMAxSAD.
• Поместить адрес назначения (то, где должны оказаться данные) REG_DMAxDAD.
• Поместить количество данных, трансфер которых будет осуществляться, и опции трансфера в REG_DMAxCNT.

Обратите внимание, что 'x' в названии регистра нужно будет заменить на номер DMA-канала. Теперь вы знаете, что нужно сделать.

**Загрузка картинки в режиме 3 с помощью DMA**

Пожалуйста, скачайте dma.h (находится в архиве `wasm_files.zip`: `files/0406/dma.h`).

Мы сейчас загрузим картинку в режиме 3 с помощью DMA. Так как мы используем DMA, нам не нужен будет цикл, а также потребуется всего лишь 2 регистра. Пожалуйста, сконвертируйте картинку под названием "pic.bmp", как это мы сделали в Дне 3, и назовите получившийся файл "pic.asm" (кавычки не нужны :) ). Теперь вместо того, чтобы менять старый исходник, мы просто сделаем новый. Вот его начало: (он включает всё, что нужно и устанавливает 3-ий режим экрана).

```
;;--- НАЧАЛО ИСХОДНОГО КОДА ---;;

@include screen.h   ; здесь всё, что нужно для экрана
@include dma.h      ; здесь всё, что нужно для DMA
b start             ; прыгаем через данные
@include pic.asm
start

ldr r1,=REG_DISPCNT  ; r1 указывает на регистра управления экраном
ldr r0,=(MODE_3|BG2_ENABLE)  ; в r0 - необходимые значения для установки режима 3 и фона 2
str r0,[r1]          ; устанавливаем регистр контроля, сохраняя значения в памяти

;;--- КОНЕЦ КОПИРОВАНИЯ ---;;
```

Метка в pic.asm будет называться как имя .bmp-файла плюс "Bitmap", то есть, если вы переименовали bmp в "pic.bmp" перед конверсией, у вас получится pic.asm, внутри которого метка будет называться picBitmap, которая и будет являться адресом источки. VRAM или по-другому "экран" будет адресом назначения. Мы начнём трансфер немедленно и будет осуществлять его 16-ти битными блоками. Мы будем использовать 3-ий канал DMA, так как это официльно канал общего назначения.

```
;;--- КОД ПРОДОЛЖАЕТСЯ ---;;

ldr r1,=picBitmap     ; \
ldr r0,=REG_DMA3SAD   ; - устанавливаем адрес источника в регистре адреса
                      ; источника 3-его канала
str r1,[r0]           ; /

ldr r1,=VRAM          ; \
ldr r0,=REG_DMA3DAD   ; - устанавливаем адрес назначения в регистре адреса
                      ; назначения 3-его канала

str r1,[r0]           ; /

ldr r1,=(38400|DMA_16NOW)  ; \
ldr r0,=REG_DMA3CNT       ;  - указываем регистру контроля, сколько блоков копировать
; на экране 240x160=38400 пикселей, мы перегоняем их 16-битными блоками, поэтому нам
; не нужно уполовинивать это число
str r1,[r0]      ; сразу начинается DMA-трансфер, который перегоняет 38400
                 ; 16-ти битных блоков данных, и программа не продолжит выполняться,
```

```
     ; пока эта операция не будет закончена. Но она довольно быстра.

infinite
b infinite

;;--- КОНЕЦ КОПИРОВАНИЯ ---;;
;;--- КОНЕЦ ИСХОДНОГО ФАЙЛА ---;;
```

Разве это не просто? Я всегда считал, что DMA - это что-то очень сложное, но оказалось, что это просто инициализация трёх регистров и всё! DMA - это круто. :)

**Загрузка данных из Дня 5 с помощью DMA**

Как вы помните, в Дне 5 у нас было 2 загрузочных цикла, которые были прекрасными кандидатами на замену DMA-трансфером. Один из них использовался для загрузки палитры, а другой - для загрузки спрайтов. В обоих случаях у нас есть всё, что нужно для DMA-трансфера: исходник (спрайт или палитра), назначение (память для символов или память для палитры) и сколько раз нужно обратиться к памяти. Для палитры это количество при трансфере 32-х битными блоками будет равным 128, так как в палитре 256 элементов. Для спрайта оно будет тем же самым, потому что.... ох, я не знаю! Мы делали это 128 раз в Дне 5, используя 32-х битные загрузку/сохранение, поэтому если мы будем придерживаться того же размера, то и обращаться к памяти придётся ровно столько же.

```
;;--- ЭТОТ КУСОК КОДА ИЗ ДНЯ 5 НАДО БУДЕТ ЗАМЕНИТЬ ---;;

ldr r1,=OBJPAL     ; r1 указывает на палитку спрайта (0x500200)
ldr r2,=128            ; 128 - сколько раз обращаемся к палитре
ldr r3,=pallete        ; палитра задана в sprit.asm (полученый конвертацией)

palloop:        ; Этот цикл - типичный пример загрузки данных с помощью цикла
ldr r7,[r3]+4!  ;  4 байта из палитры помещается в r. каждый раз не забываем
                ; увеличить указатель на 4
subs r2,r2,1    ;  здесь количество обращений уменьшается на 1
bne palloop     ; проверяем на равно

; если r2 = 0   (r2==0 для тех, кто привыки писать на C). Условие выполняется
; только, если значение не равно 0.

;;--- КОНЕЦ КУСКА КОДА ---;;
```

Заменить вышеуказанное на следующее:

```
;;--- НАЧАЛО КОДА ---;;

ldr r1,=pallete       ; \
ldr r2,=REG_DMA3SAD   ;  - адрес источника в DMA-регистре источника
str r1,[r2]           ; /

ldr r1,=OBJPAL        ; \
ldr r2,=REG_DMA3DAD   ;  - адрес назначения в DMA-регистре назначения
str r1,[r2]           ; /

ldr r1,=(128|DMA_32NOW) ; \
ldr r2,=REG_DMA3CNT     ;  - количество данных для трансфера
str r1,[r2]             ; какого размера блоки используются (мы используем 32-х
                        ; битные)

;;--- КОНЕЦ КОДА ---;;
```

Ок, теперь замените следующее:

```
;;--- ЭТО НАДО БУДЕТ ЗАМЕНИТЬ ---;;

ldr r1,=CHARMEM   ; смотрите описание ниже
ldr r2,=128
ldr r3,=obj0

sprloop:
```

```
ldr r7,[r3]+4!
str r7,[r1]+4!
subs r2,r2,1
bne sprloop

;;--- КОНЕЦ ---;;
```

Обратите внимание, что я оставил комментарии в коде из Дня 5 просто для того, чтобы вам было легче его найти.

Замените вышеприведённое на следующее:

```
;;--- НАЧАЛО КОДА ---;;

ldr r1,=obj0       ; \
ldr r2,=REG_DMA3SAD   ;  - адрес источника в DMA-регистре источника
str r1,[r2]           ; /

ldr r1,=CHARMEM        ; \
ldr r2,=REG_DMA3DAD   ;  - адрес назначения в DMA-регистре назначения
str r1,[r2]           ; /

ldr r1,=(128|DMA_32NOW) ; \
ldr r2,=REG_DMA3CNT     ;  - количество данных для трансфера

str r1,[r2]             ; какого размера блоки используются (мы используем 32-х
                        ; битные)

;;--- КОНЕЦ КОДА ---;;
```

Обратите внимание, что этот практически тот же самый код, не считая изменившихся адресов источника и назначения.

Если вы скомпилируете и запустите код из дня 5 с данными исправлениями, то он будет делать ТО ЖЕ САМОЕ, разве что загрузится быстрее. Вы можете заметить, что полученный бинарник меньше размером, по сравнению с тем, где DMA не использовалось.

# GBA ASM - День 9: Простой бэкграунд

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-07-16

## Что мы собираемся сделать сегодня

Во-первых и главным образом вам потребуется понять DMA, так как теперь я буду делать всю загрузку с помощью DMA. Во-вторых, вам потребуется следующее:

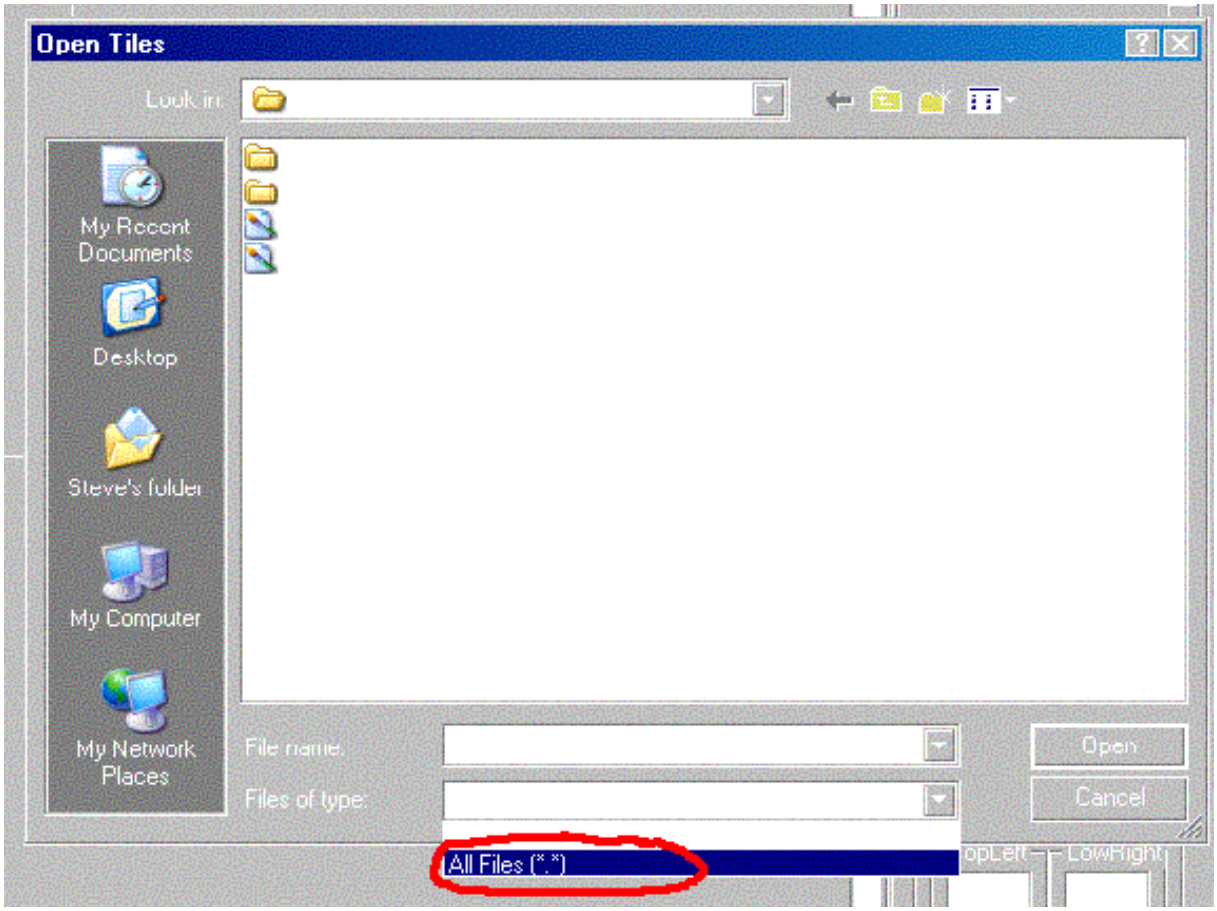
Инструменты, необходимые для бэкграундов:

1. GBA Map Editor Beta 4 (находится в архиве `wasm_files.zip`: `files/0407/GBAMeditBeta4.zip`) - Он багованный, но работает, если вы попробуете несколько раз :).
2. Gifs2Sprites (находится в архиве `wasm_files.zip`: `files/0407/gifs2sprites.zip`) - у вас уже должна быть эта утилита, так как мы использовали её для конвертации тайлов.
3. Картинка для использования в качестве тайлов - вы можете использовать спрайт и Дня 5. Вы увидите позже, как это сделал я.
4. CtoASM.exe (находится в архиве `wasm_files.zip`: `files/0407/ctoasm.zip`) - все наши инструменты выводят конечный результат в формате C, используйте мою программу, что получить ассемблерный код.
5. Вот и всё, что вам нужно для бэкграундов - много рисования, но мало программирования.

Создание бэкграундов на самом деле не слишком сложно, как это может показаться, прочитав соответствующий туториал на ThePernProject. Просто 3 загрузки и установка 2-х регистров и вуаля - мгновенный бэкграунд. В следующей секции мы подготовим тайлсет.

## Подготовка тайлсета

Для подготовки тайлсета (т.е. тайлов, которые образуют карту/бэкграунд) используйте .gif-файл из Дня-5 или можете нарисовать свой собственный тайлсет в MS Paint. Запустите GBA Map Editor. В окне "Tool" кликните по меню "Tiles" и выберите "Load Tiles". У вас должен открыться следующее диалоговое окно.



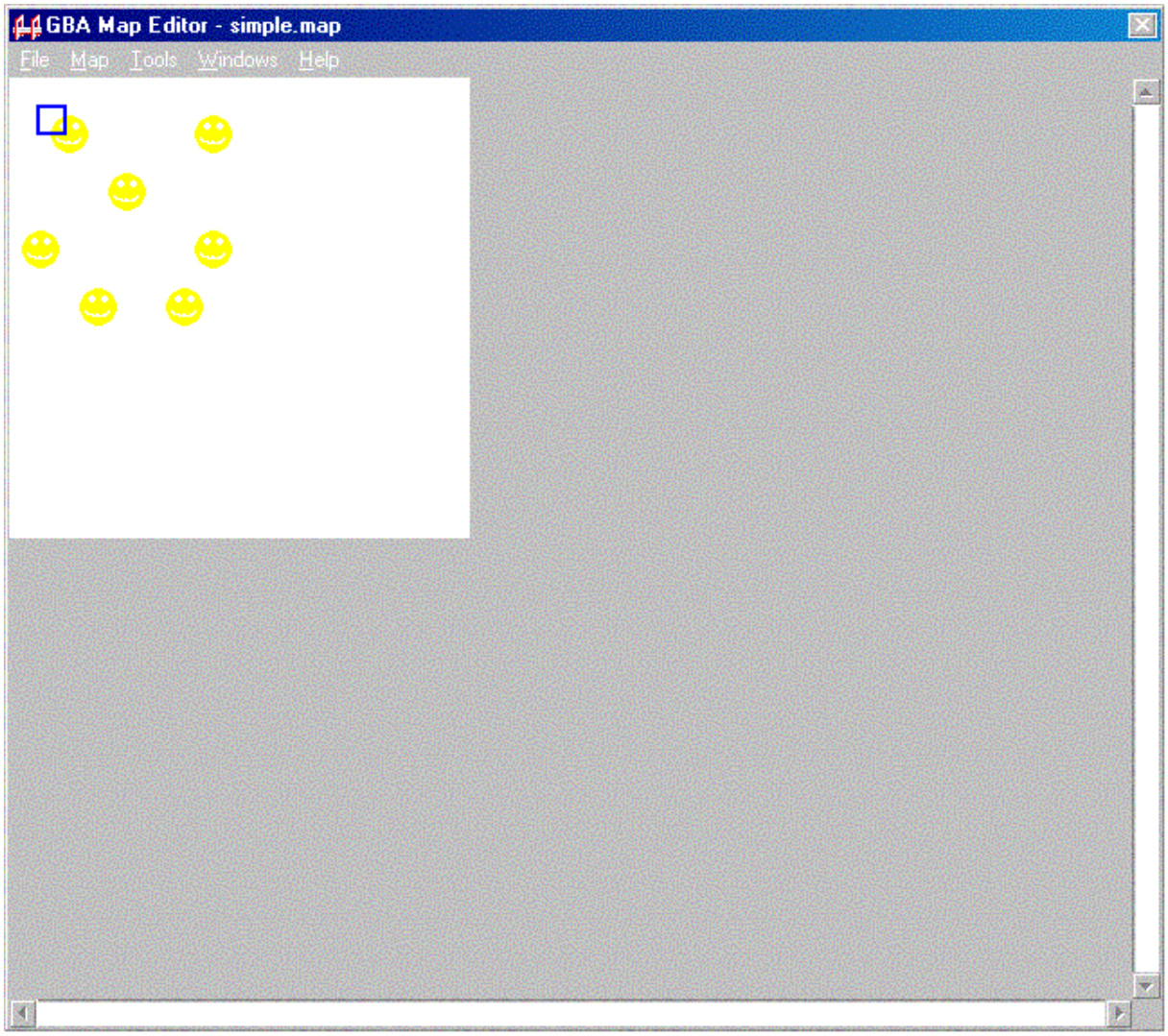

Поскольку у нас gif-файл, вам придётся заменить "Files of type" на "All Files" (я пометил это на картинке). Сделайте это и откройте gif. Теперь картинка должна отображаться у вас в окне "Tool" в прокручивающемся фрейме, который называется "Choose Tile". Вы заметите, что там есть маленький красный квадрат. Кликните в любом месте картинки и этот квадрат переместится на этот тайл. Вы всегда можете выяснить, какой тайл используется в настоящий момент, посмотрев на окно "Tool" под "Selected Tile".

Теперь сделаем карту. В окне большего размера откройте меню "File" и кликните "New". Появится окно "Map Properties". Напечатайте в поле "Map Name" слово "simple", потому что эта карта будет очень проста (от англ. simple - "простой", прим. пер.), так как сейчас нам нужно лишь научиться делать карты и только. Убедитесь, что выбранный размер карты равен 256x256 и нажмите кнопку "OK" внизу окна.

В следующей секции рисуем карту!

## Создание карты

Вы можете заметить опцию "Block Select" и решить, чтобы было бы неплохо её испльзовать. Как я сказал раньше, GBA Map Editor beta 4 - это очень багованная программа. Когда я использовал эту опцию, то в конце концов карта была обрезана. Этого не случится, если вместо опции "Block Select" вы пойдёте в меню "Tools" большого окна, наведёте мышью на "Tile Size" и кликните "16x16". Теперь красный квадрат в 4 раза больше, что не является полноценной заменой опции "Block Select", но на данный момент сгодится. Нарисуйте карту в большом чёрном квадрате/прямоугольнике/чём-бы-это-ни-было в большом окне. Это не должно быть чем то очень красивым, но мне понадобилось добрых пять минут, чтобы сделать из GIF'a из Дня 5 улыбающееся лицо. Вот как выглядит моё большое окно:




Давайте сохраним карту на тот случай, что редактор сглючит и не экспортирует нашу карту корректно. Откройте меню 'File" и кликните "Save". Сохраните под именем "simple.map". Хорошо.

Теперь мы экспортируем нашу карту в C. Идите в меню "File" в большом окне и поместите мышь над пунктом "Tools" и выберите "Export to C". Появится диалоговое окно сохранения файла. Сохраните под именем "simple.c". Теперь, перед тем как закрывать редактор карт, посмотрите на размер simple.c. Если размер меньше 5kb, то редактор сглючило (это происходит в половине случаев). Если это случилось, то либо попытайте сохранить снова и снова, либо закройте и запустите редактор карт заново - и так, пока не получите то, что вам нужно.

Другой путь определить корректность экспорта - это открыть simple.c. Вы увидите, что на первой строке говорится что-то вроде "32*32". Это значит, что должно быть 32 ряда и 32 колонки чисел. Просто посчитайте количество линий. Зачастую там будет 17, что неправильно. Используйте метод, приведённый выше, чтобы добиться нужного результата.

## Ещё одна конвертация Gif в C

В этот раз мы должны использовать 256 цветов. Предположив, что ваш файл называется "sprit.gif", вот команда, которую вы должны напечатать:

Gifs2sprites 256 sprit.h sprit.gif

Обратите внимание, что имя создаваемого заголовочного C-файла задаётся первым, а уже ПОТОМ имя gif(ов). На выходе у вас должен получиться "sprit.h".

Итак, теперь у вас есть следующее:

1. sprit.gif - картинка, которую вы использовали в качестве исходной.
2. siple.map - карта, которую можно в случае чего отредактировать в редакторе.
3. simple.h - C-код для чисел тайлов в карте.
4. sprit.h - C-код для изображения sprit, которое мы использовали для тайлов.

Теперь нам нужно сконвертировать C-код в ассемблер!

## Конвертация в ассемблер

Убедитесь, что у вас находятся 2 .h-файла и CtoASM.exe в одной директории, и запустите CtoASM.exe. Напечатайте "simple.h" в верхнем боле и "simple.asm" в нижнем. Нажмите кнопку "GO". Вы должны получить файл "simple.asm". Теперь закройте CtoASM. Вы его точно закрыли? (да-да, закрыли - прим. пер.) Теперь запустите его снова. Я попросил вас это сделать, потому что тестировал программу по конвертации за запуск, и не уверен насчёт её стабильности, если конвертировать несколько файлов подряд. Теперь напечатайте "sprit.h" и "sprit.asm" в соответствующих полях. Нажмите "GO". Если всё прошло успешно, то теперь у вас есть файл "sprit.asm".

Я надеюсь, что моя Windows-программа (я люблю XP и до сих пор не видел, чтобы она упала, пока :) ) пригодиться вам.

## Регистры, о которых нам нужно знать

Есть 2 регистра о которых нам действительно нужно знать. Я начну с самого простого под названием REG_DISPCNT. Да, этот тот же регистр, что контролирует режим экрана. С его помощью нам нужно включить режим бэкграунда №2. Если вы заметили, что мы делали это всегда и до этого, то не волнуйтесь, поэтому я и сказал, что это будет просто! Другое, что вам нужно знать - это то, что сегодня мы будем использовать режим экрана №1, а не №3. Таким образом, в регистр REG_DISPCNT мы загрузим значение (MODE_1|BG2_ENABLE).

Другой регистр REG_BG2CNT чуть более сложен в понимании. Во-первых, если вы хотите использовать другой бэкграунд, включите его в регистре режима экрана, а затем замените '2' в

'REG_BG2CNT' на номер бэкграунда. REG_BG2CNT требует много информации, но для наших целей (простой бэкграунд) нам нужно загрузить в него всего лишь 4 вида данных.

Необходимая информация для REG_BG2ENABLE при 256 цветах 256x256 переменном (rotation) бэкграунде.

1. Size - будет равен ROTBG_SIZE_256x256.
2. Цвет - будет равен BG_COLOR_256. Переменные бэкграунды не могут использовать 16 цветов.
3. Базовый блок экрана (Screen Base Block), далее SBB - куда вы поместите карту. Посмотрите на туториал о бэкграунда на ThePernProject, чтобы узнать больше информации об экране и базовых блоках символов (Char Base Blocks), чем даю здесь я.
4. Базовый блок символов (Char Base Block), далее CBB - куда вы поместите сами битмапы тайлов (это и правда перевод с английского? - прим. пер.)

Есть 4 CBB и 32 SBB, но они пересекаются друг с другом, поэтом вам нужно поместить карту в блок, не находящийса в середине выбранного CBB. Все эти блоки в VRAM, поэтому мы не можем использовать такой битмап-режим как №3, потоу что иначе у нас на экране будет мусор.

Начинаем кодить!

Давайте же начнём кодить! Как обычно, я буду приводить код блоками, а потом его пояснять. И как обычно, мои комментарии будут приведены в вероятно большем количестве, чем требуется.

Теперь скачайте заголовочный файл бэкграундов backg.h (находится в архиве `wasm_files.zip`: `files/0407/backg.h`).

Первый блок:

```
;;--- КОД НАЧИНАЕТСЯ ---;;
@include screen.h  ; всякие подсобные данные
@include dma.h     ; нужно для использования DMA
@include backg.h   ; содержатся необходимые данные для бэкграудов

b start   ; перепрыгиваем через бэкграунды и тайлы
@include simple.asm    ; наша карта, имеет метку objB0
@include sprit.asm     ; наш тайл с меткой obj
start    ; метка

;устанавливаем режим экрана и включаем бэкграунд 2
ldr r1,=REG_DISPCNT
ldr r2,=(MODE_1|BG2_ENABLE)
str r2,[r1]       ; обратите внимание, что инструкция str сохраняет
                  ; первый операнд во втором в отличии от инструкции ldr,
                  ; которая помещает второй в первый

; помещаем в регистр управления бэкграундами необходимую информацию
ldr r1,=REG_BG2CNT
ldr r2,=(ROTBG_SIZE_256x256|BG_COLOR_256|(28<<SCREEN_SHIFT))
str r2,[r1]
;;--- СТОП КОПИРОВАНИЕ ---;;
```

Вы можете спросить, что за (28<<SCREEN_SHIFT) был использован в последней инструкции ldr. SCREEN_SHIFT задаётся в backg.h, и мы сдвигаем 28 на значение SCREEN_SHIFT, чтобы поместить это число в место в регистре, где содержится номер SBB. Как вы можете видеть, мы используем 28-ой SBB и 0-й CBB. Мы не обязаны использовать именно 0-й, но это дефолтное значение, поэтому зачем усложнять себе жизнь? Мне кажется, что SBB №28 далековато располагается от CBB №0, но пока они не пересекаются друг с другом, это не должно нас волновать. Вот загрузка карты и тайла:

```
;;--- КОД ПРОДОЛЖАЕТСЯ ---;;
;теперь загружаем тайл с помощью DMA
ldr r1,=REG_DMA3SAD  ; \
ldr r2,=obj     ; - Адрес источника - obj, заданный в sprit.asm
str r2,[r1]     ; /
```

```
ldr r1,=REG_DMA3DAD    ; \
LoadCharBlockAddr r2, 0   ; - Подсобный макрос, загружает в r2 адрес СВВ #0,
str r2,[r1]           ; являющийся нашим адресом назначения для этого
                      ; DMA-трансфера

ldr r1,=REG_DMA3CNT          ; \
ldr r2,=(8192|DMA_32NOW)   ; - перемещаем 8192 32-х битных блоков тайла в
                           ; СВВ #0
str r2,[r1]               ; /

;------------------------------divider to make reading easier

ldr r1,=REG_DMA3SAD  ; \
ldr r2,=objB0     ; Адрес источника - objB0, заданный в simple.asm
str r2,[r1]     ; /

ldr r1,=REG_DMA3DAD    ; \
LoadScreenBlockAddr r2, 28   ; - подсобный макрос, загружает адрес
                      ; SBB #28, являющийся адресом назнанчения для этого
str r2,[r1]           ; DMA-трансфера

ldr r1,=REG_DMA3CNT          ; \
ldr r2,=(512|DMA_32NOW)   ; - перемещаем 512 32-х битных блоков карты в
str r2,[r1]               ; SBB #28
;;--- STOP COPYING ---;;
```

Обратите внимение, что комментарии начинаются с ';' и игнорируются Goldroad'ом. Не помню, говорил ли я вам об этом раньше, поэтому скажу здесь :).

Теперь я хотел бы объяснить, как я определил, сколько блоко необходимо переместить. В первый раз это было изображение 32x32. Это 1024, умножаем на 32, что является количество байтов в тайле. Получается 32768 блоков по 8 бит. Делим на 4 и получаем 8192 32-х битных блоков, которые необходимо переместить. Я не помню, как я получить 512 во втором случае, но это правильное значение.

Вы помните, что я говорил о трёх загрузках? Третья - это палитра, она загружается в VPAL, которая находится в память до спрайтовой палитры (0x5000000). Таким образом, загрузка палитры и бесконечный цикл будут являться последним блоком кода в этом Дне. Итак:

```
;;--- CODE CONTINUE ---;;
ldr r1,=REG_DMA3SAD  ; \
ldr r2,=pallete     ; - Адрес источника - pallete, задано в in sprit.asm
str r2,[r1]     ; /

ldr r1,=REG_DMA3DAD    ; \
ldr r2,=VPAL   ; - Адрес назначения VPAL (палитра для бэкграунда и режима
               ; экрана 4)
str r2,[r1]            ; /

ldr r1,=REG_DMA3CNT       ; \
ldr r2,=(128|DMA_32NOW)   ; - перемещаем 128 32-х битных блоков кода в VPAL.
str r2,[r1]               ; / Думаю, это будет верно для всех палитр, так
                          ; как в них 256 16-ти битных элементов

infin   ;\
b infin ; - бесконечный цикл
;;--- КОНЕЦ КОДА ---;;
```

Вот и всё, компилируйте и запускайте свой код!

## Обзор этого дня

Не знаю, что и сказать, кроме того, что у меня заняло два выходных написать этот День в отличии от предыдущих, на которые требовалось по одному. Надеюсь, он вам понравился!

# GBA ASM - День 10: Прерывания и таймеры

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-07-17

**Об этом дне** Сегодня мы должны изучить прерывания и таймеры. Я не уверен, что у прерываний есть иное полезное применение кроме процедуры HBlank или VBlank, поэтому именно это и будет продемнстрировано в коде. Таймеры, по крайней мере насколько могу судить я, имеют несколько применений, включая.. гм.. тайминг :). Сходу я могу назвать одну вещь, которая требует применение DMA, таймеров и прерываний одновременно - это вывод звука (например, проигрывание wav-файла - мы сделаем это в другой раз). **О прерываниях**

Прерывание - это точно то, что вы думаете (возможно :) ). Они прерывают работу маленького ARM7-процессора GBA и заставляют его начать выполнение другой части программы, чтобы затем вернуться обратно, как только всё, что нужно будет сделано. Вы можете решить, что это уничтожить значения, находящиеся в регистрах, но на самом деле этого не происходит, так как сначала управление переходит к BIOS'у, который сохраняет значения регистров, перед тем как передать управление обработчику прерывания. После того, как обработчик отработал, значения регистров восстанавливаются (обратите внимание, что под здесь под регистрами я подразумеваю регистры процессора - r0 и подобные).

Для указания GBA, что нам требуется обрабатывать прерывание, мы должны сделать несколько вещей.

Шаги для обработки прерывания:

1. Включить прерывания в REG_IME. Я написал специальный макрос для этого, который находится в interrupt.h.
2. В регистре REG_IE необходимо указать, какое прерывание мы хотим обрабатывать.
3. В REG_INTADDR задаётся адрес обработчика прерывания.
4. Включить прерывания снова с помощью регистра, который зависит от того, какое прерывание мы собираемся использовать.
5. Вот и всё, теперь прерывание будет происходить, и GBA автоматически будет передавать туда управление!

**Давайте напишем прерывание**

Скачайте специальный заголовочный файл interrupt.h (находится в архиве `wasm_files.zip`: `files/0408/interrupt.h`).

Почему бы вам не сконвертировать изображение в ассемблер, как мы делали это раньше. Не забудьте заменить "DCW" на "@DCW". Откройте его и убедитесь, что метка внутри - picBitmap.

```
;;--- НАЧАЛО КОДА ---;;
@include screen.h  ; нужно почти всегда
@include interrupt.h ; для прерываний
@include dma.h  ; макрос DrawMode3Pic использует DMA

b start
@include pic.asm  ; подключаем картинку - не забудьте перепрыгнуть через неё.
start

; устанавливаем режим экрана и включаем режим бэкграунда 2
ldr r1,=REG_DISPCNT
ldr r2,=(MODE_3|BG2_ENABLE)
str r2,[r1]

Interrupts_Enable  ; мой макрос, который включает в (REG_IME)
                   ; поддержку прерываний
```

```
ldr r1,=REG_IE
ldr r2,=INT_VBLANK  ; эти три строки указывают, что мы будет обрабатывать
str r2,[r1]         ; прерывание VBlank

addr r1,inthandler   ; а эти три загружают в REG_INTADDR адрес
ldr r2,=REG_INTADDR  ; обработчика прерывания. ADDR - это особенность
str r1,[r2]   ; ассемблера, помещающий адрес метки в регистр CPU,
; который в дальнейшем мы помещаем в регистр памяти REG_INTADDR
;;--- STOP COPYING ---;;
```

Следующий регистр памяти, который мы должны инициализировать зависит от того, какое прерывание мы хотим обрабатывать. Нам нужно прерывание VBlank, поэтому необходимо указать экрану его генерировать. Мы делаем это с помощью регистра REG_DISPSTAT.

```
;;--- CODE CONTINUE ---;;
ldr r1,=REG_DISPSTAT
ldr r2,=STAT_VBLANK
str r2,[r1]

infin
b infin
;;--- СТОП КОПИРОВАНИЕ ---;;
```

Это конец основной части программы, но мы ещё НЕ закончили. На необходимо написать обработчик прерывания. Когда вы посмотрите и потестируете сам код, вы поймёте больше, чем я могу объяснить так, поэтому продолжнаем:

```
;;--- КОД ПРОДОЛЖАЕТСЯ ---;;
inthandler   ; метка начала обработчика

DrawMode3Pic picBitmap    ; передаём метку картинки макросу
; заметье, что он использует DMA, поэтому нам и понадобился dma.h

ldr r9,=INT_VBLANK      ; обратите внимание, что нам нужно передать регистр
                        ; макросу ReturnFromHandler
ReturnFromHandler r9   ; говорим GBA, что прерывание обрабатывается
; Также отметьте, что вам необходимо использовать хотя бы r4 для этого макроса,
; так как r0-r3 используются внутри него.

bx lr ; возвращение из прерывания, это ветвчение к адресу в r13, так что не
      ; не обращайте на это большого внимания. Вероятно, имеет смысл
      ; поместить эту строку в ReturnFromHandler
;;--- КОНЕЦ КОДА ---;;
```

Теперь, когда вы запустите программу отобразится картинка. Ничего особенного. Но когда мы изучим как использовать рабочую память, вы увидите, насколько важными могут быть прерывания и в особенности VBlank. Есть несколько вещей, которые лучше делать во время VBlank, поэтому наличие соответствующего прерывания очень полезно. Обратите внимание, что прерывания могут создавать другие вещи, мы лишь прикоснулись к самой поверхности. Теперь переходит к таймерам!

**О таймерах**

Я думаю, что слово таймеры может ввести в небольшое заблужение, так как они скорее являются счётчиками, чем таймерами. Тем не менее, они дают возможность следить за верменим. Я не уверен, что следует конкретно сделать в приводимом мной коде, потому что единственное применение, которое пришло мне в голову - это вывод звука, а его мы будем изучать позже. Пожалуйста, помните, что даже хотя мы использовали Timer #3, всего их 4 (0-3). При необходимости замените 3 на требуемый номер в соответствующих макросах.

Теперь вам следует скачать timer.h.

**Программирование таймеров**

Прежде чем мы начнём, пара замечаний. Таймеры могут находится на 1 из 4 частот, что влияет на то, как быстро они считают. Таймеры считают от 0 до 655535. Переход от 65535 к 0 называется

переполнением, и таймер можно заставить генерировать соответствующее прерывание. Мы не будем этого делать, но знайте, что это возможно.

Частоты таймеров:

1. Системная частота - 15 оборотов.
2. 64-х кратная системная частота
3. 256-ти кратная системная частота (почти секунда, ох).
4. 1024-х системая частота.

Полагаю, что 3-я наиболее подходит. Нам осталось написать макрос WaitSeconds, что не должно оказаться слишком сложным. Я уже написал его, и он находится в timer.h, но рекоменду пройтись со мной по его созданию :).

Чтобы запустить таймер, вам потребуется включить его и загрузить в REG_TMxCNT частоту. Включение таймера происходит с помощью макроса EnableTMx (x=№ таймера):

```
ldr r10,=TIME_FREQUENCY_256
EnableTM3 r10
```

Обратите внимание, что мы не можем передать значение, определённое директивой define в силу ограничения Goldroad. Поэтому вам нужно загрузить его в регистр CPU, и передать уже его.

Теперь мы можем начать писать макрос WaitSeconds. Ок, это тот случай, когда нам нужен цикл и нельзя использовать DMA. Поехали:

ПОЖАЛУЙСТА, ПОМОГИТЕ! Я НЕ МОГУ ЗАСТАВИТЬ ЭТО РАБОТАТЬ! (опа - прим. пер.)

```
;;--- НАЧАЛО КОДА ---;;
@macro WaitSeconds3 Number, arglabel

ldr r2,=Number  ; Number будет заменён на переданное при вызове макроса число
arglabel   ; вам нужно передать имя, которое будет использовано в качестве
           ; метки для цикла
;;--- КОД ПРИОСТАНОВЛЕН. ПОЯСНЕНИЕ ---;;
```

Здесь начинается макрос. Мы загружаем в r2 количество секунд и создаём метку для основного цикла.

```
;;--- ПРОДОЛЖЕНИЕ КОДА ---;;
ldr r1,=REG_TM3D   ; сравниваем содержимое таймера с 0,
ldrh r3,[r1]       ; и если оно не равно нулю, то
ldr r4,=0          ; то переходим обратно на arglabel
cmp r3,r4
bne arglabel
;;--- КОД ПРИОСТАНОВЛЕН. ПОЯСНЕНИЕ ---;;
```

Здесь мы проверяем, произошло ли переполение. Если нет, то переходим обратно к arglabel. Заметьте, что поскольку мы включили таймер на частоте в 256 раз большей системной, то нам нужно просто замерить столько переполнений, сколько секунд нам нужно подождать. И закончим.

```
;;--- CODE CONTINUE ---;;
subs r2,r2,#1
bne arglabel

@endm
;;--- CODE STOP ---;;
```

Это было весело. Я надеюсь, что у вас ещё осталась та картинка, которую вы сконвертировали для первой части этого Дня, потому что сейчас мы напишем маленькую программу, которая будет ждать 30 секунд перед выводом картинки. Давайте начнём:

```
;;--- НАЧАЛО КОДА ---;;
@include screen.h  ; нам это нужно
@include dma.h   ; это нужно макросу, отрисовывающему картинку
@include timer.h ; нужно для таймеров
```

```
b start
@include pic.asm  ; подключаем данные с картинкой и перепрыгиваем через них
start

ldr r1,=REG_DISPCNT         ; устанавливаем режим
ldr r2,=(MODE_3|BG2_ENABLE)  ; экрана
str r2,[r1]                  ; №3

ldr r10,=TIME_FREQUENCY_256
EnableTM0 r10

WaitSeconds0 30, waitlabel1  ; ждём 30 секунд и используем waitlabel1 в
                             ; качестве метки внутри макроса

DrawMode3Pic meBitmap  ; отрисовываем картинку на экране

infin
b infin  ; бесконечный цикл
;;--- КОНЕЦ КОДА ---;;
```

Я знаю, что здесь немного кода, но за ним стоит очень немало всего!

**Обзор этого Дня**

Я надеюсь, что вам понравился этот День. Он отнял у меня немало сил. Извините, что код таймера работает не очень хорошо :(.

# GBA ASM - День 11: Переменные и рабочая память

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-07-22

**Почему?** Сначала я собирался в этот День писать о проигрывании wav-файла. Затем я понял, что не рассказал об одной вещи, которую всегда использовал, но никогда не говорил о ней ничего конкретного, а именно - загадочная область памяти, располагающая в 0x3000000. Мы немного использовали это пространство для некоторых (32-х битных) переменных, т.е. не совсем переменных, но это так близко к переменным как может быть для нас, программистов на ассемблере. **Как это делается**

Начнём с начала. Если вы программировали на C или QBasic или чём-угодно, вы знаете, что у переменных есть тип. Ок, сейчас я развею это заблуждение. Наши переменные всегда 32-х битные (на самом деле не всегда, они могут быть размером с байт или 16-ти битными, об этом позже). Да, именно так, никаких типов, считайте, что это почти тоже самое, что регистр CPU (gbaguy весьма волен в допущениях - прим.пер.), не считая того, что требуется 2 строки кода, чтобы получить к ним доступ.

Чтобы объявить переменную, нужно сделать следующее:

```
variable_name @DCD initial_value
```

Соответственно, если я хочу объявить переменную под названием num1 с начальным значением 0x0000F6A2, то делаю так:

```
num1 @DCD 0x0000F6A2
```

Мило, правда?

**Несколько проблем**

Если вы действительно искусны, вы можете заметить, что этот способ объявления переменные порождает несколько проблема.

• Переменная теперь находится в ROM.
• Объявление переменной в ROM заставить CPU выполнить байты, находящиеся в ней.

Мне нужно признаться кое в чём. Я уже говорил об этом, но повторю ещё раз: наши переменные не большее метки адреса в памяти, и Goldroad, вероятно, должен назначить им адреса. Как бы то ни было, избавиться от этих проблем также легко, как добавить две строки с директивами Goldroad'a (которые начинаются с символа '@'). Вот эти строки:

```
@textarea 0x3000000 и @endarea.
```

Т.о., получилось следующее:

```
@textarea 0x3000000
num1 @DCD 0
@endarea
```

Обратите внимание, что в этот раз я инициализовал область памяти (нашу переменную) нулём. Я люблю помещать все свои переменные в один блок @textarea, так как если вы используете другой, то

```
@textarea 0x3000000
num2 @DCD 0x9
@endarea
```

обратите внимание, что теперь у вас есть две переменные, которые указывают на одну и ту же область в памяти. Измените одну и попробуйте получить доступ к другой и вы увидите, что её

значение так же изменилось, они обе - это алиасы одной и той же области памяти. Поэтому помещайте все ваши "переменные" в один блок @textarea@!

**Установка значения WRAM**

Теперь, когда мы их объявили, нам следует установить их значение. Вот, что нужно сделать для этого, предположив, что в r1 находится нужное нам нужно нам значение.

```
ldr r0,=num1     ; помещаем адрес num1 в r0.
str r1,[r0]    ; помещаем содержимое r1 в память по адресу, хранящемуся в r0.
; помните, что наши переменные - всего лишь алиасы для адресов памяти.
```

Вот и всё. Довольно просто. Получение значения будет немного сложнее.

**Получение значения переменной в WRAM**

Ок, я просто сделаю это и прокомментирую:

```
ldr r0,=num1    ; помещаем адрес num1 в r0
ldr r1,[r0]     ; загружаем в r1 содержимое памяти по адресу, хранящемуся в r0
; там хранился адрес нашей переменной, а значит теперь в r1 - её значение.
```

Обратите внимание, что последняя строка кода - это прямая противоположность инструкции str, так как если бы мы написали str r1, [r0], то мы бы поместили r1 в память по адресу в r0, вместо того, чтобы поместить в r1 содержимое памяти по адресу в r0. Как вы можете видеть, это обратные друг другу инструкции.

Это всё, что вам нужно знать. Не слишком сложно, но важно.

**Обзор этого Дня**

Сожалею, если вы ожидали, что здесь будет какой-нибудь код, выводящий что-нибудь крутое. Но мне кажется, что использование рабочей памяти довольно важно, особенно в большой игре, или где-то ещё, где даже 13 регистров не хватает.

Вот и всё на сегодня, надеюсь, вам понравилось.

# GBA ASM - День 12: Инструкция BL

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-08-06

**Что мы делаем сегодня?** Сегодня мы рассмотрим подпрограммы (или функции), используя инструкцию BL. Странно, что я никогда не рассказывал о ней, ведь она так проста... **Инструкция BL**

Инструкция BL расшифровывается как Branch with Link (Ветвление со связью). Это означает, что она делает то же, что и обычная инструкция B, не считая того, что помещает в r14 определённое значение. r14 также известен как lr. Вы можете использовать оба варианта, т.к. они означают одно и то же, и Goldroad'y не важно, что вы выберите. Если я никогда не говорил этого раньше, то r15 также известен как pc.

С регистр r15 (или pc), как вы уже должны знать из Дня 2, лучше не связываться лишний раз, т.к. он содержит адрес следующей инструкции, которая должна быть выполнена. Если вы меняете значение в нём, то фактически вы совершаете переход на какое-то место в памяти. Позже мы сделаем это, чтобы вернуться из одной из наших "функций".

**Это значение в r14...**

Я сказал ранее, что инструкция BL кладёт некое значение в 14. Это значение - адрес возврата. Если говорит точнее, это адрес инструкции, идущей сразу после BL. В отличии от x86-процессоров, инструкции RET здесь нет. Мы должны скопировать адрес возврата из r14 в r15. Вот простой способ сделать это.

```
mov r15,r14
```

Это и есть наша инструкция возврата. Вы также можете использовать LDR, но мне кажется, я где-то читал, что MOV быстрее, когда дело касается копирование содержимого из одного регистра CPU в другой.

Как только эта строка будет выполнена, CPU вернётся к инструкции, которая шла после инструкции BL, которую вы использовали для перехода к функции.

**(Очень) Маленький пример**

Хорошо, теперь приведу действительно маленький, бесполезный пример, что это работает.

```
;;--- НАЧАЛО КОДА ---;;
@include screen.h ; screen.h с необходимыми константантами,
                  ; требуемые даже в такой маленькой программе.

ldr r1,=REG_DISPCNT
ldr r2,=(MODE_3|BG2_ENABLE) ; 3 строки для установки режима экрана
str r2,[r1]

bl DrawDot   ; переход на процедуру DrawDot

infin     ; после возврата начнётся бесконечный цикл
b infin
;;--- СТОП КОПИРОВАНИЕ ---;;
```

Вот процедура DrawDot:

```
;;--- НАЧАЛО КОДА ---;;
DrawDot  ; метка под названием DrawDot

ldr r1,=VRAM+0x2410 ; эти три линии отрисовывают
ldr r2,=0x45678912  ; 2 пикселя на экране.
str r2,[r1]

mov r15,r14 ; возвращение в основную программу
```

```
;;--- КОНЕЦ КОДА ---;;
```

Обратите внимание, что вам не должны ничего делать с r14 внутри, иначе вы потеряете адрес возврата. Также никогда ничего не делайте с r15, кроме вышеуказанного случая возврата из процедуры.

**Обзор этого дня**

Не знаю, почему не объяснил вам это раньше, ведь это так легко. Сожалею, если вы ожидале чего-то более "крутого". Больше здесь не очень рассказывать. Хорошего программирования.

# GBA ASM - День 13: Эффекты со спрайтами

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-08-10

**Какие именно эффекты со спрайтами?** Мы рассмотрим следующие: вертикальный и горизонтальный кувырок (flip), мозаику, вращение и масштабирование. Тем не менее, относительно вращения, я не буду рассказывать о SIN() и COS(), поэтому мы только обсудим что и где поместить в данных для поворота/масштабирования, а остальное вам придётся додумать самим. Я мог бы обсудить создание таблицы значений для массива вращения, но это не слишком вписывается в тему данного Дня. **Флаги кувырка**

Это настолько просто, что я даже не буду давать полный пример. Флаг вертикального кувырка - это бит #12 в атрибуте 0. Необходимые константы уже содержатся в sprites.h (находится в архиве `wasm_files.zip`: `files/0416/sprites.h`):

• VERTICAL_FLIP
• HORIZONTAL_FLIP

Флаг горизонтального кувырка - это бит #13 в атрибуте 0. Чтобы включить флаг кувырка, вам нужно поместить содержимое первых 32-х битов OAM-элемента спрайта в регистр и поEORить (то же самое, что и поXORить) его с VERTICAL_FLIP или HORIZONTAL_FLIP. Ок, вот немного кода:

```
ldr r2,=OAM
ldr r3,[r2] ; загружаем r3 из памяти
; вышеприведённое загрузить r3 атрибутом 0 и атрибутом 1, т.к. регистр
; 32-х битный, а OAM-атрибуты 16-ти битные.
eor r3,r3,VERTICAL_FLIP   ; XORим r3 с VERTICAL_FLIP и помещаем значение в r3.
; EOR и XOR - это одно и то же, главное помните, что инструкция называется EOR.
str r3,[r2]  ; сохраняем модифицированные атрибуты обратно в OAM.
```

Этот код включает флаг вертикального кувырка. Я думаю, вы поняли идею, и сможете включить флаг горизонтального кувырка самостоятельно.

**Мозаика**

Регистр мозаики находится по адресу 0x400004C. Регистр 16-ти битный, биты #15-12 - это вертикальная мозаика, а #11-8 - горизонтальная. Другие биты предназначются для бэкграунда. Кроме инициализации REG_MOSAIC, нам также понадобится установить флаг мозаики в атрибуте 0. Его можно включить, заменив в коде выше VERTICAL FLIP на MOSAIC. Обратите внимание, что вы можете cORить их с помощью символа '|'. Вот изменённый пример:

```
; инициализируем регистр мозаики
ldr r4,=REG_MOSAIC
ldr r3,=0x1200  ; 1 для горизонтальной, 2 для вертикальной, ничего для
                ; бэкграунда
str r3,[r4]   ; сохраняем его

ldr r2,=OAM
ldr r3,[r2] ; загружаем r3 из памяти
; вышеприведённое загрузить r3 атрибутом 0 и атрибутом 1, т.к. регистр
; 32-х битный, а OAM-атрибуты 16-ти битные.
eor r3,r3,MOSAIC   ; XORим r3 с MOSAIC и помещаем значение в r3.
; EOR и XOR - это одно и то же, главное помните, что инструкция называется EOR.
str r3,[r2]  ; сохраняем модифицированные атрибуты обратно в OAM.
```

И вуаля! У вас есть мозаичный спрайт.

**Вращение**

Я надеюсь, что вы прочитали День 3 C-туториала на ThePernProject.COM, потому что если нет, всё это будет достаточно нелегко. Идите, и прочитайте его.

...

...

Прочитайте ещё раз :).

...

...

Думаете, что вы его поняли? Если нет, то неважно, так как я всё равно собираюсь объяснить это сам.

Чтобы с чего-то начать, скажу, что есть нечто под названием "данные вращения" (Rotation Data), в каждом элементе которых есть 4 16-ти битных "пятна". Поняли? Хорошо.

Данные вращения находятся в OAM. Как, спросите вы, разве в OAM не атрибуты спрайтов? Я отвечу: да. Помните, что случилось с атрибутом 3? Мы никогда его не использовали, в то время, как этот атрибут 3 и является данным вращения. Вот маленькая диаграма, показывающая отображение OAM на элементы вращения данных и pa, pb, pc, pd, о которых рассказывается в Дне 3 на ThePernProject:

```
Sprite 0 Att#0
Sprite 0 Att#1
Sprite 0 Att#2
Sprite 0 Att#3 - Элемент данных вращения #0 - 16bit "spot"#1 - pa
Sprite 1 Att#0
Sprite 1 Att#1
Sprite 1 Att#2
Sprite 1 Att#3 - Элемент данных вращения #0 - 16bit "spot"#2 - pb
Sprite 2 Att#0
Sprite 2 Att#1
Sprite 2 Att#2
Sprite 2 Att#3 - Элемент данных вращения #0 - 16bit "spot"#3 - pc
Sprite 3 Att#0
Sprite 3 Att#1
Sprite 3 Att#2
Sprite 3 Att#3 - Элемент данных вращения #0 - 16bit "spot"#4 - pd

Sprite 4 Att#0
Sprite 4 Att#1
Sprite 4 Att#2
Sprite 4 Att#3 - Элемент данных вращения #1 - 16bit "spot"#1 - pa
Sprite 5 Att#0
Sprite 5 Att#1
Sprite 5 Att#2
Sprite 5 Att#3 - Элемент данных вращения #1 - 16bit "spot"#2 - pb
Sprite 6 Att#0
Sprite 6 Att#1
Sprite 6 Att#2
Sprite 6 Att#3 - Элемент данных вращения #1 - 16bit "spot"#3 - pc
Sprite 7 Att#0
Sprite 7 Att#1
Sprite 7 Att#2
Sprite 7 Att#3 - Элемент данных вращения #1 - 16bit "spot"#4 - pd
---- И в таком духе продолжается дальше по всему OAM ----
```

Я поинмаю, что эта диаграмма получилось довольно большой, но я надеюсь, что она хорошо иллюстрирует расположение данных вращения. Обратите, что любой спрайт может использовать любой элемент данных вращения. Есть 32 элемента, то есть 32 спрайта могут вращаться/масштабироваться.

Тогда что нам нужно поместить в каждое из 16-ти битных "пятен" (pa, pb, pc, pd)? Ок, ThePernProject даёт нам несколько очень неплохих уравнений:

```
spot 1 - pa needs to be the COSine of the angle.
spot 2 - pb needs to be the SINe of the angle.
```

```
spot 3 - pc needs to be the NEGATIVE SINe of the angle.
spot 4 - pd needs to be the COSine of the angle.
```

Здесь есть ещё кое-что, но оно относится к вращению, которое мы рассмотрим в другой раз.

Обратите внимание, что "пятно" 3 должно быть отрицательным синусом угла, и это то, что делает вращение таким сложным (по крайней мере для меня). Чтобы инициализировать "пятна", нужно получить атрибуты 2 и 3 какого-либо спрайта, очистить атрибут 3 и "пятно" с помощью AND, а затем установить новое значение с помощью OR. Помните, это 16-ти битные числа, поэтому если мы просто инициализируем атрибут 3 нужным значением, мы можем случайно обнулить атрибут 2.

Чтобы облегчить эту задачу, я сделал несколько (на самом деле, довольно много) новых define'ов в sprites.h (находится в архиве `wasm_files.zip`: `files/0416/sprites.h`). Они предназначаются для данных вращения и выглядят как PA_0, PB_0, PC_0, PD_0. Замените '0' номером элемента данных вращения, который вы хотите использовате. Держите в уме, что в sprites.h (находится в архиве `wasm_files.zip`: `files/0416/sprites.h`) они определены до PD_19 включительно.

Также, чтобы сделать все это проще, так нам нужно только изучить, как вращать спрайты, мы не будем контролировать обнуление атрибута 2 (мне следует написать соответствующий день о байтовых масках). Вместо этого, мы просто используем спрайт #0 и элемент данных вращения #0. Таким образом, мы можем безболезненно обнулять атрибут 2, так как он и так должен быть равен 0.

**Начнём кодирование!**

Я надеюсь, что у вас ещё остался sprit.asm из Дня 5. Держите его и дальше, потому что он понадобится в будущем ещё не раз. Давайте глянем на код!

```
;;--- НАЧАЛО КОДА ---;;
@include screen.h
@include dma.h
@include keypad.h
@include sprites.h

b start
@include sprit.asm
start

ldr r1,=REG_DISPCNT
ldr r2,=(MODE_1|BG2_ENABLE|OBJ_ENABLE|OBJ_MAP_1D)
str r2,[r1]

ldr r1,=REG_DMA3SAD
ldr r2,=pallete
str r2,[r1]
ldr r1,=REG_DMA3DAD
ldr r2,=OBJPAL
str r2,[r1]
ldr r1,=REG_DMA3CNT
ldr r2,=(128|DMA_32NOW)
str r2,[r1]

ldr r1,=REG_DMA3SAD
ldr r2,=obj
str r2,[r1]
ldr r1,=REG_DMA3DAD
ldr r2,=CHARMEM
str r2,[r1]
ldr r1,=REG_DMA3CNT
ldr r2,=(512|DMA_32NOW)
str r2,[r1]
;;--- СТОП КОПИРОВАНИЕ ---;;
```

Полагаю, вы должны понять, что здесь происходит.

```
;;--- ПРОДОЛЖЕНИЕ КОДА ---;;
```

```
ldr r1,=OAM
ldr r2,=((SQUARE|COLOR_256|30))|((SIZE_32|(0<<9)|10)<<16)
str r2,[r1]
;;--- СТОП КОПИРОВАНИЕ ---;;
```

Здесь есть кое-что новенькое. Количество данных вращения сдвигается на 9, чтобы оказаться в правильном "пятне". Мы используем элемент данных вращения #0.

```
;;--- ПРОДОЛЖЕНИЕ КОДА ---;;
ldr r1,=PA_0    ; сохранение будет происходить в PA.
ldr r2,=70<<16  ; COS(45 градусов) = 0.70, в первом байте регистра, который
str r2,[r1]     ; работает с числами с фиксированной запятой в данных вращения.

ldr r1,=PB_0    ; сохранение будет происходить в PB
ldr r2,=70<<16  ; SIN(45 градусов) = 0.70. так же как и в предыдущем случае.
str r2,[r1]

ldr r1,=PC_0    ; сохранение будет происходить в PC
ldr r2,=0       ; Отрицательный SIN(45 degrees) = -0.70
sub r2,r2,70    ; вычитаем 70 от 0, чтобы получить отрицательное значение, и
mov r2,r2,lsl 16   ; сдвигаем, чтобы получить только необходимые 16 бит.
str r2,[r1]

ldr r1,=PD_0    ; сохранение будет происходить в PD
ldr r2,=70<<16  ; COS(45 градусов) = 0.70 снова.
str r2,[r1]  ; сохраняем в атрибуты  #2&3 спрайта 3.
;;--- СТОП КОПИРОВАНИЕ ---;;
```

Обратите внимание, что мы работаем с 16-ти битными числами с фиксированной запятой, поэтом нам нужны эти 3 сдвига на 16 бит не только для того, чтобы получить значение в нужной части регистра, но также и потому, что 70 - это 0.70 и должно находиться в дробной части числа.

```
;;--- ПРОДОЛЖЕНИЕ КОДА ---;;
infin

ldr r1,=KEYS           ; первые 3 строки проверяют, нажата ли клавиша A.
ldr r1,[r1]            ; эти строки включают флаг вращения
ands r1,r1,#KEY_A      ; и флаг двойного размера.
ldreq r1,=OAM          ; однократное нажатие включает эти два флага.
ldreq r2,[r1]          ; получаем атрибуты 0 & 1 спрайта.
eoreq r2,r2,#(ROTATION_FLAG|SIZE_DOUBLE)  ; устанавливаем биты флагов с помощью
                                          ; инструкции EOR (аналог XOR).
streq r2,[r1]
; Обратите внимание, что последние 4 строки выполняются только, если нажата
; клавиша A.

b infin
;;--- КОНЕЦ КОДА ---;;
```

Компилируйте и запускайте! Я надеюсь, что вам всё понятно.

**Обзор этого дня**

Я рад, что доделал этот День, было весело. Да, 13 - счастливое число, вы знаете? Хе-хе. Пока.

# GBA ASM - День 14: Фоновые эффекты

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-08-31

**Какие именно эффекты?** Как и вчера, мы рассмотрим следующее: мозаику, вращение и масштабирование. Правда, что касается вращения, я не буду рассказывать о SIN() и COS() - мы только обсудим, где и что именно нужно вставить в данных вращения/масштабирование, а остальное вам придётся додумать самим. **Мозаика**

Регистр мозаики находится по адресу 0x400004C. Он 16-ти битный - биты 0-3 предназначаются для вертикальной мозаики, а 7-4 - для горизонтальной. Другие биты предназначаются для спрайтов. Кроме инициализации REG_MOSAIC нам также потребуется указать соотвествующий CNT-регистр. Вот код, в котором предполагается, что фон уже отображён:

```
;инициализируем регистр мозаики
ldr r4,=REG_MOSAIC
ldr r3,=0x0012  ; 1 для горизонтальной, 2 для вертикальной или ничего для спрайтов.
str r3,[r4]   ; сохраняем

ldr r2,=REG_BG2CNT  ; BG 2. Если вы используете другой фон, измени номер!
ldr r3,[r2] ; загружаем в r3 значение из памяти, на что указывают '[' и ']'
; код выше загружает в r3 атрибут 0 и атрибут 1 одновременно, так как CPU-регистры
; 32-х битные, а OAM-атрибуты 16-ти битные.
eor r3,r3,BG_MOSAIC_ENABLE   ; XORим r3 с BG_MOSAIC_ENABLE и помещаем значение в r3.
; EOR и XOR - это одно и то же, главное помните, что сама инструкция называется EOR.
str r3,[r2]  ; сохраняет измененные значение обратно в память.
```

И вуаля! У вас получился фон с эффектом мозаики.

*Думаю, понятно, что это совсем как День 13, просто отредактированный для работы с фоном.*

**Вращение (здесь вставьте торжественную музыку!)**

Я надеюсь, что вы хотя бы прочитали День 4 из C-туториала на ThePernProject.COM, так как в противном случае объяснить будет гораздо сложнее. Идите и прочитайте его.

...

...

Прочитайте ещё раз :).

...

...

Думаете, вы всё поняли? Если нет, то не страшно, так как я всё равно попытаюсь это объяснить.

Хотя всего соответствующих регистров 4, нам потребуются только PA и PC, куда мы будем сохранять параметры вращения. Обратите внимание, что иногда фон можно увращать куда-нибудь за пределы видимости.

(Далее я объясню только вращение фон (что очевидно).)

• REG_BGxPA должен содержать косинус угла.
• REG_BGxPB должен содержать синус угла.
• REG_BGxPC должен содержать отрицательный синус угла.
• REG_BGxPD должен содержать косинус угла.

Заметьте, что основное уравнение такое же, как и то, которое мы использовали для спрайтов.

**Давайте начнём писать код!**

Мы будем основываться на коде из Дня 9, поэтому если вы уже забыли, о чём там говорилось, вы можете прочитать его ещё раз. Никогда не бывает лишним что-нибудь перечитать. Поэтому вставьте следующий код между последней строкой и метрой infin (или что я там использовал для метки бесконечного цикла):

```
;;--- НАЧАЛО КОДА ---;;
ldr r1,=70<<16|70
ldr r2,=REG_BG2PA     ; этом поместит значение 70 в PA и PB.
str r1,[r2]

ldr r1,=70<<16  ;
ldr r3,=0             ; этот код
sub r3,r3,70          ; помещает в r3 PC и PD
mov r3,r3,lsl 16      ; в r3 - отрицательное число, поэтому нам нужно переместить
mov r3,r3,lsr 16      ; его в нижние биты регистра (70 находится в верхних).
orr r3,r3,r1          ; Дальше OR'им вместе и сохраняем полученное значение в PC.
ldr r2,=REG_BG2PC
str r3,[r2]
;;--- КОНЕЦ КОДА ---;;
```

Это примерно так же, как и для спрайтов, не считая того, что нам не нужно указывать в соответствующем REG_BGxCNT, что нам нужно вращение, мы просто задаём значения и фон вращается.

Вставляйте, компилируйте и запускайте!

# GBA ASM - День 15: Прокрутка бэкграунда

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

Ок, теперь перейдём к прокрутке бэкграунда. Вы можете удивиться, почему я сразу не поместил её в День 14. Причиной это является то, что наполовину забыл об этом, наполовину поленился (родственная душа - прим. пер.). Так что вот так.

**Как?**

Для текстовых бэкграундов, которые мы не обсуждали, вам нужно просто поместить значение, на которое вы хотите прокручивать в REG_BGxHOFS (горизонтальная) и REG_BGxVOFS (вертикальная) (x - номер бэкграунда).

Для вращающихся бэкграундов регистры REG_BGnX и REG_BGnY (n - номер бэкграунда) - это те, с которыми мы должны иметь дело. Так как они оба 32-х битные (в отличии 2, упомянутых в предыдущем параграфе, которые 16-ти битные), мы можем установить каждый из них отдельно.

Просто поместите значения в регистры и вуаля! Бэкграунд прокручивается!

Пример:

```
ldr r0,=REG_BG2X
ldr r1,=1000  ; Я не уверен, на сколько это его прокрутит.
str r1,[r0]

ldr r3,=REG_BG2Y
ldr r1,=1000
str r1,[r3]
```

Просто, не правда ли?

**Обзор этого дня**

Надеюсь, вам понравится прокручивать бэкграунды!

# GBA ASM - День 16: Маскировка и переключение битов

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

**Маскировка**

Сегодня мы перейдём прямо к делу. Наложение маски - это очищение определённых битов, оставляю другие нетронутыми. Это делается с помощью инструкции AND (по крайней мере в большей части случаев).

```
and (CPUregister),(CPUregister),(number or register)
```

Скажем, нам нужно очистить последние 16 битов регистра, но оставить другие как есть. Предположим, что мы используем r3:

```
and r3,r3,0xFFFF0000
```

Помните, что применение AND над битом и 1 всегда будет давать то значение, которое и было в AND изначально, а если использовать 0, то и результат всегда будет 0, так как логическая таблица AND следующая:

```
 Таблица AND
-------------------------------
Старт  | ANDдится с | Результат
-------------------------------
         0    |     1      |     0
         0    |     0      |     0
         1    |     0      |     0
         1    |     1      |     1
```

Хорошо, если вы уже знаете это, такие таблиц нужны для людей, которым нужно понять всю эту двоичную логику (AND, OR, XOR, NOT) собственными силами, потому что они не использовали ассемблер ранее.

Ок, теперь вам, возможно, интересно, как шестнадцатиричные числа соотносятся с двоичными. Вот таблица:

```
  Шестнадцатиричные в двоичные
---------------------------------
 0 |    0000
 1 |    0001
 2 |    0010
 3 |    0011
 4 |    0100
 5 |    0101
 6 |    0110
 7 |    0111
 8 |    1000
 9 |    1001
 A |    1010
 B |    1011
 C |    1100
 D |    1101
 E |    1110
 F |    1111
```

Теперь вы знаете (если не знали раньше) как сконвертировать любое шестнадцатиричное число в двоичное. Так 0x0000FFFF становится %00000000000000001111111111111111. Процент - это символ, который используется в Goldroad'e для двоичных чисел.

Теперь, что если вы хотите замаскировать определённые биты. Вы можете использовать двоичную систему для инструкции AND. Какие биты вы сделает 0, так и будут 0, когда AND закончен, а те, что были 1 сделают так, что соответствующий бит останется тем, что и был (1 или 0). Пример:

```
; мы хотеим, чтобы биты #3 и #15 обнулились. Не забывайте считать биты
; от #0 слева направо.
and r3,r3,%11111111111111110111111111110111
```

Обратите на подчёркнутые '0'. Соответствующие биты в r3 будут также нулями, но всё остальное останется таким же, каким и было.

Теперь, если нам нужно установить биты в единицу, то мы используем OR. Посмотрите на таблицу этой инструкции:

```
 Таблица OR
-----------------------------
Start  | ORed with | Result
-----------------------------
         0    |     1      |   1
         0    |     0      |   0
         1    |     0      |   1
         1    |     1      |   1
```

Вы можете видеть, что единица является результатом AND, если оба аргумента равны единице. Результатом OR является единица, если последняя является одним из аргументов (или оба).

Чтобы установить определённые биты в 1 (не знаю, зачем вам это может понадобится, но метод, о котором я вам говорю, работает), вам нужно применить на них OR. Биты, которые вы не хотите меня, вам нужно OR'ить с помощью 0. Таким образом, чтобы установить биты #6 и #4 регистра r3 в 1, нам нужно сделать следующее:

```
orr r3,r3,%00000000000000000100000001000000
```

Обратите внимание, что инструкция для OR - это ORR. Мне кажется, что это в каком-то смысле традиция для ARM-инструкций быть 3-х буквенными. Я надеюсь, что вы поняли, что над битами, которые нам нужно установить в 1, мы применяем 1, а над остальными 0.

Также заметье, что GBA - это 32-х битная система, поэтому нам нужно все 32 бита в наших двоичных числах. Если вы не поместите все 32, то думаю, что Goldroad будет считать неуказанные биты равными 0, но я не уверен.

**Переключение**

Во-первых, давайте определим, что это такое. Переключение - это изменение состояния между 1 и 0. Метод, показанный здесь полезен, чтобы переключать биты с помощью одной инструкции.

Простейший способ переключить все биты регистры сразу - это использовать инструкцию MVN. Её синтаксис следующий:

```
MVN (CPUregister),(CPUregister)
```

Все биты во втором регистре будут переключены, а результат возвращён в первый. Поэтому если вы просто хотите переключить все биты r3, сделайте так:

```
mvn r3,r3
```

Просто! MVN расшифровывается как "MoVe Negated" или "MoVe Not").

NOT - это битовая операция как OR и AND, но на входе она имеет лишь один аргумент:

```
 Таблица NOT
--------------
Начало | Итог
--------------
         0    |   1
   1    |   0
```

NOT - это просто, не так ли? В то же время, иногда вам нужно переключить лишь некоторые биты, а не все из них. Для этого вам нужен XOR. Вот соответствующая таблица:

```
 Таблица XOR
-------------------------------
Начало | ПоXORены с | Результат
-------------------------------
  0    |     1      |     1
  0    |     0      |     0
  1    |     0      |     1
  1    |     1      |     0
```

XOR возвращает 1, только если один (но не оба) из аргументов был равен 1. Поэтому XOR с единицей переключит бит. Синтаксис XOR'а следующий:

```
eor (CPUregister),(CPUregister),(number or register)
```

Обратите внимание, что инструкция для XOR'а - это EOR. Оба они расшифорвываются как "Exclusiv OR" (EOR) или "eXclusive OR" (XOR). Поэтому если мы хотим переключить биты #1 и #20:

```
eor r3,r3,%00000000000010000000000000000010
```

XORинг с нулём оставит бит в том же состоянии, что и раньше, но XORинг с единицей установит его в противоположное значение.

**Несколько вещей, которые нужно запомнить**

Несколько замечаний относительно переключения. Зачастую вам нужно будет задавать бит/биты, которые вам нужно переключить, с помощью константы. Вы можете переключить более чем один бит за раз, отделив константы символом '|':

```
eor r3,r3,SIZE_DOUBLE | MOSAIC
; переключаем SIZE_DOUBLE и MOSAIC, полагая, что
; r3 содержит соответствующие OAM-атрибуты
```

Надеюсь, что это было не слишком сложно.

**Обзор этого дня**

Я действительно надеюсь, что вы узнали сегодня что-то новое. Переключение и маскировка битов - это весело!

# GBA ASM - День 17: Обзор

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

**Что нас ждёт дальше**

Прежде всего, это не заключительная речь, скорее речь о том, что будет дальше. Теперь, когда в туториале 16 секций, можно сказать, что основы были рассказаны. Теперь знаете большую часть того, что вам может понадобится для того, чтобы написать не самую простую демку или даже маленьку игру. Я знаю, что некоторые из Дней не слишком хороши, такие как День 10, где код не работает (мне почти удалось его заставить) и День 6, который практически не объясняет сложные уравнения для звука или код для вращения спрайтов и бэкграунда, использующий заранее вычисленные значения (т.е. не вычисляющий COS и SIN на лету). Теперь, когда я сказал об этом, скажу о том, что будет дальше.

Остальные Дни будут посвящены более сложным вещам, таким, как обсуждение BIOS'a. Также, возможно, что будут обсуждены столкновения BG/Спрайтов или графические алгоритмы для режима 3. Не думаю, что мы упустили какие-либо из базовых возможностей GBA кроме Direct Sound, что не под силу мне даже на C.

И, наконец, я хотел бы поблагодарить GBADev.ORG, SimonB и krom.

Следующая глава - про BIOS!

# GBA ASM - День 18: О BIOS'e

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

**BIOS?** Да, BIOS. Эта аббревиатура расшифровывается как Basic Input Output System. Не знаю, что они называют BIOS'ом GBA, на экране, по крайней мере, оно ничего не отображает. BIOS GBA - это больше подсобная библиотека с несколькими полезными функциями, которые можно использовать. **Что есть в BIOS'e**

В BIOS'e есть следующие функции, которые могут быть нам интересными:

• Sqrt - Square Root Function, puts Square Root of r0 in r0.
• Div - Divide Function, divides r0 by r1 and puts the answer in r0. Also returns other things in r1 and r2, we'll get to those in Day 20.
• SoftReset - I suppose we could try doing this one tomorrow, maybe.

Чтобы вызвать функцию BIOS'a, вам нужно просто заполнить определённые регистры CPU, а потом вызвать функцию с помощью инструкции, которую мы обсудим завтра.

**Где находится BIOS?** BIOS начинается с 0x0000000 в памяти и идёт до 0x1000000 (думаю). BIOS также содержит несколько распаковочных инстуркций и несколько процедур для установки свойств вращения/масштабирования как для спрайтов, так и для бэкграундов. **Обзор этого дня**

Это всё на сегодня, завтра мы будем вызывать сами функции.

# GBA ASM - День 19: Инструкция SWI

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

**Что?!?** Вы можете спросить, почему я не сказал о SWI в прошлый раз. Я сделал это для того, чтобы говорить об одной теме за раз. SWI расшифровывается как "SoftWare Interrupt" (программное прерывание) и используется для вызова функций BIOS. **Как использовать SWI**

SWI используется.. ох, просто смотрите код:

```
swi 0x60000  ; вызовет функцию деления
```

Деление - это SWI #6, и чтобы вызвать функцию, просто добавьте к её номеру 4 нуля. Я не знаю, зачем они нужны, но без них ничего работать не будет.

Вы можете спросить, откуда я узнал, что номер Divide - #6, идите сюда и смотрите секцию "BIOS".

**Вызов SoftReset**

SoftReset перегружает GBA и стартует код по адресу 0x8000000 (ROM) или 0x2000000 (RAM, некоторые люди перемещают туда свой код при загрузке) в зависимости от определённого регистра в памяти, который по умолчанию равен 0, что значит 0x8000000, так что у нас не будет каких-либо проблем с эим. Давайте вызовем его!

```
swi 0x00000  ; SoftReset - #0 , после которого ещё 4 нуля.
```

Просто, не правда ли?

# GBA ASM - День 20: Использование деления и квадратного корня

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

**Деление**

Давайте перейдём прямо к делу. Вот прокомментированный код:

```
lda r0,=20  ; r0 = 20
lda r1,=5   ; r1 = 5
swi 0x60000  ; r0 = r0/r1
      ; Та же:
      ; r1 = r0 Mod (%) r1
      ; r2 = Abs(r0/r1)
```

Вот небольшой исходник, ассемблируйте его и запустите:

```
;;--- НАЧАЛО КОДА ---;;

; нам не нужны заголовочные файлы, так как мы не собираемся делать
; вывод чего-либо или использовать какие-либо константы

@textarea 0x3000000
g @dcd 0  ; 32-х битная "переменная" g.
@endarea

lda r0,=20  ; Эти строки
lda r1,=5   ; делят 20
swi 0x60000 ; на 5.

lda r3,=g  ; загружает в r3 адрес g
sta r0,[r3] ; сохраняет результат деления (должно быть равно 4
    ; в "переменной" g, которая располагается в 0x3000000.

infin  ; infinite loop
b infin

;;--- КОНЕЦ КОДА ---;;
```

Ассемблируйте, а потом запустите в VisualBoy Advance. Откройте просмотровщик памяти и идите в "0x3000000 - IRAM".

Вы должны будете увидеть '4' в первых 32-х битах 0x3000000, так как 20/5=4.

**Квадратный корень**

Sqrt имеет номер 8. Поэтому, чтобы изменить код выше для работы с ней, сделайте три вещи:

1. Сделайте так, чтобы в r0 загружалось 81
2. Либо закомментируйте "lda r1, =5", либо оставьте как есть, это не окажет эффекта, так как r1 не используется в качестве аргумента для этой функции.
3. Измените "swi 0x60000" на "swi 0x80000".

Протестируйте, используя тот же метод, что я описал ранее.

**Обзор этого дня**

Использование деление и квадратного корня может очень пригодится, так как соответствующих инструкций нет.

# GBA ASM - День 21: LDRH и STRH

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

**Повтори?**

Сегодня мы рассмотрим две инструкции, о которых в силу некоторых причин я совершенно забыл. Фактически, из-за этого я писал довольно неэффективный код. Эти инструкции ldrh и strh.

Эти инструкции загружают/сохраняют половину слов. Они работают так же как и обычные STR и LDR, не считая того, что используют только нижние 16 битов регистра(ов). Очевидно, что это должно помочь с теми регистрами памяти, которые только 16 битов размером.

Однажды я был на форуме GBADev, чтобы узнать, как решить проблему с одним моим макросом. Он должно был устанавливать номер спрайта в OAM, но в силу неустановленных причин он не работал.

Поэтому один очень умный человек дал мне пример кода, использующий LDRH и STRH, и который работал превосходно. Я совсем забыл о существовании этих двух инструкций. Поэтому, чтобы познакомить вас с этими магическими инструментами перемещающей данные силы, я объясню этот макрос.

**Переходим к нему**

Вот сам макрос:

```
@macro SetNumber sprite, number
 ldr r3,=(OAM+(sprite*8))+6  ; загружаем в r3 адрес номера тайла
 ldr r4,=number  ; затем загружаем номер в нижние биты (как обычно)
 strh r4,[r3]  ; и сохраняем нижние 16 бит куда нужно
@endm
```

Я понимаю, что это демонстрирует только инструкцию STRH, чтобы загрузить что-то используйте LDRH следующим образом:

```
ldrh r3,[r1]  ; обратите внимание, что мы берём 16 битов по адресу в r1
              ; и помещаем в r3
```

Также как и в старые времена, не считая 16-ти бит.

**Обзор этого дня**

Мои благодарности помогшему на форуме GBADev человеку, который напомнил мне об этих инструкциях.

Эти инструкции очень полезны для создания более полезного кода. Используйте их правильно.

# GBA ASM - День 22: Сохранение и 0xE000000

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

**Введение** Ок... Я понял, что нужно затронуть ещё один важный вопрос. Сегодня мы собираемся рассмотреть, как сохранить что-то в картридж. **Инструкции LDRB/STRB**

Эти инструкции загружают или сохраняют в/из памяти и нужны для того, чтобы читать/писать в память сохранения (SRAM). Они работают так же, как и другие инструкции загрузки/сохранения.

Память сохранения находится по адресу E000000. Всё, что вам нужно, это записать что-либо по любому адресу с 0xE000000 до 0xE00FFFF (это 64kb памяти) с помощью инструкции STRB. Чтобы читать из SRAM, используйте инструкцию LDRB. Просто!

Чтение первого байта из SRAM:

```
ldr r1,=0xE000000
ldrb r1,[r1]
```

Запись первого байта в SRAM:

```
ldr r1,=0xE000000
ldr r2,=0xFF  ; мы собираемся записать 0xFF в SRAM
; обратите внимание, что мы, скорее всего, используем обычный LDR
       ; для загрузки числа в регистр
strb r2,[r1]
```

Надеюсь, это поможет всем, у кого проблемы с SRAM, хотя думаю, что это не так, потому что её очень легко использовать.

**Обзор этого дня**

Учитывая, что сохранение настолько просто, удивительно, что множество игр используют систему паролей...

# GBA ASM - День 23: Использование AS

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

**AS?** Похоже, что Goldroad Assembler больше не обновляется, поэтому давайте посмотрим на GNU Assembler. **Базовый синтаксис ассемблера**

Насколько мне известно, базовый синтаксис тот же самый, никаких различий в использовании инструкций нет:

```
ldr r1,=0x4000000  ; так же, как и раньше
```

**Константы и препроцессор**

Одно из основных различий - это константы. Чтобы объявить последнюю, используйте EQU следующим образом:

```
.equ var, 0x2000000 ; байты 0-3, (32 бита на переменную)
.equ another, 0x2000004 ; байты 4-7
```

или

```
.equ REG_DISPCNT, 0x4000000
```

Вы можете захотеть сконвертировать заголовочные файлы, но есть и другая опция. Вы можете ассемблировать ваши .S-файлы (другое расширение для исходников), если вы сделаете следующее:

```
gcc -c test.S
objcopy -O binary test.o test.gba
```

Вы сможете использовать #define, поэтому надо будет заменить '@' на '#' и всё! (Обратите внимание, что синтаксис макросов другой, поэтому вам придётся их убрать.)

**День 1 с GNU**

Вот код, выводящий белую точку, который можно использовать с GCC:

```
@-- НАЧАЛО КОДА --;
@ комментарии начинаются с @
#define REG_DISPCNT 0x4000000

.text @ the text section
start:
 ldr r1,=0x403 @Режим 3 , BG2 включен
 ldr r2,=REG_DISPCNT
 str r1,[r2]

 ldr r1,=0xFFFF @Белый, (Знаю, что на самом деле это 0x7FFF, но какая разница)
 ldr r2,=0x6000200  @где-то на экране
 str r1,[r2]

infin:
 b infin
@-- КОНЕЦ КОДА --;
```

Чтобы скомпилировать исходник, поместите test.S на рабочий стол. Я предполагаю, что директория с GNU-ассемблером - C:\devkitadv\:

```
set path=C:\devkitadv\bin\
cd PATH_TO_YOUR_DESKTOP_HERE
gcc -c test.S
objcopy -O binary test.o test.gba
```

Надеюсь, что у вас не будет проблем с компиляцией файла, так как у GNU-программ есть куча заморочек...

**Обзор этого дня**

Я надеюсь получить ссылку на ассемблерный исходик, использующийся с C/C++.

# GBA ASM - День 24: Компилирование вместе с C

**Автор:** Mike H, пер. Aquila

**Дата:** 2006-09-10

Компиляция с C?

Да, сегодня мы используем как ASM, так и C++-функции в отдельных файлах, скомпилировав оба.

**C++-файле**

В test.cpp :

```
/* НАЧАЛО КОДА */
#include "C:\devkitadv\include\gba.h"

extern "C" {
void SetThePixel(); /* ASM-функция использует C-соглашение */
}

int main() {
 REG_DISPCNT = 0x403; /* режим 3 */
 SetThePixel(); /* ASM-функция */
 while((*KEYS)&KEY_START);  /* ждём нажатия на клавишу 'Start' */
 REG_DISPCNT = 0; /* нет режима (точка исчезнет) */
 while(1);
 return 0;
}
/* КОНЕЦ КОДА */
```

**ASM-файл**

В test.S :

```
@-- НАЧАЛО КОДА --@
 .global SetThePixel
SetThePixel:
 ldr r1,=0x6000200
 ldr r2,=0x00FF
 str r2,[r1]      @ красная точка где-то на экране
 bx lr  @return;
@-- CODE END --@
```

Замечание: сделайте перевод строки в конце каждого из файлов.

**Компиляция**

Считая, что оба исходника находятся в одной директории и C:\devkitadv\bin\ находится на пути, то для компиляции потребуется следующее:

```
gcc test.S test.cpp
objcopy -O binary a.out test.gba
```

Затем запустите TEST.GBA, чтобы убедиться в работоспособности примера.

**Обзор этого дня**

Ссылки:

• Пост по этой теме на форуме GBADev.ORG.
• Ссылка, где объясняется, почему необходим "extern "C"".

Надеюсь, вам понравилось.

# 22. Байт-код

# Unsafe Java I - Небезопасная жаба

**Автор:** Stiver

**Дата:** 2006-05-20

1. Класс sun.misc.Unsafe
2. Структуры виртуальной машины
3. Особенности версии 1.5
4. Применение на практике
5. Бесконечный final

Как известно при разработке языка Java с самого начала делался упор на "безопасность" кода (так называемый "safe code"). Помимо всего прочего это означало отказ от указателей, работы с памятью и тому подобных низкоуровневых средств. Совсем отказаться правда не удалось, пришлось оставить лазейку, в первую очередь естественно для собственных классов. Но все, что использует Java Runtime, можем использовать и мы. В этой статье мы научимся писать небезопасный код на Яве и используем новоприобретенные знания для решения некоторых интересных проблем, которые штатными средствами Явы не решаются.

Рассматривать мы будем только Sun'овскую виртуальную машину, по двум причинам. Во-первых она применяется наиболее широко и является своеобразным эталоном. Во-вторых до IBM'овской у меня руки еще не дошли, а больше никаких (реально использующихся) я не знаю. Все приведенные ниже примеры протестированы с Java 1.4.2_11 и 1.5.0_06. Предполагается что читатель достаточно хорошо разбирается как в Яве, так и в общих принципах программирования.

**1. Класс sun.misc.Unsafe**

Малоизвестный класс sun.misc.Unsafe входит в комплект Sun Java Runtime начиная с первых версий. Как и все остальные классы в package sun.*, Unsafe не документирован, но имена (в большинстве своем нативных) функций, видимые при декомпиляции, говорят сами за себя. Явно присутствуют функции работы с памятью (allocateMemory, freeMemory,...), чтения и записи значений по заданному адресу(putLong, getLong,...) и некоторые более специализированные(throwException, monitorEnter,...). То есть в принципе все, что нам нужно.

Правда так просто инстанциировать Unsafe не удастся. Единственный constructor - приватный, а в getUnsafe() проверяется загрузчик вызвавшего класса и объект возвращается только если класс загружен Bootloader'ом. В противном случае получаем SecurityException.

```
public static Unsafe getUnsafe()
{
    Class class1 = Reflection.getCallerClass(2);
    if(class1.getClassLoader() != null)
        throw new SecurityException("Unsafe");
    else
        return theUnsafe;
}
```

К счастью существует еще внутренняя переменная theUnsafe, до которой мы можем добраться с помощью Reflection. Всю черновую работу соберем в один класс (назовем его UnsafeUtil), который будем расширять по мере надобности.

```
public class UnsafeUtil {

 public static Unsafe unsafe;
 private static long fieldOffset;
```

```
 private static UnsafeUtil instance = new UnsafeUtil();

 private Object obj;

 static {
  try {
   Field f = Unsafe.class.getDeclaredField("theUnsafe");
   f.setAccessible(true);

   unsafe = (Unsafe)f.get(null);
   fieldOffset = unsafe.objectFieldOffset(UnsafeUtil.class.getDeclaredField("obj"));
  } catch (Exception ex) {
   throw new RuntimeException(e);
  }
 };
}
```

Конечно можно просто внести UnsafeUtil в список загружаемых Bootloader'ом классов (указав путь в ключе -Xbootclasspath/a) и вызывать getUnsafe() в соответствии с замыслом Sun. Беда в том, что тогда все использующие UnsafeUtil классы также должны быть прописаны в bootclasspath'e (см. главу "5.3 Creation and Loading" в VM spec). Правда например package java.nio как-то ухитряется обходить это ограничение, но как именно пока не очень понятно. К тому же этот способ выходит за рамки "чистого" кода, так как требует дополнительных стартовых опций для виртуальной машины. Так что не будем мудрствовать и ограничимся чтением theUnsafe.

В первую очередь нам понадобятся естественно операции референцирования и дереференцирования, ObjectToAddress и AddressToObject соответственно.

```
public static long ObjectToAddress (Object o){
 instance.obj = o;
 return unsafe.getLong(instance, fieldOffset);
}

public static Object AddressToObject (long address){
 unsafe.putLong(instance, fieldOffset, address);
 return instance.obj;
}
```

С ними мы уже достаточно хорошо вооружены в техническом плане, не хватает только информации по внутреннему устройству Явы. Ее мы найдем в следующем разделе.

Очень похожую реализацию кстати сделал Don Schwarz (http://don.schwarz.name/index.php?p=30). Это одно из очень немногих мест, где можно найти хоть какие-то примеры работы с классом Unsafe. К сожалению Don в свое время не оценил потенциал низкоуровнего программирования в Яве и остановился, сделав всего пару робких шагов. Мы же пойдем дальше.

**2. Структуры виртуальной машины** Теперь посмотрим, в каком виде виртуальная машина (версии 1.4) хранит данные в памяти. Поскольку Ява работает с классами и их инстанциями, то ими и займемся. **Инстанция:**

```
instance_struct {
 0 magic // всегда равен 1
 4 class   // указатель на структуру класса, class_struct*
 8 ...     // Дальше идут подряд переменные инстанции(то есть все которые не static),
 12 ...     // порядок пока не очень понятен, судя по всему в порядке объявления и
 16 ...     // обьекты перед примитивными типами
 ...
}
```

Переменные типа double и long занимают 64 бита, остальные по 32. В памяти инстанции выравниваются по 64-битной границе, дополняются при необходимости нулями. То есть по сути мы имеем обыкновенную сишную структуру плюс указатель на ее описание.

**Класс:**

```
class_struct {
```

```
 0 magic    // всегда равен 1
 4 class    // class_struct*, указатель на структуру класса более высокого уровня, зачем нужен -
    непонятно
 8 ...      // значение неизвестно
 12 super_count     // количество уровней наследования: 0x18 - один(наследует от Object), 0x1c -
    два и т.д. до восьми(0x34), потом 0x10. У интерфейсов тоже 0x10.
 16    interface     // class_struct*, указатель на какой-либо из интерфейсов класса (на какой
    именно непонятно), часто просто 0
 20 interface_list  // указатель на массив с элементами типа class_struct*, все интерфейсы класса
 24 ...
 28 ...
 32 ...
 36 ...     // указатели на структуры восьми высших суперклассов начиная от Object и кончая this (
    если поместится)
 40 ...
 44 ...
 48 ...
 52 ...
 56 size  // размер инстанции класса в DWORD'ах
 60 this_class   // instance_struct*, указатель на инстанцию java.lang.Class соответствующую
    данному классу
 64 access_flags   // доступ к классу как описано в VM spec ( 0x1 - public, 0x10 - final и т.д.)
 68 ...
 ...
}
```

Здесь я привел только те куски, которые мы будем использовать в дальнейшем и в которых я более или менее уверен. На самом деле class_struct значительно длиннее и содержит кроме того указатели на функции класса, статические переменные и все остальное, что может понадобится виртуальной машине. Все эти структуры по понятным причинам нигде не документированы и разбираться надо вручную - хоть и несложно, но достаточно трудоемко. Если у кого-то есть желание помочь, буду только рад.

### 3. Особенности версии 1.5

С переходом на последнюю (на момент написания) версию 1.5.0_06 внутренние структуры виртуальной машины претерпели некоторые изменения. К счастью небольшие: изменился в основном порядок полей, значения остались в большинстве прежними. Структура класса выглядит теперь следующим образом:

```
class_struct_1_5 {
 0 magic    // всегда равен 1
 4 class    // class_struct*, указатель на структуру класса более высокого уровня, зачем нужен -
    непонятно
 8 ...      // значение неизвестно
 12 size  // размер инстанции класса в DWORD'ах
 16 super_count     // количество уровней наследования: 0x20 - один(наследует от Object), 0x24 -
    два и т.д. до восьми(0x38), потом 0x14. У интерфейсов тоже 0x14.
 20    interface     // class_struct*, указатель на какой-либо из интерфейсов класса (на какой
    именно непонятно), часто просто 0
 24 interface_list  // указатель на массив с элементами типа class_struct*, все интерфейсы класса
 28 ...
 32 ...
 36 ...
 40 ...     // указатели на структуры восьми высших суперклассов начиная от Object и кончая this (
    если поместится)
 44 ...
 48 ...
 52 ...
 56 ...
 60 this_class   // instance_struct*, указатель на инстанцию java.lang.Class соответствующую
    данному классу
 64 ...
 68 ...  // точные значения неизвестны
 72 ...
 76 ...
 80 access_flags   // доступ к классу как описано в VM spec ( 0x1 - public, 0x10 - final и т.д.)
 84 ...
```

```
  ...
}
```

Обратите внимание на переехавшее вперед поле size и сдвинутые по сравнению с версией 1.4 значения поля super_count. При написании кода придется учитывать подобные мелкие отличия. Поэтому будем в самом начале опрашивать версию виртуальной машины и сохранять результат в переменной vm1_5. Интересно кстати, что версии 1.5.0_0x, x<6 используют все еще старые структуры. То есть достаточно глобальные изменения в виртуальной машине не обязательно приурочены к значительному скачку версии - сам по себе примечательный факт.

**4. Применение на практике**

Перейдем к практической части и попробуем приспособить теорию к делу. Для начала решим одну проблему, которая существует почти столько же сколько и сама Ява. А именно напишем функцию sizeof() для объектов. Желающие могут использовать свой любимый поисковик и посмотреть(например по Java+sizeof), сколько и каких решений предлагалось за последние годы, от использования Reflection до вычисления размера занятой памяти и деления его на количество объектов. Точного ответа при этом не давало, что интересно, ни одно. Нам же достаточно просто прочитать поле class_struct.size

```
public static long sizeOf(Object object){
    return unsafe.getAddress(unsafe.getAddress(ObjectToAddress(object)+4)+(vm1_5?12:56));
}
```

Функция возвращает результат в DWORD'ax, если нужен в байтах не забудьте умножить на 4. С помощью sizeOf() можно теперь копировать содержимое инстанций - так называемая "shallow copy". "Shallow" - так как в случае внутренних переменных типа Object (и от него производных) копируются естественно только указатели, а не объекты целиком.

```
public static void copyObjectShallow(Object objectSource, Object objectDest) {
    unsafe.copyMemory(ObjectToAddress(objectSource),ObjectToAddress(objectDest),sizeOf(
    objectSource)*4);
}
```

copyObjectShallow бывает особенно полезна если иметь дело с объектами, содержащими большое количество переменных примитивных типов. То есть когда класс используется в основном для хранения данных, как структура в C. Копировать переменные по одной (единственный штатный способ Явы) - удовольствия мало.

Как известно, приведение типов в Яве осуществляется динамически, с учетом иерархии классов. Бинарного каста (reinterpret_cast в терминах C++) Ява к сожалению не поддерживает. Заполним этот пробел.

```
public static Object reinterpret_cast(Object o, Class cl){
    unsafe.putAddress(ObjectToAddress(o)+4,unsafe.getAddress(ObjectToAddress(cl)+8));
    return o;
}
```

reinterpret_cast возвращает указатель на объект o, приведенный к заданному через параметер cl типу. Имеет ли такое преобразование смысл, должен как всегда решать сам пользователь. Не следует только забывать, что ошибка при подобных манипуляциях с памятью почти всегда вызовет не безобидную Java Exception, а что-нибудь вроде Access Violation в виртуальной машине. Классический случай применения reinterpret_cast - когда один и тот же класс загружается два раза двумя разными ClassLoader'ами.

Если собираетесь писать многопоточную программу, не забывайте о синхронизации. Например имеет смысл добавить в определения приведенных выше функций слово synchronized. Иначе может случиться так, что разные потоки попытаются одновременно изменять структуры классов и радости тогда будет много.

Исходный код класса UnsafeUtil вместе с примерами использования отдельных его функций находится в приложении к статье.

**5. Бесконечный final**

Чтобы лучше оценить те практически неограниченные возможности, которые открывает перед нами манипулирование структурами классов, разберем пример посложнее.

Как известно, "final" в объявлении класса запрещает наследование от него. Например классы типов, такие как String, Integer и т.д. умышленно сделаны разработчиками языка конечными. Новички с завидным постоянством спрашивают на форумах, можно ли написать собственный класс строк, наследующий от String, и с таким же постоянством получают ответ "нельзя". И тем не менее правильный ответ - можно.

Для простоты и наглядности возьмем функцию hashCode(). Как известно хэш строк вычисляется по алгоритму

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

где n - длина строки и s[i] - i-й символ. Предположим теперь, что нас не устраивает стандартный алгоритм и мы хотим вычислять сумму не с начала строки, а с конца.
Вот так:

```
s[n-1]*31^(n-1) + s[n-2]*31^(n-2) + ... + s[0]
```

То есть нужен класс, который наследует от String и имплементирует новую hashCode(). И именно эти свойства имеет класс MagicString, который можно найти в директории /string в исходниках к статье. MagicStringWthStub реализует ту же самую идею, только чуть элегантнее, например без использования Reflection. Недостатком в этом случае является необходимость написания stub'a, что впрочем можно легко автоматизировать. MagicString обходится без дополнительных классов.

Код самих классов я здесь приводить не хочу, чтобы не загромождать статью. Основная идея состоит в том, что мы вносим String в список суперклассов(смещения 24-52 + поле super_count) и подправляем поле access_flags нужным образом (убираем final). Детали реализации можно посмотреть в приложенных исходниках. Проверим теперь MagicString в действии:

```
String s = (String)(Object)(new MagicStringWithStub("AB"));
System.out.println("Magic String hashcode: "+s.hashCode());
String s1 = new String("AB");
System.out.println("Standard String hashcode: "+s1.hashCode());
```

На выходе получим

```
Magic String hashcode: 2111
Standard String hashcode: 2081
```

Как видим, единственное небольшое неудобство состоит в том, что кастить в String приходится через Object. Понятно почему: компилятор-то о наших играх ничего не знает. В остальном мы полностью достигли цели: получили String с нестандартным хэш-кодом.

В следующей статье (при условии, что у меня дойдут руки ее написать :)) мы научимся создавать на Яве самомодифицирующийся код.

**Благодарности** Quantum - за вдумчивое и терпеливое рецензирование черновых вариантов статьи. Если я не последовал каким-либо его советам, то исключительно по причине собственной лени. **Приложение**

unsafejava1code.zip (находится в архиве `wasm_files.zip`: `files/0402/unsafe_java_1_code.zip`) (7 KB) - Примеры к статье

# Введение в MSIL - Часть 1: Hello World

**Автор:** Кенни Керр, пер. Aquila

**Дата:** 2006-07-19

Описывая C++/CLI и его связь с C#, мне часто хочется обсудить Microsoft intermediate language (MSIL), генерируемые компиляторами Visual C++ и Visual C#. Проблема состоит в то, что большинство программистов не знакомы с MSIL и такие программы как ILDASM не слишком полезны для новичка, потому что отображаемые ими MSIL для него не читаем. Поэтому я решил написать несколько статей, рассказывающие о MSIL на вводном уровне. Так как я сильно сомневаюсь, что после их прочтения кто-то действительно пойдёт и начнёт писать код на MSIL, я пропущу некоторые неинтересные детали и сфокусируюсь на таких вещах, как определение типов, написание методов, инструкции вызова и обработка исключений.

Давайте начнём с простой стартовой функции, которая отображает очень неожиданное сообщение. В отличии от C#, CLI не требует, чтобы метод принадлежал к какому-либо классу. Стартовая функция также не обязательно должна называться 'main', но для упрощение я использовал именно это имя.

```
.method static void main()
{
    .entrypoint
    .maxstack 1

    ldstr "Hello world!"
    call void [mscorlib]System.Console::WriteLine(string)

    ret
}
```

Метод main называется определением метода, так как и сигнатура и тело метода наличествуют. Когда же есть только сигнатура без тела, это называется объявлением метода. Последние обычно используются как цели вызова (когда метод вызывается), в то время как определение метода предоставляет конкретную его реализацию.

Определение метода начинается с директивы .method и может находится в зоне глобальной видимости или внутри класса. Стартовая функция приложения должна быть статической. Объявление глобального метода статическим может показаться избыточным, но компилятор ILASM в некоторых случаях жалуется, если вы опускаете ключевое слово static.

Директива .entrypoint указывает, что этот метод является стартовым для приложения. Только в одном методе приложения может быть такая директива.

Директива .maxstack говорит, сколько слотов в стеке предположительно будет использовано. Например, сложение двух чисел потребует поместить два числа в стек и вызвать инструкцию add, которая возьмёт сверху эти два числа и положит обратно результат. В этом случае вам понадобится два стековых слота.

Инструкция ldstr помещает строку, которая должна быть передана методу WriteLine, в стек. Инструкция call вызывает статический метод WriteLine класса System.Console из сборки mscorlib. Это пример объявления метода, с полный сигнатурой метода WriteLine (включая строковый аргумент), чтобы среда выполнения могла определить, какую конкретно перегрузку метода необходимо вызвать.

И наконец, инструкция ret возвращеет управление вызывающему. В случае стартовой функции это означает конец выполнения приложения.

# Введение в MSIL - Часть 2: Использование локальных переменных

**Автор:** Кенни Керр, пер. Aquila

**Дата:** 2006-07-22

В этой части я расскажу об использовании локальных переменных. Без них программы были бы не слишком интересны. Чтобы объяснить, как использовать переменные, давайте напишем простую программу для сложения чисел.

Внутри метода локальные переменные задаются с помощью директивы .locals.

```
.locals init (int32 first,
              int32 second,
              int32 result)
```

Это утверждение декларирует, что у данного метода есть три локальные переменные. В этом случае они все оказались одного типа - int32, что является синонимом типа System.Int32. init задаёт, что переменные должны быть инициализированы значением по умолчанию данного типа. Также возможно опускать имена переменных, и в этом случае вы сможете обращаться к ним по их индексу (начиная с нуля). Разумеется, использование имён существенно повышает читабельность.

Прежде чем мы продолжим, я хочу убедиться, что вы понимаете, как используется стек в MSIL. Когда вы хотите передать требуемые значения инструкции, их нужно поместить в стек. Чтобы прочитать эти значения, инструкция берёт их из стека. Похожим образом, при вызове метода вам нужно поместить в стек ссылку на объект (если нужно), а за ней - аргументы. В процессе вызова метода все аргументы вместе с ссылкой на объект будут взяты из стека. Для помещения значения в стек используется инструкция ldloc, используемую с переменной, содержащей значение. Для того, чтобы взять значение из стека, используется инструкция stloc, которой указывается переменная, куда сохраняется значение. Также держите в уме, что значения (с учётом их типа) сохраняются прямо в стек, а объекты - нет, так как CLI этого не позволяет. Вместо этого в стек помещаются ссылки на них. Это похоже на C++-объект, находящийся в "куче", с указателем на него в стеке. Никогда не забывайте о стеке, читая эту серию статей. Это поможет вам понять, почему значения постоянно помещаются и берутся из него.

Следующим шагом будет получение чисел от пользователя.

```
ldstr "First number: "
call void [mscorlib]System.Console::Write(string)
call string [mscorlib]System.Console::ReadLine()
call int32 [mscorlib]System.Int32::Parse(string)
stloc first
```

Как я упомянул в первой части, инструкция ldstr помещает строку в стек и, а инструкция call вызывает метод Write, беря свои аргументы из стека. Затем вызывается метод ReadLine, возвращающий строку, которая помещается в стек, и, поскольку она уже там, нам остаётся просто вызвать метод Int32::Parse, берущий строку из стека и помещающий в него соответствующий in32-эквиваент. Обратите внимание, что для простоты я опустил всё, что связано с обработкой ошибок. Затем инструкция stloc достаёт значение из стека и сохраняет его в локальную переменную 'first'. Получение следующего числа происходит примерно также, не считая, что значение сохраняется в переменной 'second'.

Теперь, прочитав два числа из стандартного потока ввода, настало время, чтобы их сложить. Для этого можно использовать инструкцию add.

```
ldloc first
ldloc second
```

```
add
stloc result
```

Она берёт два числа из стека и считает сумму. Чтобы поместить оба числа в стек, мы используем ldloc. После выполнения инструкции add полученный результат помещается в стек и программа сохраняет его в переменную 'result', используя stloc.

Последний шаг - это отобразить результат пользователю.

```
ldstr "{0} + {1} = {2}"

ldloc first
box int32

ldloc second
box int32

ldloc result
box int32

call void [mscorlib]System.Console::WriteLine(string, object, object, object)
```

Мы использовали перегруженный вариант WriteLine, принимающую форматирующую строку и три объекта. Каждый аргумент должен быть помещён в стек один за другим. Так как числа имеют тип int32, нам нужно сделать из них объекты, иначе получится несовпадение с сигнатурой метода.

Инструкция ldloc помещает каждый аргумент в стек. Затем для каждого аргумента int32 используется инструкция box. Она достаёт значение из стека, делает на его основе объект и помещает обратно в стек ссылку на него.

Вот полная программа.

```
.method static void main()
{
    .entrypoint
    .maxstack 4

    .locals init (int32 first,
                  int32 second,
                  int32 result)

    ldstr "First number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
    stloc first

    ldstr "Second number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
    stloc second

    ldloc first
    ldloc second
    add
    stloc result

    ldstr "{0} + {1} = {2}"

    ldloc first
    box int32

    ldloc second
    box int32

    ldloc result
    box int32
    call void [mscorlib]System.Console::WriteLine(string, object, object, object)
```

```
    ret
}
```

Последнее в этом примере, на что вам стоит обратить своё внимание, это то, что в методе указано максимальное количество используемых стек-слотов равное 4, что было сделано с учётом метода WriteLine, вызываемого в конце программы, которому требуется передать именно это количество аргументов.

# Введение в MSIL - Часть 3: Определение типов

**Автор:** Кенни Керр, пер. Aquila

**Дата:** 2006-08-06

В этой части я объясню, как задаются типы.

Вот минимальный ссылочный тип под названием House ("дом", - прим. пер.).

```
.class Kerr.RealEstate.House
{
    .method public void .ctor()
    {
        .maxstack 1

        ldarg.0 // push "this" instance onto the stack
        call instance void [mscorlib]System.Object::.ctor()

        ret
    }
}
```

Это очень простой тип. Обратите внимание, что вы должны объявить конструктор для ссылочного типа. В отличии от таких языков как C# и C++, ассемблер IL не будет генерировать для вас конструктор автоматически.

Типы задаются с помощью директивы .class, за которой следует заголовок типа. Ключевое слово class используется вместо более интуитивно понятного type в силу исторических причин. Когда вы встречаете в исходном MSIL-коде слово class, просто читайте его как type. Заголовок типа состоит из определённого количества аттрибутов, за которым следует имя типа, который вы определяете. Чтобы задать эквивалент статического класса C#, вы можете написать следующее:

```
.class abstract sealed Kerr.RealEstate.MortgageCalculator
{
    /* members */
}
```

abstract и sealed - это атрибуты типа. Нельзя сделать экземпляр абстрактного типа, а запечатанный (sealed) тип не может иметь подтипы. Есть атрибуты, которые служать для контроля видимости, такие как public и private. Есть атрибуты для контроля местоположения поля (field layout), такие как auto и sequential. Полный лист атрибутов можно узнать в спецификации CLI. Многие атрибуты применяются автоматически, что экономит на печати символов. К счастью, эти атрубты по умолчанию достаточно интуитивны, так что вы познакомитесь с ними достаточно быстро. Например, расширение System.ValueType из сборки mscorlib задаёт новый тип значений. Так как CLI требует, что типы значений были запечаны, ассемблер IL автоматически присвоит этот атрибут.

Имя типа в вышеприведённом примере было Kerr.RealEstate.MortgageCalculator. CLI не распознаёт пространства имён сами по себе, скорее речь идёт об использовании полного имени типа. Применённый выше синтаксис поддерживается ассемблером IL, который поставляется с .NET Framework версии 2.0. Если вы работаете с версией 1.x, тогда вам придётся применить директиву .namespace так. как это показано ниже. Обратите внимание, что этот синтаксис также поддерживается в версии 2.0.

```
.namespace Kerr.RealEstate
{
    .class abstract sealed MortgageCalculator
    {
        /* members */
    }
}
```

После имени типа у вас есть возможность задать базовый тип. Ключевое слово extend применяется именно с этой целью. Если базовый тип не задан, ассемблер IL автомтически сделает так, что тип будет наследоватья от System.Object из сборки mscorlib. И, наконец, в заголовке типа могут быть указан список интерфейсов, которые будут реализованы в типе и его потомках.

```
.class Kerr.RealEstate.RoomList
    extends [System.Windows.Forms]System.Windows.Forms.ListView
    implements Kerr.IView
{
    /* members */
}
```

В этом примере у типа Kerr.RealEstate.RoomList есть базовый тип System.Windows.Forms.ListView, заданный в сборке System.Windows.Forms. Помните, что CLI требует от каждого пользовательского типа расширять ровно один другой тип. Roomlise также реализует интерфейсный тип Kerr.IView.

С помощью этого введения, вы теперь должны суметь задавать более интересные типы. Чтобы определить интерфейс, просто используйте атрибут interface в заголовке типа. Если вам нужен тип значения, известный в C# как struct, просто расширет тип System.ValueType из сборки mscorlib. Теперь вы видите, почему название директивы .class, возножно, далеко не самое лучшее.

```
.class interface Kerr.IView
{
    /* members */
}

.class Kerr.RealEstate.HouseData
    extends [mscorlib]System.ValueType
{
    /* members */
}
```

# Unsafe Java II - Мутагенез земноводных

**Автор:** Stiver

**Дата:** 2006-09-11

Первая статья была посвящена классам, объектам и общим принципам работы с памятью виртуальной машины. Сейчас пришло время сделать еще один шаг вглубь и посмотреть на важнейшие составные части классов, а именно функции. В плане практического применения мы научимся:

• изменять байткод после загрузки
• вызывать функции, не импортируя их

Но сначала небольшое лирическое отступление. После появления "Unsafe Java I" я получил большое количество отзывов, общий смысл которых сводился к "а нафиг все это нужно?". Поэтому пользуюсь случаем подчеркнуть, что никоим образом не призываю применять ниже описанное в повседневной работе. Случаи, требующие настолько глубокого копания в виртуальной машине Явы действительно достаточно редки. Однако они существуют. И в конце концов бывает просто приятно "перелезть через забор" :)

Примеры к этой статье, за исключением простейших, оформлены для разнообразия в виде небольшого crackme. Так что имеет наверное смысл сначала попробовать разобраться с ним собственными силами, а потом уже читать "решение". Хотя особо нетерпеливые или ленивые читатели могут конечно сделать и наоборот. Для тех новичков, которые читают по-немецки: по адресу http://www.buha.info/board/showthread.php?t=52196 можно найти подробную пошаговую (не поленился же **DarkTom**, за что ему большое спасибо) инструкцию, как исследовать этот, а значит и подобные, crackme.

Все примеры протестированы с Sun Java под Windows версии от 1.4.2_08 до 1.5.0_08 включительно. К своему стыду должен признаться, что несмотря на обещание, так пока до Linux'a и не добрался. Так что у кого есть время - может заняться. Байткод примеров скомпилирован версией 1.4.2_08, другие компиляторы могут немного отличаться, что приведет к другим смещениям в коде. Соответственно если у кого-то примеры в собственной компиляции не работают, первым делом сравните получившийся байткод с моим. Что поделать, хотите работать на низком уровне - учитесь не доверять компилятору.

**1. Где живут функции** Как всегда, первым делом нужно определить, где копать. Поэтому посмотрим снова на уже знакомые нам по первой части структуры классов и поищем в них поля, отвечающие за хранение функций и кода. Так как внутренняя жизнь виртуальной машины довольно, я бы даже сказал неоправданно, часто претерпевает более или менее мелкие именения, то придется нам рассмотреть в общей сложности три случая. Опять же, здесь я приведу только те поля, которые в дальнейшем действительно понадобятся. **Версии 1.4.x, где x >7**

```
class_struct{
 ... ...
 100 functions // указатель на массив с элементами типа func_struct*
 ... ...
 128 constantpool // указатель типа constantpool_struct*
 ... ...
}

func_struct{
 0 magic
 4 class   // class_struct*
 8 constantpool  // constantpool_struct*
 12 ...
 16 strsize   // размер структуры в DWORD'ах
 18 ...
```

```
 ... ...
 32 name_index  // индекс имени функции в массиве констант
                 // (т.е. в структуре constantpool_struct)
 34 sig_index  // индекс сигнатуры (описание параметров и
                // возвращаемого значения) в массиве констант
 36 ...
 40 bc_length  // длина байткода в байтах
 ... ...
 48 invocation_counter // количество вызовов функции
 ... ...
 56 code   // псевдоуказатель на машинный код
 ... ...
 72 bytecode     // здесь начинается сам байткод
 ... ...
}

constantpool_struct{
 ... ...
 8 pool_size  // количество констант + 1
 12 tags  // constant_tags*
 16 pool_cache // указатель на кэш методов, смотри пояснения в тексте
 ... ...
 28 constants // начало массива констант
 ... ...
}

constant_tags {
 ... ...
 8 length // длина массива
 12 array // байтовый массив
 ....
}
```

Возможно кстати, что приведенные выше смещения справедливы также и для более старых версий, т.к. значения x<8 я просто не проверял.

**Версии 1.5.x, где x <6**

```
func_struct
{
 0 magic
 4 class_struct
 8 funcconst   // funcconst_struct*
 12 constantpool   // constantpool_struct*
 ... ...
 24 strsize   // размер структуры в DWORD'ax
 26 ...
 ... ...
 36 invocation_counter // количество вызовов функции
 ... ...
 44 code   // псевдоуказатель на машинный код
 ... ...
}

funcconst_struct{
 0 magic
 4 class_struct
 ... ...
 24 strsize  // размер структуры в DWORD'ax
 ... ...
 30 bc_length // длина байткода в байтах
 32 name_index // индекс имени функции в массиве констант
                // (т.е. в структуре constantpool_struct)
 34 sig_index // индекс сигнатуры (описание параметров и
               // возвращаемого значения) в массиве констант
 36 ...
 ... ...
 48 bytecode    // здесь начинается сам байткод
 ... ...
}
```

Остальные структуры совпадают с версиями 1.4.x

**Версии 1.5.x, где x >5**

```
funcconst_struct{
 0 magic
 4 class_struct
 ... ...
 32 strsize  // размер структуры в DWORD'ax
 ... ...
 38 bc_length // длина байткода в байтах
 40 name_index // индекс имени функции в массиве
                // констант (т.е. в структуре constantpool_struct)
 42 sig_index // индекс сигнатуры (описание параметров и
                // возвращаемого значения) в массиве констант
 44 ...
 ... ...
 48 bytecode   // здесь начинается сам байткод
 ... ...
}

constantpool_struct{
 ... ...
 8 pool_size  // количество констант + 1
 12 tags  // constant_tags*
 16 pool_cache // указатель на кэш методов, смотри главу 3
 ... ...
 32 constants // начало массива констант
 ... ...
}
```

Остальные структуры соответствуют предыдущему случаю 1.5.x с x<6

Внимательный читатель наверняка уже заметил, что я скачу "галопом по Европам". Неразобранными остались например constantpool_struct и constant_tags - интересные структуры с достаточно нетривиальным устройством, про кэш методов вообще по сути ничего не сказано, кроме того, что он есть. За бортом остались и отличия серверной VM, хотя все примеры написаны так, что будут работать и с ключом -server тоже. На самом деле я начал было писать полноценные объяснения, но статья быстро разрослась до совершенно неприличного размера и пришлось их снова убрать, переместив в следующую часть. Так что желающие непременно получить полную картину смотрят исходники и/или терпеливо ждут третьей части цикла.

**2. Модификация байткода**

На настоящий момент метод ClassLoader.defineClass0(...), задачей которого является физическая загрузка класса и инициализация его структур, представляет собой своеобразный переломный пункт в жизненном цикле байткода. Официально считается, что изменять загружаемый класс можно только до передачи его (в виде байтового массива) в defineClass0. Действительно, создание и редактирование классов "на лету" до загрузки - вручную или с помощью многочисленных библиотек типа BCEL - пользуется устойчивой популярностью. И при необходимости успешно отлавливаются простой заменой стандартного класса ClassLoader на свой собственный. Нас же интересует сейчас именно общий случай, то есть возможность изменения байткода в любой момент работы программы.

Как ни странно, никаких особых ухищрений с нашей стороны не потребуется. Достаточно знать где лежит код и четко представлять, в каком виде он там лежит. Сложностей с местонахождением возникнуть не должно, смотрим предыдущую главу. Придется только опять вычислять смещения в зависимости от версии Явы - Sun'овцы видимо все еще находятся в процессе творческого поиска. В отношении формата есть однако свои нюансы.

Правила ассемблирования и формат скомпилированного кода полностью описаны в VM spec, особенно обратите внимание, что все числовые значения хранятся по схеме big endian. В целом код в памяти будет таким же, как и в class файле, за одним большим исключением: ссылки на

константы классов, полей и функций в constant pool'e заменяются ссылками на кэш, то есть по сути просто на порядковый номер обьекта. Сейчас разберем на наглядном примере.

В методе main класса TestSelfmod есть строчка

```
Unsafe unsafe = UnsafeUtilEx.unsafe;
```

Компилируется она в

```
0002 B20011 getstatic #0011 <sun.misc.Unsafe crackme.UnsafeUtilEx.unsafe>
0005 4D astore_2
```

Для удобства код отформатирован так же, как показывает его JavaBite (http://www.wasm.ru/baixado.php?mode=tool&id=284). 0x0011 - номер константы (тип Fieldref) в массиве констант. Но в памяти окажется следущее:

```
B20100
4D
```

Здесь 0x01 - индекс в кэше методов. По индексу 0x00 будет лежать метод TestSelfmod.<init>, а по индексу 0x02 - поле UnsafeUtilEx.vm1_5. То есть порядок констант в constant pool'e сохраняется и в кэше.

Итак с форматом мы тоже разобрались, можно править. Просто пишем свой код поверх старого, простенький пример смотрите в классе TestSelfmod. Полезной работы здесь всего три строчки:

```
        int i = 0;
...
        i += 23;
...
        System.out.println(i);
```

По идее программа должна вывести 23 на консоль. Но с помощью

```
unsafe.putAddress(address+0x85, 0xB22A0184L);
```

мы подставляем в команду 0x840117 (т.е. iinc 1 23) число 42 (результат - 0x84012A, не забывайте про big endian) и именно его и получаем на выходе. Заметьте, что функция изменяет собственный код во время выполнения - классическая самомодификация. Еще одним примером является TestLoop, там вызванная функция изменяет вызывающую, чтобы выйти из бесконечного цикла (ifne заменяется на ifeq).

Также посмотрите на строки

```
unsafe.putAddress(key_func_data, unsafe.getAddress(key_func_data) ^ 0xEA6716E2L);
unsafe.putAddress(key_func_data+4, unsafe.getAddress(key_func_data+4) ^ 0xB21E0C9AL);
unsafe.putAddress(key_func_data+8, unsafe.getAddress(key_func_data+8) ^ 0x7F131577L);
unsafe.putAddress(key_func_data+12, unsafe.getAddress(key_func_data+12) ^ 0x003D86EFL);
```

в ValidatorMain. А когда поймете, что и как они делают, начинайте экспериментировать сами.

На самом деле конечно не все так безоблачно и при экспериментах придется учитывать еще как минимум два момента. Момент первый: весь код проходит через верификатор. Абсолютно весь, в том числе и тот, который добавляется динамически. Так что вписать на место кода всякий мусор не удастся, это должны быть более или менее осмысленные команды. Правила верификатора (почти все) описаны все в той же VM spec. Есть правда возможность отложить по времени некоторые проверки, сыграв на чрезмерном "уме" верификатора - его умении распознавать "мертвый" код. Для такого кода выполняются по-видимому только самые общие проверки и в то же время нам ничто не мешает с помощью свежеприобретенных умений в нужный момент послать его на выполнение, занопив например безусловный переход.

Кроме того не следует забывать, что Ява - язык полукомпилируемый. На некотором моменте (когда именно, зависит от настроек) байткод будет пропущен через настоящий компилятор и выполнятся будет уже машинный код. В связи с этим при изменении байткода обычно имеет смысл выставлять

значения invocation_counter и code в ноль. Если код уже прошел через HotSpot compiler, то таким образом будет произведена принудительная перекомпиляция. Если функция правится еще до ее первого вызова, сбросом результатов компиляции можно естественно пренебречь.

**3. Подмена функций**

Как известно, имена всех использующихся классов и функций лежат в скомпилированном class файле открытым текстом. Поэтому первым инструментом при исследовании написанных на Яве программ и является не какой-нибудь хитрый декомпилятор, а обыкновенный grep. Присутствие java.io.* выдает работу с файлами, BigInteger.modPow может быть признаком RSA и так далее. Для разработчиков этот факт естественно неприятен. Встает вопрос, можно ли вызвать функцию так, чтобы её не было видно в импорте?

Первое что приходит в голову - использовать reflection. Самой функции тогда действительно не будет видно. Беда в том, что вместо нее появится пакет java.lang.reflect, в частности например Method.invoke, который не менее безошибочно обратит на себя внимание. Поэтому уточним задачу: нужно вызвать функцию так, чтобы вызывающий класс остался "чистым". Импортирование Unsafe естественно тоже запрещено.

Разберемся сначала с теорией. Как уже упоминалось выше, вызов функции в байткоде осуществляется по ее порядковому номеру в кэше. По этому индексу в кэше лежит в свою очередь указатель на соответствующую func_struct (строго говоря не только и не всегда, но в самые тонкости лезть не будем). Чтобы переправить функцию A на B, достаточно подменить этот самый указатель, и тогда последующие обращения к A будут на самом деле вызывать B, несмотря на то, что код останется прежним. Тот же самый алгоритм будет работать кстати и в отношении открытых переменных (полей) классов.

Допустим, мы по каким-либо причинам хотим спрятать возведение числа в степень, то есть вызов Math.pow(...). Давайте подумаем и распишем необходимые для этого действия. Во-первых нужен конечно подходяший вызов какой-нибудь функции A, которую нам не жалко подменить. С этим сложностей не предвидится - можно взять из стандарных библиотечных, можно просто определить собственную. Значительно сложнее и интересней будет вторая половина задачи, т. е. получение указателя на Math.pow. Чтобы найти указатель в массиве class_struct.functions нужно для начала загрузить класс Math (нигде не сказано, что он уже присутствует в системе) и получить указатель на его class_struct. Так как Math не должен присутствовать в импорте, то загружать придется через Class.forName("java.lang.Math").

Все бы хорошо, но тогда в импорте окажется Class.forName. Она правда встречается на порядок чаще, чем reflection и особых подозрений вряд ли вызовет, но как известно лучше уж перестараться, чем наоборот. Поэтому вызов Class.forName мы тоже спрячем по описанной выше схеме, с той только разницей, что Class уже загружен и получить его структуру - задача тривиальная. План действий выглядит в итоге следующим образом:

1. Определение "ненужных" функций A и B
2. Получение указателя на Class.forName
3. Перенаправление B на Class.forName
4. Вызов B с параметром "java.lang.Math"
5. Получение указателя на Math.pow
6. Перенаправление A на Math.pow
7. Вызов A

Теперь проследим эти теоретические построения в коде. ValidatorMain.wktm станет у нас Class.forName, а KeyValidator.edtb сыграет почетную роль Math.pow. Вспомогательная функция ValidatorMain.atbz возьмет на себя задачу, которую мы до сих пор в наших рассуждениях ловко обошли молчанием, а именно собственно поиск нужного указателя в массиве functions. Казалось бы

тривиальное действие, но дело в том, что определенного порядка в расположении функций судя по всему не существует, он меняется от версии к версии практически случайным образом и задается похоже методом среднепотолочного тыка. Посмотрите например на безобразную конструкцию в TestLoop.doSomething, где мы имеем дело всего с двумя функциями и я жестко задал их номер в массиве. Поэтому остается только громоздкий поиск по имени и сигнатуре, что и проделывает ValidatorMain.atbz.

Отсюда по шагам, сверяйтесь с кодом примеров. Получаем указатель на Class.forName:

```
long class_struct = unsafe.getAddress(unsafe.getAddress(this_struct+15*4)+4);
long func_data = atbz(class_struct, 0x4E726F66, 0x25);
```

Пишем его в кэш на место ValidatorMain.wktm и загружаем Math:

```
unsafe.putAddress(this_const_cache+33*4, func_data);
Object obj = wktm(s);
```

С целью немного усложнить crackme строка s тоже собирается динамически, но это уже мелочи. Получаем указатель на Math.pow:

```
long math_struct = unsafe.getAddress(UnsafeUtilEx.ObjectToAddress(v)+8);
long math_func_data = atbz(math_struct, 0x00776F70, 0x5);
```

Заменяем KeyValidator.edtb на Math.pow:

```
unsafe.putAddress(k_const_cache+13*4, math_func_data);
```

Теперь вызов edtb(name, i) в KeyValidator.mshv возведет name в степень i. Причем импорт класса KeyValidator остался совершенно "чистым".

**4. Заключение** Не исключено, что при применение описанных приемов в "промышленном масштабе" придется учитывать ряд дополнительных, не упомянутых здесь тонкостей. Некоторое представление о них можно получить отсюда: http://www.cracklab.ru/f/index.php?action=vthread&topic=5363&forum=2&page=-1 С другой стороны большинство замечаний **bsl_zcs** носят опять же чисто академический характер: опции -XX: точно так же недокументированы, как и Unsafe, и систем скажем с -XX:CompileThreshold=2 мне еще никогда не встречалось и очень вряд ли встретятся.

До сих пор я цитировал только отдельные поля из структур виртуальной машины. В следующей, третьей, части я приведу полные (по возможности :)) структуры с описанием.

**Приложения**

unsafejava2code.zip (находится в архиве `wasm_files.zip`: `files/0431/unsafe_java_2_code.zip`) (8 KB) - Примеры к статье

javacrackme.zip (находится в архиве `wasm_files.zip`: `files/0431/java_crackme.zip`) (4 KB) - Crackme

**(c) Stiver**, 09.09.2006

# Введение в MSIL - Часть 4: Определение членов типа

**Автор:** Кенни Керр, пер. Aquila

**Дата:** 2006-09-21

В 3-ей части данного введения я рассказал о синтаксисе определения типов. Используя директиву .class, вы можете задать ссылочные типы и типы значений. Правильно выбирая атрибуты типа, вы можете получить полный контроль над его определением.

```
.class abstract Kerr.Sample.Object
{
}
```

В этой главе мы узнаем как объявлять члены типа. Для упрощения примеров я иногда буду опускать определения классов, фокусируя внимание на задании членов.

**Конструкторы**

Давайте начнём с инициализаторов, известных в таких языках как C++ и C# как конструкторы. CLI поддерживает как инициализаторы типа, так и инициализаторы экземпляра. Инициализаторы типа очень удобная вещь, помогающая справиться со многими мультитредовыми проблемами, с которыми вы можете столкнуться, реализуя синглетон в "родном" (native) C++. Инициализатор типа выполняется только один раз для данного типа и среда выполнения гарантирует, что ни один метод не будет доступен, пока конструктор не завершит работу.

```
.method static void .cctor()
{
    .maxstack 1

    ldstr ".cctor"
    call void [mscorlib]System.Console::WriteLine(string)

    ret
}

.method public void .ctor()
{
    .maxstack 1

    ldarg.0
    call instance void [mscorlib]System.Object::.ctor()

    ldstr ".ctor"
    call void [mscorlib]System.Console::WriteLine(string)

    ret
}
```

.cctor и .ctor являются специальными именами метода. Инициализатор типа, который является необязательным, задаётся в качестве статического метода .cctor без возвращаемого значения. В C++/CLI и C# инициализаторы типа называются статическими конструкторами.

Инициализатор экземпляра обычно называют просто конструктором. Он задаётся как метод .ctor также без возвращаемого значения. Конструктор вызывается, когда экземпляр типа создаётся с помощью инструкции newobj. Вот пример:

```
.locals (class TypeName obj)
newobj void TypeName::.ctor()
stloc obj
```

Вышеприведённые конструкторы, будучи выполненными, выведут на экран следующие строки:

```
.cctor
.ctor
```

Инструкция newobj создаёт экземпляр типа и инициализирует все его поля, присваивая им значения, эквивалентные нулю. Затем она вызвает соответствующий конструктор, передавая ему первым аргументом ссылку на созданный экземпляр. Как только конструктор завершает работу, ссылка на объект помещается в стек.

**Методы**

Хотя CLI задаёт конструкторы и свойства как методы (с дополнительными метаданными для свойств), в этой секции мы будем рассматривать методы, как они понимаются в C++ или C#. Конечно, большая часть того, что связанно с методами, также применима к конструкторам и функциям установки/считывания свойств. Большая часть того, что вы можете делать с метода задаётся атрибутами метода. Давайте рассмотрим некоторые из них.

Статические методы задаются с помощью атрибута static. Статические метода, как вы, вероятно, и ожидаете, ассоциируются с типом, но не с экземпляром.

```
.method static void StaticMethod() { /* impl */ }
```

Методы экземляра задаются через атрибут instance вместо static. Ассемблер IL присваивает этот атрибут по умолчанию, поэтому вам редко понадобится указывать его явно в определении метода.

```
.method void InstanceMethod() { /* impl */ }
```

Обратно верно и для вызова методов. Инструкция call по умолчанию предполагает, что метод статический, если не указано обратное. Вот пример для вызова обоих вышеуказанных методов:

```
call void TypeName::StaticMethod()

ldloc obj
call instance void TypeName::InstanceMethod()
```

Не забывайте помещать ссылку на экземпляр в стек перед вызовом его метода.

Вызовы виртуальных методов являются важной часть объектно-ориентированного дизайна, и CLI предоставляет значительную гибкость в том, статический или динамический тип объекта будет использован для обслуживания вызова, так же как и в том, каким образом данного поведение может быть переопределно в классах-наследниках (говоря о статических и динамических типах я имею в виду то, как эти понятия понимаются в C++: статические - известные во время компиляции, а динамические - известные во время выполнения). Это обычно называется полиморфизмом. Программируя на MSIL, вам нужно держать в уме два аспекта, связанных с виртуальными функциями. Первый - это как объявлять метод экземпляра, чтобы он поддерживал виртуальный вызов, а второй - как именно вызывать метод. Очевидно, что статические методы по определению не могут быть виртуальными.

Метод становится виртуальным с помощью добавления атрибута virtual.

```
.class House
{
    .method public virtual void Buy()
    {
        .maxstack 1

        ldstr "House::Buy"
        call void [mscorlib]System.Console::WriteLine(string)

        ret
    }

    /* etc */
}
.class TownHouse
```

```
    extends House
{
    .method public virtual void Buy()
    {
        .maxstack 1

        ldstr "TownHouse::Buy"
        call void [mscorlib]System.Console::WriteLine(string)

        ret
    }

    /* etc */
}
```

У типа House есть виртуальный метод Buy. Тип TownHouse расширяет House, и у него также есть виртуальный метод с таким же именем, поэтому говорят, что TownHouse::Buy переопределяет House::Buy. Как же мы указываем среде выполнения, какой метод выбрать. Очевидно, что если у меня экземпляр House, то я хочу вызвать House::Buy, однако если у меня экземпляр TownHouse, то мне нужно, чтобы это решение принималось во время выполнения, основываясь на типа экземпляра.

Пока что я использовал инструкцию call, которая всегда вызывает указанный метод, независимо от динамического типа. Инструкция callvirt, напротив, позволяет среде выполнения определить, какую именно реализацию виртуального метода вызывать, основываясь на типе экземпляра. Посмотрите следующий пример:

```
newobj instance void House::.ctor()
stloc house
newobj instance void TownHouse::.ctor()
stloc townHouse

ldloc house
call instance void House::Buy()

ldloc townHouse
call instance void TownHouse::Buy()

ldloc townHouse
call instance void House::Buy()

ldloc townHouse
callvirt instance void House::Buy()
```

Будучи выполненным, он выдаст на консоль следующее:

```
House::Buy
TownHouse::Buy
House::Buy
TownHouse::Buy
```

Первое обращение к методу Buy с сслыкой на house вызывает реализацию House::Buy, так как инструкция call интересуется только статическим типом. Второе обращение к Buy с ссылкой на townhouse вызовет реализацию TownHouse::Buy по той же причине. Третье обращение снова вызывает House::Buy несмотря на то, что объект, на который даётся ссылка принадлежить к типу TownHouse. Должно быть понятно, что использование инструкции call подразамевает выбор того, какой метод вызывать, только во время компиляции. В последнем случае используется инструкция callvirt для вызова виртуального метода House::Buy, и так как ссылка указывает на TownHouse, то используется TownHouse::Buy.

Если вы хотите объявить виртуальный метод, но не хотите переопределять унаследованный виртуальный метод с таким же именем, вы можете использовать атрибут newslot при объявлении нового виртуального метода в субклассе. Если вы учтёте, что вызов виртуальных методов средой выполнения не привязан к имена, тогда вы должны понять, как это возможно. Просто думайте о

newslot как о добавлении нового указателя на виртуальную функцию в vtbl данного типа.

Виртуальные методы CLI очень интересны, особенно, когда видишь, в какой степени они используются C++/CLI и C#.

# Введение в MSIL - Часть 5: Обработка исключений

**Автор:** Кенни Керр, пер. Aquila

**Дата:** 2006-09-25

В этой части "Введения в MSIL" я расскажу о конструкциях, которые CLI предоставляет для обработки исключений.

Блок try используется для защиты блока инструкций. Если одна из них бросает исключение или один из методов явно или неявно, вызываемых в защищённом блоке, то контроль передаётся соответствующему обработчику инструкций. Блок try объявляется с помощью директивы .try.

```
.try
{
    /* защищённый код */

    leave.s _CONTINUE
}
<exception handler>

_CONTINUE:
```

Последняя инструкция в блоке try - это leave.s, которая передаётся контроль код с меткой _CONTINUE. Эта инструкция гарантирует, что стек будет очищен, а соответствующие блоки finally будут выполнены. Из этого следует, что если выход из защищённого блока произошёл каким-либо другим образом, то было брошено исключение. Обработчик исключения должен следовать непосредственно за блоком try.

CLI предлагает четыре разных типа обработчика, из которых вы можете выбирать в зависимости от вашего языка или приложения. Давайте рассмотрим каждый из них.

**Catch**

Обработчик catch или блок catch является одним из самых известных, так как он напрямую предоставляется как C++, так и C#. Этот блок объявляется с помощью ключевого слова catch, вместе с которым задаётся тип исключения, для которого предназначается данный обработчик, а также сам код, которому передаётся контроль. Catch-блоки также для удобство могут быть объединены в цепь и использовать один блок try. Посмотрите следующий пример:

```
.try
{
    ldstr "I'm not a number"
    // ldnull
    // ldstr "123"
    call int32 [mscorlib]System.Int32::Parse(string)

    leave.s _CONTINUE
}
catch [mscorlib]System.ArgumentNullException
{
    callvirt instance string [mscorlib]System.Exception::get_Message()
    call void [mscorlib]System.Console::WriteLine(string)

    leave.s _CONTINUE
}
catch [mscorlib]System.FormatException
{
    callvirt instance string [mscorlib]System.Exception::get_Message()
    call void [mscorlib]System.Console::WriteLine(string)

    leave.s _CONTINUE
}
```

Здесь мы просим метод Int32::Parse обработать "I'm not a number", что очевидным образом вызывает исключение FormatException. Первый catch-обработчик никогда не запускается. Обработчик FormatException послушно пишет в консоль сообщение, описывающее ошибку, а затем вызывает инструкцию leave.s, чтобы передать контроль коду по метке _CONTINUE. Где находился объект исключения? Среда выполнения гарантирует, что ссылка на исключения будет помещена в стек прежде, чем будет вызван обработчик. Вы можеет поэкспериментировать, закомментировав инструкцию ldstr и раскомментировав ldnull. Это поместит в стек null-ссылку, из-за чего вознинет исключение ArgementNullException.

**Filter**

Для C++-программиста filter-обработчик является довольно странной конструкцией. Данный обработчик предоставляет блок видимости, где он может решить, хочет ли он обрабатывать исключение. Если да, то в стек помещается значение 1, в противном же случае - 0. Далее следует блок обработки, где, собственно, и находится код, обрабатывающий исключение.

```
.try
{
    // ldstr "I'm not a number"
    ldnull
    // ldstr "123"
    call int32 [mscorlib]System.Int32::Parse(string)

    leave.s _CONTINUE
}
filter
{
    ldstr "filter evaluation\n\t"
    call void [mscorlib]System.Console::Write(string)

    callvirt instance string [mscorlib]System.Exception::get_Message()
    call void [mscorlib]System.Console::WriteLine(string)

    ldc.i4.1
    endfilter
}
{
    ldstr "filter handler\n\t"
    call void [mscorlib]System.Console::Write(string)

    callvirt instance string [mscorlib]System.Exception::get_Message()
    call void [mscorlib]System.Console::WriteLine(string)

    leave.s _CONTINUE
}
```

Try-блок вызовет исключение ArgumentNullException, так как в стек в качестве аргумента метода Int32::Parse была помещена пустая ссылка.

Фильтру даётся шанс определить, будет ли он обрабатывать исключение или нет. В данном случае он просто пишет сообщение об ошибке и помещает в стек значение 1, используя инструкцию ldc.i4.1, чтобы указать, что он собирается обрабатывать исключение. Инструкция endfillter вызывается, чтобы возвратиться из фильтра.

Затем вызывается блок обработки. Обратите внимание, что как один, так и другой блок могут обратиться к исключению на лету, взяв соответствующий объект из стека. Держите в уме, что могут быть брошены ссылки на объекты различных типов, поэтому не ожидайте, что ссылка на объект, которую вы достанете из стека в блоке обработки обязательно будет экземпляров System.Exception. Это не проблема для catch-обработчика, так как вы уже знаете тип исключения.

**Finally**

Finally-обработчик исключения должны узнать C#- и C++-программисты, знакомые с Structured Exception Handling (SEH). Блок finally, ассоцированный с блоком try выполняется всегда, вне

зависимости от того, передаёт ли последний контроль нормальным образом или в результате исключения. Это предоставляет надёжный механизм для того, чтобы гарантировать выполнение определённого блока кода для языков, которые не предоставляют деструкторы, такие как C и C#. Хотя язык C не использует CLI, SEH часто применяется, и слово __finally, предоставляемое компилятором Microsoft C/C++, используется именно для этой цели.

```
.try
{
    /* защищённый код */

    leave.s _CONTINUE
}
finally
{
    /* очищающий код */

    endfinally
}
```

**Fault**

Fault-обработчик исключения похож на finally, не считая того, что он запускается только если в ассоциированном с ним блоке try произошло исключение. После того, как отрабатывает fault-обработчик, исключение продолжает искать соответствующий catch-обработчик.

```
.try
{
    /* защищённый код */

    leave.s _CONTINUE
}
fault
{
    /* очищающий код */

    endfault
}
```

Теперь настало время сменить тему и в следующей части мы поговорим о том, как всё это соотносится с такими популярными языками программирования, как C# и C++/CLI.

# Введение в MSIL - Часть 6: Стандартные языковые конструкции

**Автор:** Кенни Керр, пер. Aquila

**Дата:** 2007-10-08

В части с 1 по 5 "Введения..." мы рассматривали MSIL и конструкции, предоставляемые им для написания управляемого кода. В следующих секциях мы изучим некоторые стандартные языковые возможности, которые не являются особенностями подобных MSIL'у языков, основанных на инструкциях. Иметь представление, какой код генерится для обычных конструкций языка программирования очень важно, чтобы понимать, как улучшить качество вашего кода и быстрее находить трудноотлавливаемые баги.

В этой части мы рассмотрим несколько стандартных конструкций, которые встречаются во многих языках. Я буду приводить примеры на C#, хотя соответствующие объяснения применим ко многих языкам программирования, имеющим эквиваленты этих конструкций. Так как я буду стараться объяснить общий принцип, то не всегда могу предоставить точный код, который сгенерирует компилятор. Идея в том, чтобы лучше понять, что как он работает. Я призываю вас сравнивать инструкции, генерируемые разными компиляторам с помощью таких инструментов как ILDASM.

Такие языки программирования как C и C++ позволяют более быструю разработку программного обеспечения в сравнении с классическим программированием на ассемблере. Значительное количество языковых средств играет в этом немалую роль. Такие концепции как выражения, условные конструкции (например, **if** и **switch**), а также циклы (**while** и **for**) - это то, что делает эти портабельные языки такими мощными. Ещё большая скорость была достигнута с помощью новых концеций вроде объектов, но прежде чем они были признаны, конструкции, упомянутые выше, дали огромное преиущество ассемблерному программисту, которому требовалось писать большие приложения и операционные системы. Теперь, давайте, перейдём к делу. **Конструкция if-else**

Следующий пример проверяет перед выполнением основного кода, не равен ли аргумент null.

```
void Send(string message)
{
    if (null == message)
    {
        throw new ArgumentNullException("message");
    }

    /* impl */
}
```

Конструкция **if** выглядит довольно безобидно. Если выражение равно true, управление переходит к блоку внутри фигурных скобок и бросается исключение. Если выражение равно false, управление переходит к следующей строке после конструкции **if**. Описание **else** я оставляю вам в качестве домашнего упражнения.

Даже если вы никогда не использовали C# раньше, этот код для вас должен быть вполне ясен (если вы программист). Тогда как компилятор превращает эту несложную, маленькую конструкцию **if** в что-то понимаемое средой выполнения? Как и случае с большинством программистских задач, есть несколько путей решить проблему.

Во многих случаях компилятору необходимо создавать временные объекты. Обычный их источник - это выражения. На практике компиляторы избегают создание временных объектов настолько, насколько это возможно. Гипотетический неоптимизирующий компилятор превратит код выше в что-то вроде следующего:

```
bool isNull = null == message;
```

```
if (isNull)
{
    throw new ArgumentNullException("message");
}
```

Я не хочу сказать, что оптимизирующий компилятор не будет использовать временные объекты подобным образом. Они могут часто помочь сгенерировать более эффективный код - это зависит от обстоятельств. Здесь обрабатывается результат выражения, который вначале преобразуется в булевое значение, передающееся затем выражению **if**. Это ещё не все отличия от предыдущего примера, но после прочтения последних 5 частей этого цикла, должно быть ясно, как это применяется в языке вроде MSIL. Вся идея в том, чтобы разбить конструкции и выражения на простые инструкции, которые могут быть выполнены один за другим. Давайте рассмотрим одну из реализаций метода Send.

```
.method void Send(string message)
{
    .maxstack 2

    ldnull
    ldarg message
    ceq

    ldc.i4.0
    ceq

    brtrue.s _CONTINUE

    ldstr "message"
    newobj instance void [mscorlib]System.ArgumentNullException::.ctor(string)
    throw

    _CONTINUE:

    /* impl */

    ret
}
```

Если вы изучили материал из прошлых глав цикла, то должны многое понять из объявления метода. Давайте рассмотрим его по-быстрому. Временный объект, который может молча сгенерировать компилятор, явно объявлен в MSIL'e, хоть он и безымянный и проживёт очень короткую жизнь в стеке. (Думаю, явный он или нет - вопрос спорный). Можете ли вы его найти? Сначала мы сравниваем сообщение с null. Инструкция **ceq** берёт два значения из стека, сравнивает их, а затем помещает на стек 1, если они равны и 0, если нет (это временно). Код может показаться излишне усложнённым. Причина этого состоит в том, что в MSIL'e нет cneq, то есть "compare-not-equal-to", операции противоположной предыдущей. Поэтому мы сначала сравниваем сообщение с null, а затем полученный результат с нулём, таким образом инвертируя предыдущее сравнение.

Теперь, когда мы получили результат выражения, инструкция **brtrue.s** делает примерно тоже, что и конструкция **if**. Она передает управление на заданную метку, если верхний элемент стека не равен нулю, пропуская в этом случае связанный с условием блок кода.

Конечно, это не единственный путь. Вышеприведённый пример очень похож на то, что генерирует мой компилятор C# с той разницей, что он использует локальную переменную для сохранения результата выражения. Эта реализация кажется немного неуклюжей. Как правило, это не играет роли, так как JIT-компилятор, вероятно, нивелирует эту разницу в результате компиляции. Тем не менее, будет полезно потренироваться, упростив эту реализацию. Я упомянул, что нет инструкции "cneq". С другой стороны, есть обратная версия инструкции **brtrue.s**. Как и следовало ожидать, она называется **brfalse.s**. Используя её, можно полностью убрать необходимость во втором сравнении.

Наконец, как C++-программист вы могли бы ожидать, что компилятор использует оператор равенства, так как один из операндов является типом System.String, для которого такой оператор определён, и это именно то, что делает C++-компилятор.

```
.method void Send(string message)
{
    .maxstack 2

    ldnull
    ldarg message
    call bool string::op_Equality(string, string)

    brfalse.s _CONTINUE

    ldstr "message"
    newobj instance void [mscorlib]System.ArgumentNullException::.ctor(string)
    throw

    _CONTINUE:

    /* impl */

    ret
}
```

JIT-компилятор оптимизирует код настолько хорошо, насколько может, применяя по возможности инлайн-методы. Вас не должно удивлять, что две разные реализации могут приводить к одному и тому же машинному коду.

**Конструкция for** Прежде чем мы перейдём к реализации этой конструкции, которую часто называют "циклом", давайте бысто проведём обзор того, как она работает. Если вашим основным занятием является Visual Basic или даже C#, то вы можете быть вовсе незнакомы с этой очень полезной конструкцией. Даже C++-программистам рекомендуют по возможности избегать её в пользу более безопасного алгоритма std:: **for** _each из стандартной библиотеки C++. Тем не менее, у цикла есть много интересных применений, которые не имеют ничего общего с пробегом по контейнеру от начала до конца.

Следующий просто псевдокод демонстрирует данную конструкцию.

```
for ( начальное выражение ; выражение условия ; выражение цикла )
{
    код
}
```

Она состоит из трёх выражений, разделённых точкой с запятой, за которыми следует связанный блок кода. Вы можете использовать любое выражение, дающее результат, который может быть интерпретирован как истина или ложь. Первое выражение выполняется в самом начале и только один раз. Оно обычно используется для инициализации индексов цикла или других переменных, которые могут потребоваться данной конструкции. Выражение условия запускается следующим и выполняется перед каждой итерацией. Если его результат истинен, то выполняется прикреплённый блок код, если нет, то управление передаётся на первую строку за ним. Выражение цикла выполняется после каждой итерацией. Оно может использоваться для увеличения значения индексов цикла, движения курсоров или чего угодно другого, подходящего по контексту. После этого снова выполняется выражение условия и так далее. Для более подробного разъяснения обратитесь к документации по вашему языку программирования. Вот простой пример, который отображает числа от нуля до девяти в отладочном режиме.

```
for (int index = 0; 10 != index; ++index)
{
    Debug.WriteLine(index);
}
```

Мы уже говорил о том, как конструкция **if** может быть реализована с помощью MSIL. Давайте теперь используем это знание, чтобы разобрать **for** на более мелкие конструкции, которые могут быть интерпретированы компьютером, пока что используя C#.

```
    int index = 0;
    goto _CONDITION;

_LOOP:

    ++index;

_CONDITION:

    if (10 != index)
    {
        // for statements
        Debug.WriteLine(index);

        goto _LOOP;
    }
```

Это выглядит ничем иным, как конструкцией **for**! Здесь я использую печально известный goto, который чаще упоминают как инструкцию ветвления. Она неизбежна в языках, которые не поддерживают конструкции выбора и цикла, но не имеет значительного смысла в таких языках как C++ и C#, кроме, разве что, запутывания кода. (Если вы сторонник использования данной конструкции, то, пожалуйста, не надо делиться этим со мной. Напишите собственную статью о её достоинствах.) Кодогенераторы, тем не менее, довольно часто её используют, так как для них использовать эту конструкцию гораздо легче, нежели, например, **for**.

Из примера выше вы можете понять, что мы реализуем конструкцию **for**, сначала проверяя условие, а затем переходя к выражению цикла, чтобы сделать следующую итерацию. Теперь давайте посмотрим, как мы можем это реализовать в MSIL.

```
    .locals init (int32 index)
    br.s _CONDITION

_LOOP:

    ldc.i4.1
    ldloc index
    add
    stloc index

_CONDITION:

    ldc.i4.s 10
    ldloc index
    beq _CONTINUE

    // for
    ldloc index
    box int32
    call void [System]System.Diagnostics.Debug::WriteLine(object)

    br.s _LOOP

_CONTINUE:
```

Инициализовав локальную переменную index мы переходим к метке _CONDITION. Чтобы выполнить условие, я поместил в стек значение 10, за которым последовало значение index. Инструкция **beq** (или branch on equal) достаёт из стека два значения и сравнивает их. Если они рвны, то она передает контроль коду по метке _CONTINUE, заканчивая, таким образом, цикл. В противном случае контроль передаётся блоку кода, относящемуся к циклу. Чтобы послать значение index в вывод отладки, я поместил его в стек, использовал box и вызвал статический метод WriteLine ссылочного типа System.Diagnostics.Debug из сборки System. После этого используется

инструкция br.s для передачи управления выражению цикла для проведения следующей итерации.

# Введение в MSIL - Часть 7: Приведение и преобразование типов

**Автор:** Кенни Керр, пер. Aquila

**Дата:** 2008-02-22

Приведение и преобразование типов часто вызывает различные беспокойства среди программистов: например, по поводу скорости выполнения, безопасности или просто не совсем хорошее понимание последствий данных операций. В этой главе цикла я расскажу о них. Для примеров я буду использовать в основном C++, так как он имеет богатый набор языков средств, относящихся к приведению типов. Конечно, мы также будем рассматривать инструкции CLI, которые генерируются в итоге.

**Статическое приведение типов**

Когда происходит преобразование типа меньшего размера в тип большего размера, то можно использовать неявное приведение, так как нет угрозы потери данных. Это безопасно, пока используемое целевым типом пространство является надмножеством пространства исходного типа. Конвертация из 32-х битного целого числа в 64-х битное целое число будет точным, поэтому компиляторы не требуют от программиста непосредственного предупреждения. С другой стороны, когда просходит преобразование из большего типа в меньший, возможно обрезание данных, так что компилятору обычно требуется подтверждение в виде приведения типа. Это называется статическим приведением типов, так как для определения вида конвертации используется только статичная информация. Такое приведение часто считается небезопасным, так как компилятор полностью возлагает правильность приведения на программиста. Рассмотрим следующий C++-код:

```
Int32 small = 123;
Int64 big = small;

small = static_cast<Int32>(big);
```

Преобразование из меньшей переменной в большую является неявным, но обратная операция требует оператора static_cast, чтобы избежать предупреждения компилятора о возможной потере данных. Хотя статическое приведение типов может и должно считаться небезопасным, компилятор по мере возможностей пытается убедиться, что полученное значение будет правильным во время выполнения. Он делает это, оценивая информацию о статическом приведении. В случае с типами, заданными пользователем, он пытается определить, заданы ли какие-нибудь операторы преобразования. Все они задаются во время компилирования. Рассмотрим следующую реализацию в виде MSIL:

```
.locals init (int32 small,
              int64 big)

// Int32 small = 123;
ldc.i4.s 123
stloc small

// Int64 big = small;
ldloc small
conv.i8
stloc big

// small = static_cast<Int32>(big);
ldloc big
conv.i4
stloc small
```

Инструкция ldc.i4.s помещает 4-х байтное (32-х битное) число 123 в стек. Затем оно зачастую сохраняется в маленькую локальную переменную, используя инструкцию stloc. Чтобы присвоить значение маленькой переменной большой, значение сначала помещается в стек с помощью инструкции ldloc. Затем инструкция conv.i8 преобразует значение в 8-ми байтное (64-х битное) число, после чего оно забирается из стек и сохраняется в большой локальной переменной, используя инструкцию stloc. Наконец, чтобы преобразовать значение большой переменной обратно в маленькую, значение первой помещается в стек, а затем 8-ми байтное значение на стеке конвертируется в 4-х байтное посредством инструкции conv.i4, после чего результат с помощью stloc сохраняется в маленькой локальной переменной.

Как вы можете видеть, MSIL не делает различия между явным и неявным приведением типов. Всё является явным. Тем не менее, защита от переполнения должна запрашиваться отдельно, так как по умолчанию не производится никакой проверки. В вышеприведённом коде этого не делалось. Код является безопасным, потому что мы знаем, что это так, но если бы значение, сохранённое в большой переменной было бы слишком велико для 4-х байтного числа, то переполнение произошло бы без каких-либо предупреждений во время выполнения.

**Преобразование с проверкой**

Чтобы обнаружить ошибки переполнения, вы можете просто заменить инструкцию conv.i4 в предыдущем примере на conv.ovf.i4. Если значение на стеке слишком велико, чтобы быть представленным указанным типом, то будет брошен объект OverflowException. Дизайн языка C++/CLI, представленный в Visual C++ 2005 ещё не предоставляет языковых возможностей для использования преобразования с проверкой на переполнение. Есть вероятность, что она будет добавлена в виде ключевого слова checked, также как в C#:

```
checked
{
    int small = 123;
    long big = small;

    small = (int) big;
}
```

Все арифметические операции, преобразования и выражения в блоке checked будут включать проверку на переполнение. Как указано выше, множество инструкций conv.<type> заменяется на conv.ovf.<type>. Другие инстуркции, которые могут привести к переполеннию, также имеют соответствующие аналоги. Например, инструкции add, описанная в 2-ой части, соответствует add.ovf и add.ovf.un (сложение с проверкой на переполнение - со знаком и без соответственно).

**Динамическое приведение типов**

Статическое приведение может использоваться и для приведения классов, например, выполняя приведение полиморфного типа, если при этом делается предположение, что такое приведение будет всегда совершаться успешно во время выполнения. Полезно избегать накладных расходов, связанных с проверкой типов во время выполнения, но как правило, такое предположение весьма опасно.

Динамическое приведение типов используется, когда нужно отложить определение того, можно ли сделать данное приведение, до времени выполнения. Конечно, это имеет смысл только с полиморфными типами. Как и в случае с статическим приведением типов, компилятор пытается удостовериться, что производимая операция хотя бы правдоподобна. Попытка динамического приведения целого числа к объекту House будет отловлена компилятором как ошибка. Рассмотрим следующий C++-код, исходя из того, что BeachHouse и Townhouse - это подклассы House.

```
House^ house = gcnew BeachHouse;

if (nullptr != dynamic_cast<BeachHouse^>(house))
{
```

```
    Console::WriteLine("It's a beach house!");
}

if (nullptr != dynamic_cast<Townhouse^>(house))
{
    Console::WriteLine("It's a townhouse!");
}
```

Этот код прекрасно скомпилируется, и компилятор так и не узнает, пройдут ли приведения типов. В обоих случаях требуется проверка во время выполнения, чтобы определить, является ли дом BeachHouse'ом или Townhouse'ом. dynamic_cast возвращает nullptr, C++-представление для null-указателя или хэндла, если приведение не может быть выполнено. Давайте посмотрим, как это можно сделать в MSIL:

```
.locals init (class House house)

newobj instance void BeachHouse::.ctor()
stloc house

ldloc house
isinst BeachHouse
ldnull

beq.s _TOWN_HOUSE

ldstr "It's a beach house!"
call void [mscorlib]System.Console::WriteLine(string)

_TOWN_HOUSE:

ldloc house
isinst TownHouse
ldnull

beq.s _CONTINUE

ldstr "It's a town house!"
call void [mscorlib]System.Console::WriteLine(string)

_CONTINUE:
```

Если вы дочитали до этой главы, то код должен быть вам понятен. Создаётся объект BeachHouse, а ссылка на него сохраняется как локальная переменная. Динамическое приведение выполняется путём помещения ссылки в стек и использования инструкции isinst для проверки типов. Инструкция isinst определяет, является ли ссылка на стек BeachHouse'ом или подклассом BeachHouse. Если да, тогда ссылка приводится к типу, заданному инструкцией isinst и помещается на стек, в противном случае в стек будет помещено значение null. Выражение для if конструируется помещением null-ссылки в стек и использованием инструкцией beq.s, чтобы передать управление цеи, если isinst также поместила null в стек. Такая же проверка типов и условное ветвление сделаны для типа Townhouse. При этом выполнение продолжается несмотря ни на что.

Если ваше приложение предполагает, что динамическое приведение должно всегда оканчиваться успешно, но вы не хотите прибегать к статическому приведению и ловить исключения, вы можете использовать инструкцию castclass вместо isinst. Она выполняет такую же проверку типов, но вместо помещения в стек null, если проверка не удалась, она бросает объект InvalidCastException. Если вам нужно именно это, то в C++ есть оператор safe_cast или простое (в стиле C) приведение типов в C#.

# Введение в MSIL - Часть 8: Конструкция for each

**Автор:** Кенни Керр, пер. Aquila

**Дата:** 2008-05-23

Конструкция for each: приобрёвшая популярность благодаря Visual Basic, едва признанная C++ и ставшая бессмертной из-за ECMA-334 (некотоыре люди называют это просто C#).

В этой главе я будую говорить об одном из самых популярных конструкций в коммерчески успешных языка. Будете ли программировать на Visual Basic, COM, Standard C++ или .NET (или на всех вышеперечисленных), вы неизбежно встретитесь с какой-либо формой этой конструкции.

Вы можете заметить определённую тенденцию в этом цикле. По мере того, как мы начинаем копать более интересные вещи, фокус всё больше смещается с собственно инструкций MSIL на дизайн языка и реализации компилятора. В инструкциях, с помощью которых реализуется конструкция for each, нет ничего особенно нового. Тем не менее, полезно понимать, как она работает и в какие команды MSIL в каком контексте она переводится. Вы можете поблагодарить вашего начальника, что он позволяет вам писать код на языке, где конструкции абстрагированы от грязных секретов, скрывающихся за их красивым фасадом. for each - как раз такая инструкция.

Те или иные варианты for each, когда дело касается .NET, могут быть найдены в C#, Visual Basic .NET и C++/CLI. Могут быть и другие языки, мне неизвестные, где применяются похожие конструкции. В примерах этой главы будет использоваться только C++/CLI, так как мы уже достаточно много говорили о C# и VB. Как я сказал ранее, я не хочу сосредотачиваться исключительно на одной реализации компилятора, но скорее продемонстрировать техники, которые могут быть использованы компиляторами в общем случае.

Давайте начнём с простого примера.

```
array<int>^ numbers = gcnew array<int> { 1, 2, 3 };

for each (int element in numbers)
{
    Console::WriteLine(element);
}
```

numbers - это синтезированный ссылочный тип, расширающий встроенный тип System.Array. Он инициализируется тремя элементами, и конструкция for each запускает инструкции в своём блоке видимости для каждого из этих элементов. Достаточно логично. Я думаю, что вы можете представить эквивалентный код в любом выбранном вами языке. Достаточно предсказуемо, что это можно реализовать с помощью следующего набора инструкций.

```
    .locals init (int32[] numbers,
                  int32 index)

// Создаём массив

    ldc.i4.3
    newarr int32
    stloc numbers

// Заполняем массив

    ldloc numbers
    ldc.i4.0 // позиция
    ldc.i4.1 // значение
    stelem.i4

    ldloc numbers
    ldc.i4.1 // позиция
    ldc.i4.2 // значение
```

```
    stelem.i4

    ldloc numbers
    ldc.i4.2 // позиция
    ldc.i4.3 // значение
    stelem.i4

    br.s _CONDITION

_LOOP:

    ldc.i4.1
    ldloc index
    add
    stloc index

_CONDITION:

    ldloc numbers
    ldlen
    ldloc index
    beq _CONTINUE

// конструкция for each

    ldloc numbers
    ldloc index
    ldelem.i4
    call void [mscorlib]System.Console::WriteLine(int32)

    br.s _LOOP

_CONTINUE:
```

Единственные инструкции, ещё не затрагивавшиеся в этом цикле, это те, которые относятся к массивами. Инструкция newarr достаёт целочисленное значение (integer), задающее количество элементов в массиве, из стека. Затем она создаёт массив заданного типа и помещает ссылку на него в массив. Инструкция stelem заменяет элемент в массиве указанным значением. В стек перед этим должно быть помещена ссылка на массив, позиция элемента в массиве и новое значение элемента. Наконец, инструкция ldelem загружает элемент из массива и помещает его в стек, предварительно взяв из последнего ссылку на массив и позицию элемента.

Как вы можете видеть, этот код похож на тот, который мог бы быть сгенерирован для конструкции for, пробегающей через те же элементы (см. главу 6). Конечно, конструкция for each можно использовать не только с типами, происходящими от System.Array. На самом деле, она может перебирать элементы любого типа, которые реализуют интерфейс IEnumerable или даже типы, предоставляющие ту же функциональность, не не реализующие вышеуказанный интерфейс. Степерь поддержки естественным образом зависит от решений, реализованных в вашем конкретном языке. Некоторые языки могут даже предоставлять поддержку для использования совершенно других контейнеров в конструкции for each. Рассмотрим следующий пример, используя класс контейнера вектора из стандартной библиотеки C++ (STL).

```
std::vector<int> numbers;
numbers.reserve(3);
numbers.push_back(1);
numbers.push_back(2);
numbers.push_back(3);

for each (int element in numbers)
{
    std::cout << element << std::endl;
}
```

В данном примере компилятор вызывает метод класса begin, чтобы получить итератор, указывающий на первый элемент контейнера. Затем этот итератор используется для перебора

элементов. Значение итератора каждый раз увеличивается, пока не станет равно значению итератора, возвращённого методом end vector'a. При каждом проходе итератор разыменовывается, а получившееся значение помещается в соответствующий элемент. Элегантность заключается в том, что использование for each позволяет избежать многих распространённых ошибок, связанных с переполнением буфера и других, подобных этим.

Давайте вернёмся к управляемому коду. Если конструкция for each должна поддерживать коллекции, а не массивы, то, естественно, это ведёт к иной реализации, зависящей от использования конструкции. Рассмотрим следующий пример:

```
Collections::ArrayList numbers(3);
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

for each (int element in %numbers)
{
    Console::WriteLine(element);
}
```

Держите в уме, что большая часть контейнеров STL, включающих конструктор, принимающий одно целочисленное значение, обычно устанавливают начальный размер контейнера, в кто время как .NET-коллекция используют конструктор для установки начальной вместимости.

Так что же делает for each в данном случае? Ну, тип ArrayList реализует интерфейс IEnumerable с единственным методом GetEnumerator. Конструкция for each вызывает метод GetEnumerator, чтобы получить реализация интерфейса IEnumerator, который затем используется для перебора коллекции. Давайте проверим простую реализацию для данного кода:

```
    .locals init (class [mscorlib]System.Collections.ArrayList numbers,
                  class [mscorlib]System.Collections.IEnumerator enumerator)

// Создаём массив

    ldc.i4.3
    newobj instance void [mscorlib]System.Collections.ArrayList::.ctor(int32)
    stloc numbers

// Заполняем массив

    ldloc numbers
    ldc.i4.1
    box int32
    callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
    pop

    ldloc numbers
    ldc.i4.2
    box int32
    callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
    pop

    ldloc numbers
    ldc.i4.2
    box int32
    callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
    pop

// Получаем перечислитель

    ldloc numbers

    callvirt instance class [mscorlib]System.Collections.IEnumerator
        [mscorlib]System.Collections.IEnumerable::GetEnumerator()

    stloc enumerator
    br.s _CONDITION
```

```
_CONDITION:

    ldloc enumerator
    callvirt instance bool [mscorlib]System.Collections.IEnumerator::MoveNext()
    brfalse.s _CONTINUE

// конструкция for each

    ldloc enumerator
    callvirt instance object [mscorlib]System.Collections.IEnumerator::get_Current()
    call void [mscorlib]System.Console::WriteLine(object)

    br.s _CONDITION

_CONTINUE:
```

Ок, выглядит неплохо, но чего-то нехватает. Можете сказать, чего именно? Так как при использовании конструкции for each создаётся новый объект, когда вызывается метод GetEnumerator, то разумно предположить, что этот объекту требуется в определённый момент "подчистить", чтобы избежать утечек ресурсво. Перечислитель (enumerator) делает это, реализуя интерфейс IDisposable. К сожалению, комплиятор не всегда может распознать это во время компиляции, хотя если у него есть достаточно статической информации о типах, то есть определённый смысл в том, чтобы извлечь из неё пользу и оптимизировать сгенерированные инструкции. Получается, что перечислитель, предоставляемый типом ArrayList, не реализует интерфейс IDisposable. Конечно, это может измениться в будущем, так что "оптимизирование" вашего кода в подобных случаях - не слишком мудрый шаг. Компиляторы могут использовать инструкцию isinst, чтобы определить, реализует ли перечислитель интерфейс IDisposable. Возможно, что это упущение в дизайне интерфейсов IEnumerable и IEnumerator.

Чтобы решить эту проблему, оригинальный интерфейс IEnumerator, представленный в NET Framework 2.0, наследуется от IDisposable, поэтому реализациям необходимо предоставить метод Dispose, даже если он ничего не делает. Это очевидным образом упрощает дела для того, кто совершает вызов. Collections.Generic.IEnumerator<T> - это тип значения, возвращаемого методом GetEnumerator, который относится к интерфейсу Collections.Generic.IEnumerable<T>, реализуемого новой базовой коллекцией типов. Для совместимости с существующим кодом, реализации перечислителя так реализуют старый интерфейс IEnumerator. Рассмотрим следующий пример:

```
Collections::Generic::List<int> numbers(3);
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

for each (int element in %numbers)
{
    Console::WriteLine(element);
}
```

В этом случае не возникает вопроса, должен ли перечислитель реализовывать IDisposable. Единственная остающаяся проблема - это найти надёжный способ вызвать метод Dispose несмотря на исключения. Вот подходящее решение, использующее конструкции обработки исключений, обсуждавшиеся в 5-ой части.

```
    .locals init (class [mscorlib]System.Collections.Generic.'List`1'<int32> numbers,
                  class [mscorlib]System.Collections.Generic.'IEnumerator`1'<int32> enumerator)

// Создаём массив

    ldc.i4.3
    newobj instance void class [mscorlib]System.Collections.Generic.'List`1'<int32>::.ctor(int32)
    stloc numbers

// Заполняем массив

    ldloc numbers
```

```
    ldc.i4.1 // value
    callvirt instance void class [mscorlib]System.Collections.Generic.'List`1'<int32>::Add(!0)

    ldloc numbers
    ldc.i4.2 // value
    callvirt instance void class [mscorlib]System.Collections.Generic.'List`1'<int32>::Add(!0)

    ldloc numbers
    ldc.i4.3 // value
    callvirt instance void class [mscorlib]System.Collections.Generic.'List`1'<int32>::Add(!0)

// Получаем перечислитель

    ldloc numbers

    callvirt instance class [mscorlib]System.Collections.Generic.'IEnumerator`1'<!0>
        class [mscorlib]System.Collections.Generic.'IEnumerable`1'<int32>::GetEnumerator()

    stloc enumerator

    .try
    {

    _CONDITION:

        ldloc enumerator
        callvirt instance bool class [mscorlib]System.Collections.Generic.'IEnumerator`1'<int32>::
    MoveNext()
        brfalse.s _LEAVE

    _STATEMENTS:

        ldloc enumerator
        callvirt instance !0 class [mscorlib]System.Collections.Generic.'IEnumerator`1'<int32>::
    get_Current()
        call void [mscorlib]System.Console::WriteLine(int32)
        br.s _CONDITION

    _LEAVE:

        leave.s _CONTINUE
    }
    finally
    {
        ldloc enumerator
        callvirt instance void [mscorlib]System.IDisposable::Dispose()

        endfinally
    }

_CONTINUE:
```

Учитывая комментарии и метки, код должен достаточно понятным. Давайте быстро пробежимся по нему. Запрошены две локальные переменные, одна для списка, а другая для перечислителя. Имена типов выглядят странно, так как должны поддерживать MSIL'овские дженерики. 'List`1' показывает, что используется тип List с одним параметром типа. Это позволяет отличать дженерик-типы с одним и тем же именем, но разным количество параметров. Тип List между кавычками указывает на реальные типы, используемые для конкретизации времени выполнения.

Определив локальные переменные, следующий шаг - это создать список, используя инструкцию newobj и заполнить его с помощью принадлежащего ему метода Add. Дженерики предназначены, в основном, для создания типизированных коллекций. Хорошим примером этого является код для добавления чисел в список. Предыдущий пример использовал ArrayList, поэтому числа сначала должны были пройти через инструкцию box, прежде чем быть добавленными в коллекцию. В данном случае мы можем просто положить числа в стек и вызвать метод Add. Нам просто нужно задать требуемую конкретизацию. Единственный параметр метода Add - это !0, который указывает

на параметр типа, основанный на нуле. Теперь, когда коллекция готова, мы можем реализовать сами циклы. Чтобы начать перечисление, мы сначала получаем из коллекции реализацию IEnumerator<T> и сохраняем её во временной переменной. Далее вызывается метод перечислителя MoveNext, пока он не возвратит false. Обработчик исключения finally используется для вызова метода перечислителя Dispose.

# 23. Linux/Unix

# Ассемблер в Unix

**Автор:** marlyn

**Дата:** 2003-09-01

## Введение

Так исторически сложилось, что программирование на ассемблере под unix почти не востребовано, и занимаются им только кодеры-маньяки, дзен-буддисты и прочие настоящие ассемблерщики.

Настоящий ассемблерщик - зверь крайне редкий, практически нигде и не встретишь его, разве что в заповеднике - wasm.ru. Unix-ассемблерщик еще более редкий подвид, практически вымерший, если не считать, западный, linuxassembly.org.

Для исправления такой плачевной ситуации, и была написана эта статья, а точнее цикл статей, которые по задумке автора, должны привлечь в ряды адептов-юникс-дзена множество новых членов.

В первой части (которую вы сейчас читаете) я имею честь познакомить вас с прекрасным миром unix-программирования, что выльется в написание простейшего helloworld. В следующей части - мы разберем несколько, более сложных примеров. И под конец, наверное, будет программирование под **x-windows**.

## Инструменты

Для нормально функционирования нам понадобятся следующие вещи:

1. Собственно какая-либо **unix**-совместимая ось. (например **linux**, или лучше **FreeBSD** ),
2. Компилятор fasm. ( www.flatassembler.net )
3. Линкер **ld**( есть почти в любом дистрибутиве **unix** ),
4. Особый склад ума, причем последнее - самое главное. Если у вас этого нет, то ни один, даже самый последний RedHat на пару со свежим fasm'ом вам не поможет.

И еще, о компиляторах - в **unix** обычно используются **AS** с **AT&T** синтаксисом, который для многих людей, выросших на **tasm'e** и **masm'e**, кажется полной абракадаброй. Поэтому, для начала, мы будем использовать привычные компиляторы с *Intel'овским* синтаксисом (**fasm или nasm**). Хотя позже, если найдутся желающие, можно будет рассмотреть и **AT&T** asm.

## Общие сведения

Unix, который мы будем использовать - 32 битная система, работающая в защищенном режиме, и использующая плоскую модель памяти.

Как и большинство операционных систем, Unix предоставляет программе набор различных функций (по другому - Api). Но, в отличие от, например, WinAPI, где вызовы производятся с помощью call'ов, в unix - больше свободы: можно вызывать функция ядра напрямую, а можно использовать многочисленные библиотеки. Рассмотрим для начала первый способ.

Системный вызов производится с помощью прерывания **0x80**(чаще всего). К сожалению, (а может и к счастью) существует несколько конвенций вызова, что приводит к несовместимости кода между многими **unix-like** осями. Я рассмотрю только две, самые популярные платформы: **Linux** и***BSD**. **FreeBSD (а также OpenBSD и NetBSD)** Эта система использует традиционную **unix** конвенцию вызова: номер функции помещается в **eax**, параметры в стек, вызов производится с помощью функции содержащей int **0x80**, а результат возвращается в **eax**.

Наверное, понятнее будет, если рассмотреть это на примере:

```
sys_call:
             int  0x80
             ret
       start:
         push msg_len   ; размер строки
         push msg       ; адрес строки
         push 1         ; stdout
         mov  eax,4     ; номер системной функции - sys_write
         call sys_call
         add  esp,4*3   ; очищаем за собой стек
```

Впрочем, от функции **sys_call** можно отказаться, достаточно просто помещать в стек лишний **dword**:

```
start:
              push msg_len   ; размер строки
              push msg       ; адрес строки
              push 1         ; stdout
              mov  eax,4     ; номер системной функции - sys_write
              push eax       ; все что угодно
              int  0x80
              add  esp,4*3   ; очищаем за собой стек
```

Также **FreeBSD** поддерживает конвенцию вызова, применяемую в **linux**. Для это необходимо включить **linux emulation**. Еще эта эмуляция потребуется для запуска **fasm**. А еще нужна утилита **brandelf**(наверняка она у вас есть). Дело в том, что пока не существует версии **fasm'a** конкретно для **BSD** систем. Но это легко исправить, вот так:

```
Brandelf –t Linux fasm
```

Если это не сработает (а такое возможно из-за не совместимости форматов), придется перекомпилировать fasm, заменив формат файла "*format PE executable*" на простой "*format ELF*", а потом слинковать **ld**. **Linux** В линуксе используется **fastcall** конвенция. Номер функции, все так же, помещается в **eax**, а вот параметры, вместо стека, помещаются в регистры.

Пример:

```
              mov  edx,msg_len
              mov  ecx,msg
              mov  ebx,1
              mov  eax,4
              int  0x80
```

Порядок размещения параметров такой:

| № параметра | Регистры |
|---|---|
| 1 | ebx |
| 2 | ecx |
| 3 | edx |

| № параметра | Регистры |
|---|---|
| 4 | esi |
| 5 | edi |
| 6 | ebp |

Как видите максимальное количество параметров - 6. Если их больше, приходиться помещать все параметры в структуру и передавать ее адрес в **ebx**.

## Описание системных функций

После того как вы разобрались с вызовом функций, будет логичным вопрос: "А где взять описание этих самых функций?".

Ничего похожего на **msdn**, в **unix** среде к сожалению не существует, но не нужно забывать: **unix** - система с открытым исходном кодом и все нужное, можно найти там.

```
Для linux:
        arch/i386/kernel/entry.S
        include/asm-i386/unistd.h
        include/linux/sys.h
Для FreeBSD:
        i386/i386/exception.s
        i386/i386/trap.c
        sys/syscall.h
```

Для каждой функции можно посмотреть описание, используя man(2).

## Пример программы. Hello world

Пришло время написать, тот самый, жутко всем надоевший - **HelloWorld**.

Я приведу пример только **FreeBSD** версии, переписать это под linux - будет вашим домашним заданием. (для самых ленивых - см. примеры к статье)

```
------------------[cut]-----------------------------------

format ELF
section '.text' executable
public _start
_start:
              push msg_len   ; size of message
              push msg       ; offset of message
              push 1         ; stdout
              mov  eax,4     ; 4 =  sys_write
              push eax
              int  0x80
              add  esp,4*3   ; очищаем за собой стэк

              xor  eax,eax
              push eax       ; код выхода
              inc  eax       ; 1 = sys_exit
              int  0x80

section '.data' writeable

              msg db "Hello world",0
              msg_len = $-msg

------------------[end cut]-------------------------------
```

**Сборка.**

Сначала скомпилируем файл, вот так:

```
fasm hello.asm hello.o
```

А потом слинкуем:

```
ld -o hello hello.o
```

А теперь посмотрите на размер. 600 байт, впечатляет?! ( размер можно еще очень сильно уменьшить, но об этом, как-нибудь в другой раз)

## Использование библиотеки libc

Некрасивый и совсем не *дзенский* способ, но все же мы его рассмотрим - для полноты картины.

Итак, **libc** (*c library*) - это стандартная библиотека с для **UNIX**. Она содержит в себе кучу полезных функций, типа **printf**, и используется почти во всех обычных программах (кстати сказать, многие функции этой библиотеки - простые обертки над вызовами ядра).

В **FASM** е существуют удобные макросы, для вызова си функций..., но я не буду их использовать, отдав предпочтение чистому ассемблеру.

Пример:

```
------------------[cut]-----------------------------------
format ELF
section '.text' executable

extrn printf

public main
main:
        push msg
        call printf
        add  esp,4
        ret

section '.data' writeable

        msg db "Hello world!\n",0
------------------[end cut]-------------------------------
```

Компилируется это дело так:

```
fasm hellolib.asm hellolib.o
gcc -o hellolib hellolib.o
```

Заключение.

Ну вот вы и написали свою первую программу на ассемблере под **UNIX**.

Все на много проще чем кажется, неправда ли?

Если у вас возникнут какие-либо вопросы, пишите мне на adain@mail.ru, или на форум WASM.RU.

До встречи.

Примеры к статье (находится в архиве `wasm_files.zip`: `files/0273/asminunix.zip`).

# Ассемблер в *nix – удел извращенца..?

**Автор:** Broken Sword

**Дата:** 2003-09-01

> *A lot of people ask me..*
> *stupid fuckin questions.*
> *A lot of people think that..*
> *ассемблера в *nix не существует.*
> *(C) типа Eminem*

Да, асм в ***nix таки существует. Просто многие в это почему-то отказываются верить. Данная статья скромно претендует развеять все мифы и загадки вокруг этого загадочного явления. Не ищите здесь подробного описания AT&T синтаксиса и системных вызовов – я просто попробую описать те трудности и невзгоды, которые обязательно придется преодолеть смельчаку, желающему полностью овладеть*nix -ом через асм (в смысле - научиться программировать на асме под*nix**, прим. для CyberManiac-а :) ).

## Миф I:

### “Синтаксис асма под *nix в корне отличается от оного под DOS/WIN, он кривой и к нему невозможно привыкнуть”.

Действительно, синтаксис, предложенный компанией **AT&T**, немного (безусловно, в лучшую сторону) отличается от Intel’овского. Но это ничего не значит. На самом деле, если бы все еще с детства начали кодировать, используя **AT&T** синтаксис, то интеловский очень скоро загнулся бы и не дожил до наших дней. Просто **AT&T** в свое время не стремилась делать свои поделки достоянием народных масс, это прерогатива MS и Intel (а вообще у **AT&T** уже существовала своя развитая культура и традиции, когда **MS** и **Intel** только спускались с деревьев :) ).

А уж если кто-то переборол свои страхи и все же перешел к AT&T с Intel, того точно уже за уши не оттянешь назад. По этому поводу когда-то даже родился проект DJGPP (GNU Binutils) - асм с AT&T синтаксисом под DOS (http://www.delorie.com/djgpp/). Однако в силу непонятных причин проект не прижился.

Если все эти доводы кому-то показались слишком хлипкими и невразумительными, то нет ничего проще - http://sf.net/projects/nasm/. Качайте себе NASM под *nix и будьте с Intel «вместе навсегда».

## Миф II:

### “Асм под *nix никому не нужен и вообще все это извращ, С – вот единственная дорога в светлое будущее”.

Существует известная в широких кругах поговорка: «*Линукс писан программистами для программистов*». На самом деле полностью она звучит так: «*Линукс писан сишными программистами для сишных программистов*». И с этим никто не спорит. Дело в том, что любому, кто попытается сунуться в такой монастырь как ***nix со своим уставом (асм), тому суждено упереться в огненные стены и рвы с крокодилами (об этом ниже). Однако не все так трагично – уменьшение размера кода в сто и более раз, увеличение производительности в десятки раз, да и чего греха таить, – само удовольствие, которое может доставить только кодинг на живом языке - все это стоит того чтобы научиться асму под*nix**.

Теперь, возможно это кого-то сильно удивит, но программирование на асме под ***nix по своему стилю очень напоминает... программирование на оном под DOS (да! Именно под DOS). Многие скажут:*nix – полностью 32-х разрядная ОС, и работает в защищенном режиме, используя flat -модель памяти. При чем здесь 16-битная ОС реального режима DOS ? Про такого с уверенностью можно сказать: он просто никогда не писал на асме под*nix**. На самом деле можно сделать еще более шокирующее заявление: писать программы на асме в *nix НАМНОГО проще чем под DOS... Самое важное – не сойти “с пути истинного” в самом начале его...

**#1**: когда только начинаешь писать на асме под *nix то возникает интересное ощущение: вроде бы ты попал в грязный пятибаксовый мотель (из тех, возле которых обязательно проходит метро и когда едет поезд на потолке дрожит дешевая люстра и мигает свет); здесь давно нет горячей воды, обои уродливыми клочьями свисают со стен, с потолка капает какая то мерзкая гадость и пахнет плесенью, все удобства – во дворе... На мотеле (подпертые кем-то неизвестным) стоят уже давно покосившиеся со временем неоновые буквы «*NIX для ассемблерщиков» (половина букв давно не горит, а половина с треском догорает). У мотеля нет своих постояльцев. Сюда заезжают лишь переночевать, чтобы на следующее утро убраться подальше...

Самое мерзкое во всем этом то, что через единственное окно в этой конуре, через дорогу, как будто специально, вырос семизвездочный отель, весь в рекламе, бассейнах и пальмах... Прямо над входом (к которому то и дело поминутно подъезжают все более и более крутые тачки) сверкает золотом надпись: “*NIX для сишников”. Вон видно как по террасам ходят пузатые мужики в обнимку с дорогими бабами, потягивая коктейли и куря сигары, им прислуживает армия официантов и слуг; все они смеются и живут.

Всем им наплевать на мотель напротив...

Но в миру ходят легенды, что в том самом мотельчике существует некая потайная дверь, которая открывает путь в Вечное... Ради этой двери мотель и стоит. По крайней мере, ручеек из желающих приобщиться к Вечности никогда не пересыхает.

Я говорю к тому, что ассеблерщик, сунувшийся в *nix не найдет практически никакой документации, описывающей системные вызовы на низком уровне. Здесь (http://www.lxhp.in-berlin.de/lhpsyscal.html) об этом можно получить кое-какую захудалую информацию, но многие функции описаны неправильно, либо вообще не описаны. Иногда порядок расположения параметров в регистрах при передаче в ту или иную функцию приходиться подбирать буквально вручную, методом научного тыка. Но на самом деле настоящего асм-кодера все это может только раззадорить..

Для вызова любой системной функции используется команда **INT 80h**(вспомните DOS – там для этой цели использовался **INT 21h**). Параметры передаются через регистры. Номер функции – в **AX**. Вся проблема в том, что найти полное описание того, в каком регистре какой параметр и для чего передается крайне проблематично, ресурс, указанный выше частично решает эту проблему.

Когда я говорил про все неземные блага, которые предоставляются для сишников, я имел ввиду полную документированность любой запятой, с которой только можно встретиться в увлекательном процессе программирования на C под *nix.

**#2**: вот это действительно самый большой фак: за всю историю существования *NIX никто не написал ни одного стоящего отладчика асм-кода (а может и написал, но не захотел поделиться с общественностью). Все что удалось найти (был перерыт буквально весь Интернет и опрошены десятки знающих людей) – ALD (*Assembly Linux Debugger*). Все что про него можно сказать – да, он действительно чем-то круче **MS debug-a**. Вот только чем именно - сказать довольно сложно. Все остальные отладчики дальше C-шного кода ничерта не видят. Писать программы без отладчика (а тем более на асме) – это верх извращенческого гения.

Конечно, существует еще и скромный аналог сайса под Линукс – **PrivateICE** (http://sourceforge.net/projects/pice). Единственная проблемка – на последних версиях ядер Линукса он не компилируется (вот вам и переносимость C).

Прим.: 9 июня 2003 года на sourceforge появился новый билд pice-a. Однако вот что пишет сам автор:

> Since this project was abandoned a few years ago there is no active maintainence. [skipped]. The files provided here are as it is and there is no garuantee that they will compile. [skipped]. Of course you are welcome to send in bug reports or comments on this code, just don't expect that they can be compiled out of the box. :) These sources will compile on 2.4.18. They might give problems with non-SMP enabled kernels.

Скомпилировать так ничего и не удалось :). Если кто-то вдруг найдет заклинание, по которому **pice** можно скомпилировать под последние **ASP**, **Mandrake** или **RedHat**– сообщите плз на ящик внизу. **#3**: проблема заголовочных файлов. Чуть ли не большую часть времени желающему покодить на асме в***nix придется провести за увлекательным поиском необходимой информации по заголовочным файлам. Искать придется практически все буквенные названия, которые могут встретиться в процессе (это и названия самих системных вызовов, и параметров, передаваемых в них, и вообще любых переменных). Все они как будто специально порастасканы по тысячам*.h** – файлов, которые находятся в самых неожиданных местах. Сишнику в семизвездочном отеле достаточно всего лишь щелкнуть пальцем и сделать *include ….h* – все работает. Ассемблерщику в мотелишке напротив – сначала понять к чему тот или иной параметр, затем устроить поиск по всем системным директориям, найти заголовочный файл, содержащий этот параметр, затем скопировать его в «свой», который будет понимать **GAS** или **NASM**(или, если ассемблерщик попался реальный, то запомнить его, и везде использовать численные значения параметров ?), а в довершение еще и усадить в правильный регистр перед отправкой в недра функции. **#4**: прога, написанная для***nix на асме теряет переносимость на другие*nix -платформы. Для перекомпиляции под другую*nix платформу придется изрядно повозиться, в некоторых случаях проще переписать заново весь код, чем переделывать старый с Linux под BSD . Но BSD пока не так распространен, а самые попсовые версии линуксов ( Red Hat , Mandrake , ASP**) используют одно и то же ядро, и данный фак вообщем-то не так уж страшен как его малюют.

Ну вот, а теперь вы сами убедитесь, что писать программы на асме под Линукс не сложнее чем под DOS, а может даже и проще.

Рассмотрим пример написания простейшего клиент-серверного приложения, использующего в качестве взаимодействия стек протоколов TCP/IP (подразумевается, что вы более-менее знакомы с сетевым программированием, и знаете хотя бы, что такое сокет).

Клиентское приложение посылает серверу символьную строку; Сервер шифрует символьную строку по следующему алгоритму шифрования: выполняет замену одного символа – буквы, на символ располагающийся на две позиции правее в алфавите, для последнего и предпоследнего символов в алфавите выполняется кольцевой сдвиг, например, для «Y» это будет буква «A» для «Z» - «B» (это шифр Цезаря). Сервер отправляет зашифрованное сообщение обратно; Клиент выбирает шифрограмму и выводит на экран.

Сообщения, подлежащие шифрованию, вводятся с клавиатуры. Программа сервера работает в бесконечном цикле.

Ну что ж, начнем с сервера. Я по ходу пьесы попытаюсь максимально комментировать происходящее. Начну с того, что наш сервер будет демоном (для пущего понту). Что такое демон? Демон на языке DOS-ассемблерщиков – резидент. Все. Так что ничего страшного.

```
# ****************************************************
# Server (daemon)
# by Broken Sword [HI-TECH]
# (for Linux based on Intel x86 only)

# brokensword@mail.ru
# www.wasm.ru

# Compile Instructions:
# -----------------------------------------------------
# as server.s
# ld --strip-all -o server a.out
#
    ************************************************************#*************************************
    ***************

# этот файлик содержит выдранные из какого-то файла
# определения системных вызовов (см. FUCK #3)
.include  "syscalls.inc"
# а этот – все остальные, которые только могут
# встретиться

.include  "def.inc"
  .text         # начало сегмента кода
# метка, с которой все начинается (нужно чтоб она была
# глобальной)
  .globl  _start

_start:
# итак, начинаем лепить нашего демона
# процесс создания демона в *.nix и создание резидента в DOS в корне различаются
# начинается любой демон с того, что нужно создать дочерний процесс.
# Создать дочерний процесс в линуксе проще
# пареной репы – достаточно поместить номер сис.
# вызова в EAX и сделать «а-ля int 21h», т.е. int 80h
  movl  $SYS_fork,%EAX
  int  $0x80
# все.
# Теперь у нас параллельно сосуществуют ДВА процесса:
# родительский (в котором исполнялись все предыдущие
# команды) и дочерний. Что же содержит дочерний код?
# А все то же самое, что и родительский.
# Т.е. важно понять, что # весь нижеследующий (и выше тоже)
# код находиться в памяти в ДВУХ разных местах.
# Как процессор переключается между
# ними (и всеми остальными живыми процессами)
# – читайте «Переключение задач» в интеловском мануале.
test  %EAX,%EAX
# вот эту команду необходимо осознать.
```

```
# Прежде всего, важно понять, что данная команда
# существует и в родительском
# и в дочернем процессах (об этом выше).
# Следовательно выполниться она и там и там.
# Все дело в том, что после
# int 80h родительскому процессу вернется PID сына
# (в EAX ессесно, вообще все возвращается в EAX, как и в винде)
# а что же вернется сыне? Правильно, нолик.
# Именно поэтому следующий jmp будет выполнен
# в дочернем процессе и
# не будет выполнен в родительском.

# ребенок улетает на метку _cont1
  jz  _cont1
# ...а в это время, в родительском процессе:
  xorl  %EBX,%EBX  # EBX=status code
  xorl  %EAX,%EAX  #
  incl  %EAX   # SYS_exit
# завершаем родительский процесс.
  int  $0x80

# Теперь все дети
# управляются процессом INIT


_cont1:
  movl  $SYS_setsid,%EAX
# сделаем нашего ребенка главным в группе
  int  $0x80

  movl  $1,%EBX   # SIGHUP
  movl  $1,%ECX   # SIG_IGN
  movl  $SYS_signal,%EAX
# далее сигнал SIGHUP будет игнорироваться
  int  $0x80

  movl  $SYS_fork,%EAX
# наш ребенок уже подрос и теперь сам может родить сына
  int  $0x80
# (по сути – это уже внук нашему изначальному
# родительскому процессу)

# EAX=0 в дочернем и EAX=PIDдочернего в родительском

  test  %EAX,%EAX

jz  _cont2

# внук нашего родительского (которого уже давно нет в
# живых) улетает на метку _cont2, однако отец все еще
# жив!!! (все как в мексиканском сериале)

  xorl  %EBX,%EBX  # EBX=status code
  xorl  %EAX,%EAX  #
  incl  %EAX   # SYS_exit
  int         $0x80

# вот уже и отец отправлен к деду на небеса (да,
# злостная программка, недаром демоном зовется)

# ..а в это время внучок получает все наследство и

_cont2:

# продолжает жить
# далее, после того,
# как все кровавые разборки и отцеубийства благополучно завершены,
# внучок,продавший душу демону,
# преспокойно создает сокет.
# Дело в том, что в линуксе есть такие системные вызовы,
# для вызова которых их номер не
```

```
# помещается в EAX.
# Вместо этого в EAX помещается номер функции-мультиплексора,
# реализующий вызов конкретной
# функции номер которой помещается в EBX.
# Так, например, происходит при вызове IPC и SOCKET-функций.
# Кроме того,
# при вызове SOCKET-функций параметры располагаются не в регистрах,
# а в стеке. Смотри как все просто:

  pushl  $0   # протокол
  pushl  $SOCK_STREAM  # тип

  pushl  $AF_INET   # домен
# ECX должен указывать на кадр в стеке, содержащий
  movl  %ESP,%ECX
# параметры, такая уж у него судьба...
  movl  $SYS_SOCKET,%EBX
# а вот это уже номер той самой конкретной функции
# SOCKET – создать сокет

  movl  $SYS_socketcall,%EAX

# в EAX - номер функции мультиплексора (по сути он
# просто перенаправит вызов в функцию, указанную в EBX

       int         $0x80

# сокет создан! Ура товарищи. В EAX возвратиться его дескриптор.

# «очистим» стек (по сути это выражение придумано специально
# для HL-программистов, на самом деле ничего не
# очищается, данную операцию необходимо производить только
# для того чтобы в дальнейшем не произошло переполнение
# стека, но в таких маленьких программках это делать вовсе
# не обязательно):

  addl  $0xC,%ESP

  movl  %EAX,(sockfd)

# сохраним дескриптор созданного сокета в переменной
# sockfd

# далее необходимо осуществить операцию BIND,
# которая называется «привязка имени сокету»,
# хотя суть этого названия
# слабо отражает смысл происходящего на самом деле.
# На самом деле BIND просто назначает конкретному сокету IP-
# адрес и порт, через который с ним можно взаимодействовать:

# размер передаваемой структуры (вообще подобран
  pushl  $0x10
# методом тыка, потому что логически непонятно почему
# именно 16)

# указатель на структуру sockaddr_in

       pushl  $sockaddr_in
# дескриптор нашего сокета
  pushl  %EAX
# ECX указывает на параметры в стеке
  movl  %ESP,%ECX
# номер функции BIND – в EBX
  movl  $SYS_BIND,%EBX
# функция-мультиплексор
  movl  $SYS_socketcall,%EAX
  int  $0x80
# теперь сокет «привязан» к конкретному IP-шнику и порту
# поднимем ESP на место
  addl  $0xC,%ESP
# далее что-либо подробно описывать я не вижу смысла,
```

```
# любой желающий сам без труда разберется, опираясь на
# полученные выше знания.
  pushl  $0   # backlog
  movl  (sockfd),%EAX
  pushl  %EAX
  movl  %ESP,%ECX
  movl  $SYS_LISTEN,%EBX
  movl  $SYS_socketcall,%EAX
  int         $0x80
  addl  $0x8,%ESP

_wait_next_client:
  pushl  $0   # addrlen
  pushl  $0   # cliaddr
  movl  (sockfd),%EAX
  pushl  %EAX   # sockfd
  movl  %ESP,%ECX
  movl  $SYS_ACCEPT,%EBX
  movl  $SYS_socketcall,%EAX
  int  $0x80
  addl  $0xC,%ESP

  movl  %EAX,(connfd)

  movl  $SYS_fork,%EAX
  int  $0x80   # create child process
  test  %EAX,%EAX
  jnz  _wait_next_client

_next_plain_text:
  movl  (connfd),%EBX
  movl  $buf,%ECX  # ECX->buf
  movl  $1024,%EDX  # 1024 bytes
  movl  $SYS_read,%EAX
  int  $0x80   # wait plain_text

  movl  $buf,%ESI
  movl  %ESI,%EDI
  movl  %EAX,%ECX
  movl  %EAX,%EDX
_encrypt:
  lodsb
  cmp  $0x41,%AL  # A
  jb  _next
  cmp  $0x5A,%AL  # Z
  ja  _maybe_small
  incb  %AL
  incb  %AL   # encryption ;)
  cmp  $0x5A,%AL
  jle  _next
  sub  $26,%AL
_maybe_small:
  cmp  $0x61,%AL  # a
  jb  _next
  cmp  $0x7A,%AL  # z
  ja  _next
  incb  %AL
  incb  %AL   # encryption ;)
  cmp  $0x7A,%AL
  jle  _next
  sub  $26,%AL
_next:
  stosb
  loop  _encrypt

  movl  (connfd),%EBX
  movl  $buf,%ECX  # ECX->chiper_text
  movl  $SYS_write,%EAX
  int  $0x80   # send plain_text

  jmp  _next_plain_text
```

```
# ***************************************************
  .data
sockfd:  .long  0
connfd:  .long  0
sockaddr_in:
sin_family: .word  AF_INET
sin_port: .word  0x3930   # port:12345
sin_addr: .long  0   # INADDR_ANY
buf:
# ***************************************************



#Клиент пишется по аналогии с сервером,
# думаю сами без труда разберетесь:

# *********************************************************
# Client
# by Broken Sword [HI-TECH]
# (for Linux based on Intel x86 only)

# brokensword@mail.ru
# www.wasm.ru

# Compile Instructions:
# -------------------------------------------------------
# as client.s
# ld --strip-all -o client a.out
#
    ***********************************************************#************************************
    ******************
       .include  "syscalls.inc"
  .include  "def.inc"
  .text
  .globl  _start
_start:
  pushl  $0   # protocol
  pushl  $SOCK_STREAM  # type
  pushl  $AF_INET   # domain
  movl  %ESP,%ECX
  movl  $SYS_SOCKET,%EBX
  movl  $SYS_socketcall,%EAX
  int  $0x80
  addl  $0xC,%ESP

  movl  %EAX,(sockfd)

  pushl  $0x10   # addrlen
  pushl  $sockaddr_in
  pushl  %EAX   # sockfd
  movl  %ESP,%ECX
  movl  $SYS_CONNECT,%EBX
  movl  $SYS_socketcall,%EAX
  int  $0x80
  addl  $0xC,%ESP

_next_plain_text:
  xorl  %EBX,%EBX  # stdin
  movl  $buf,%ECX  # ECX->buf
  movl  $1024,%EDX  # 1024 bytes
  movl  $SYS_read,%EAX
  int  $0x80   # read from stdin

  movl  (sockfd),%EBX
  movl  $buf,%ECX  # ECX->plain_text
  movl  %EAX,%EDX  # bytes read
  movl  $SYS_write,%EAX
  int  $0x80   # send plain_text

  movl  $SYS_read,%EAX
  int  $0x80   # wait chiper_text
```

```
  xorl  %EBX,%EBX
  incl  %EBX   # EBX=1 (stdout)
  movl  $SYS_write,%EAX
  int  $0x80   # disp chiper_text

  jmp  _next_plain_text
# *****************************************************
  .data
sockfd:  .long  0
sockaddr_in:
sin_family: .word  AF_INET
sin_port: .word  0x3930   # port:12345
sin_addr: .long  0x0100007F  # 127.0.0.1
buf:
# *****************************************************
```

Вот так вот все просто (вся сложность на самом деле заключена в понимании стека **TCP/IP**, а не в том, как закодировать все эти действия на асме).

Все - запускайте бесов **server**, а за ним **client**.

p.s. использовать данное приложение для шифрования важных данных на диске не рекомендуется ).

Благодарности (в алфавитном порядке :)):

Aquila [HI-TECH]
CyberManiac [HI-TECH]
Edmond [HI-TECH]
Vladimir [HI-TECH]

...и всей группе HI-TECH! Держитесь, ребята!

Примеры к статье (находится в архиве `wasm_files.zip`: `files/0274/asmunixlot.zip`).

# Ассемблер в Linux для программистов C

**Автор:** Dmitri Gribenko

**Дата:** 2008-06-06

# Введение

Эта книга ориентирована на программистов, которые уже знают C на достаточном уровне. Почему так? Вряд ли зная только пару несколько языков вроде Perl или Python кто-то захочет изучать ассемблер. Используя ассемблер и C вместе, применяя каждый язык для определённых целей, можно добиться очень хороших результатов. К тому же, программисты C уже имеют некоторые знания об архитектуре, способе организации памяти и других вещах, которые поначалу сложно понять. Поэтому изучать ассемблер после C несомненно легче, чем после других языков высокого уровня. В C есть понятие "указатель", программист должен сам управлять выделением памяти в куче и так далее — все эти знания пригодятся при изучении ассемблера, они помогут получить более целостную картину об архитектуре, а также иметь более представление о том, как выполняются их программы на C. Но эти знания требуют углубления и структурирования.

Хочу подчеркнуть, что для чтения этой книги никаких знаний о Linux не требуется (кроме, разумеется, "как создать текстовый файл" и "как запустить программу в консоли"). Да и вообще, единственное, в чем выражается ориентированность на Linux — это используемый синтаксис ассемблера. Программисты на асемблере в DOS и Windows используют синтаксис Intel, но в системах *nix принято использовать синтаксис AT&T. Именно синтаксисом AT&T написаны ассемблерные части ядра Linux, в синтаксисе AT&T компилятор GCC выводит ассемблерные листинги и т. д.

Большую часть информации из этой книги можно использовать для программирования не только в *nix, но и в Windows — нужно только уточнить некоторые системно-зависимые особенности.

# Глава 1. Архитектура

x86 или IA-32?
Регистры
Стек
Память
Порядок байтов. Little-endian и big-endian

## x86 или IA-32?

Вы вероятно уже слышали такое понятие, как "архитектура x86". Вообще оно довольно размыто, и вот почему. Само название x86 или 80x86 происходит от принципа, по которому Intel давала названия своим процессорам:

• Intel 8086 — 16 бит
• Intel 80186 — 16 бит
• Intel 80286 — 16 бит
• Intel 80386 — 32 бита
• Intel 80486 — 32 бита

Этот список можно продолжить. Принцип наименования, где каждому поколению процессоров давалось имя, заканчивающееся на 86, создал термин "x86". Но если посмотреть внимательнее, то мы увидим, что "процессором x86" можно назвать и древний 16-битный 8086, и новый Pentium 4. Поэтому 32-битные расширения были названы архитектурой IA-32 (сокращение от Intel Architecture, 32-bit). Конечно же, возможность запуска 16-битных программ осталась, и она успешно (и не очень)

используется в 32-битных версиях Windows. Так как Linux является полностью 32-битной операционной системой, то мы будем рассматривать только 32-битный режим.

Теперь, определившись наконец с предметом изучения, рассмотрим, что входит в понятие "архитектура системы". Это, в первую очередь, те понятия, с которыми придется работать программисту. А нас больше всего интересует набор команд и способы хранения и обращения к данным, над которыми будут оперировать команды. Также в архитектуру входят некоторые более абстрактные понятия (например, принцип сегментации памяти, реальный и защищённый режим) и технологии вообще (например, конвейер).

## Регистры

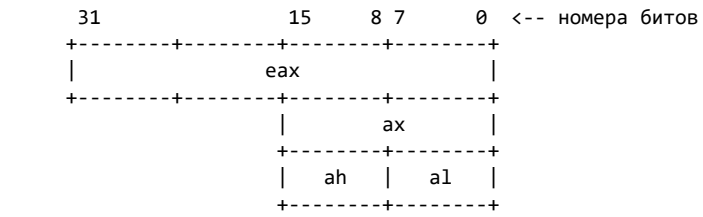
Регистр — это небольшой объем очень быстрой памяти, размещенной на процессоре. Она предназначена для хранения результатов промежуточных вычислений, а также некоторой информации для управления работой процессора. Так как регистры размещены непосредственно на процессоре, то доступ к данным в регистрах обычно в несколько раз быстрее доступа к данным в оперативной памяти.

Программисту доступны 16 пользовательских и 16 системных регистров. Они называются пользовательскими регистрами:

• Регистры общего назначения (англ. General Purpose Registers, сокращённо GPR). Размер — 32 бита.

• `%eax`: Accumulator register — аккумулятор, применяется для хранения результатов промежуточных вычислений
• `%ebx`: Base register — базовый регистр, применяется для хранения адреса (указателя на) некоторый объект в памяти
• `%ecx`: Counter register — счетчик, его неявно используют некоторые команды для организации циклов (см. loop)
• `%edx`: Data register — регистр данных, используется для хранения результатов промежуточных вычислений и ввода-вывода
• `%esp`: Stack pointer register — указатель стека. Содержит адрес вершины стека
• `%ebp`: Base pointer register — указатель базы кадра стека (англ. stack frame)
• `%esi`: Source index register — индекс источника, в цепочечных операциях содержит указатель на текущий элемент-источник
• `%edi`: Destination index register — индекс приёмника, в цепочечных операциях содержит указатель на текущий элемент-приёмник

Эти регистры можно использовать "по частям". Например, к младшим 16 битам регистра `%eax` можно обратиться как `%ax`. `%ax` в свою очередь содержит два байта: старший `%ah` и младший `%al`. Аналогично, можно обращаться к `%ebx`/`%bx`/`%bh`/`%bl`, `%ecx`/`%cx`/`%ch`/`%cl`, `%edx`/`%dx`/`%dh`/`%dl`, `%esi`/`%si`, `%edi`/`%di`.




• Сегментные регистры cs, ds, ss, es, fs, gs. В реальном режиме (16 бит) программы могли управлять этими регистрами. В защищённом режиме их полностью контролирует операционная

система, поэтому я здесь их подробно не рассматриваю, а лишь упоминаю. Также, благодаря "плоской" модели памяти (flat memory model) для программиста исчезают все проблемы, связанные с сегментацией памяти.
• Регистр флагов eflags и его младшие 16 бит, регистр flags. Он содержит информацию о состоянии выполнения программы, о самом микропроцессоре, а также информацию управляющую работой некоторых команд. Регистр флагов нужно рассматривать как массив битов, за каждым из которых закреплено определенное значение. Регистр флагов напрямую не доступен пользовательским программам, а также изменение некоторых битов eflags требует привилегий. Ниже перечислены наиболее важные флаги.

• cf: carry flag, флаг переноса

• 1 — во время арифметической операции был произведён перенос из старшего бита результата
• 0 — переноса не было

• zf: zero flag, флаг нуля

• 1 — результат последней операции нулевой
• 0 — результат последней операции ненулевой

• of: zero flag, флаг переполнения

• 1 — во время арифметической операции произошёл перенос в/из старшего (знакового) бита результата
• 0 — переноса не было

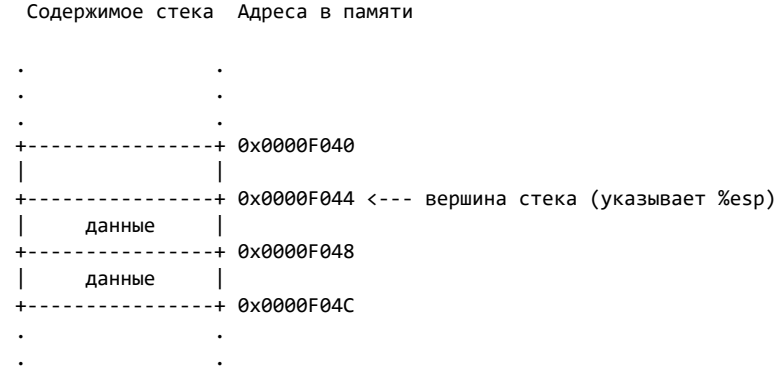
• df: direction flag, флаг направления. Указывает направление просмотра в строковых операциях

• 1 — направление "назад", от старших адресов к младшим
• 0 — направление "вперёд", от младших адресов к старшим

• Указатель команды eip (instruction pointer). Размер — 32 бита. Содержит указатель на следующую команду. Регистр напрямую недоступен, изменяется неявно командами условных и безусловных переходов, вызова и возврата из подпрограмм.

## Стек

Имея опыт программирования на C, я думаю, что вы слышали (а может и использовали) структуры данных типа стек. В микропроцессоре стек работает похожим образом: это область памяти, у которой определена вершина (на неё указывает `%esp`). Поместить новый элемент можно только на вершину стека, при этом новый элемент становится вершиной. Достать из стека можно только верхний элемент, при этом вершиной становится следующий элемент. У вас наверняка была в детстве игрушка-пирамидка, где нужно было разноцветные кольца одевать на общий стержень. Так вот эта пирамидка — отличный пример стека.



```
.                .
+----------------+ 0x0000FFF8
|     данные     |
+----------------+ 0x0000FFFC
|     данные     |
+----------------+ 0x00010000 <--- дно стека
```

Стек растёт в сторону младших адресов. Это значит, что последний записанный в стек элемент будет расположен по адресу младше остальных элементов стека.

Что же происходит при помещении нового элемента в стек? (Принцип работы команды push)

• значение `%esp` уменьшается на размер элемента в байтах (4 или 2)
• новый элемент записывается по адресу, на который указывает `%esp`

```
.                .
.                .
.                .
+----------------+ 0x0000F040 <--- новая вершина стека (%esp)
|  новый элемент |
+----------------+ 0x0000F044 <--- старая вершина стека
|     данные     |
+----------------+ 0x0000F048
.                .
.                .
.                .
+----------------+ 0x0000FFFC
|     данные     |
+----------------+ 0x00010000 <--- дно стека
```

Когда мы достаём элемент из стека, эти действия совершаются в обратном порядке: (Принцип работы команды pop)

• элемент, на который указывает esp, записывается в регистр
• значение esp увеличивается на размер элемента в байтах (4 или 2)

```
.                 .
.                 .
.                 .
+-----------------+ 0x0000F040 <--- старая вершина стека
| верхний элемент | ------------------------------ записывается в регистр --->
+-----------------+ 0x0000F044 <--- новая вершина стека (%esp)
|      данные     |
+-----------------+ 0x0000F048
.                 .
.                 .
.                 .
+-----------------+ 0x0000FFFC
|      данные     |
+-----------------+ 0x00010000 <--- дно стека
```

## Память

Память — что про неё ещё скажешь? В C после вызова `malloc(3)` вам выделяется блок памяти, и вы производите доступ к нему доступ при помощи указателя, который содержит адрес этого блока. В ассемблере то же самое — после того, как вам выделили блок памяти, у вас есть его адрес для всевозможных манипуляций.

Правда, нужно отметить ещё одну деталь. Программный код расположен в памяти, поэтому вы тоже можете получить его адрес. Стек — это тоже блок памяти, и вы можете получить указатель на какой-то элемент стека, который находится под вершиной. Таким образом можно организовать доступ к произвольным элементам стека.

## Порядок байтов. Little-endian и big-endian

Оперативная память — это массив битовых значений, 0 и 1. О порядке битов в байте мы говорить не будем, так как вы не можете указать адрес отдельного бита, лишь только адрес байта, содержащего этот бит. А как в памяти располагаются байты в слове? Предположим, что у нас есть число 0x01020304. Его можно записать в виде байтовой последовательности:

```
начиная со старшего байта: 0x01 0x02 0x03 0x04          big-endian
начиная с младшего байта:  0x04 0x03 0x02 0x01          little-endian
```

Вот эта байтовая последовательность располагается в оперативной памяти, адрес всего слова в памяти — адрес первого байта этой последовательности.

Если первым располагается младший байт (мы начинаем запись с "меньшего конца") — такой порядок байт называется little-endian или интеловским. Именно он используется в процессорах x86.

Если первым располагается старший байт (мы начинаем запись с "большего конца") — такой порядок байт называется big-endian.

У little-endian есть одно важное достоинство. Посмотрите на запись числа 0x00000033:

```
0x33 0x00 0x00 0x00
```

Если мы прочитаем его как двухбайтовое значение, получим 0x0033. Если прочитаем как однобайтовое — получим 0x33. При записи этот трюк тоже работает. Конечно же, мы не можем прочитать число 0x11223344 как байт, потому что получим 0x44, что неверно. Поэтому считываемое число должно помещаться в целевой диапазон значений.

Ссылки по теме:

• http://en.wikipedia.org/wiki/Endianness
• http://ru.wikipedia.org/wiki/Порядокбайтов

# Глава 2. Hello, world!

При изучении нового языка принято писать самой первой программу, выводящую на экран строку "Hello, world!". Сейчас мы не ставим перед собой задачу понять всё написанное. Главное — посмотреть, как оформляются программы на ассемблере и научиться их компилировать.

Вспомним, как вы писали hello world на C. Скорее всего, приблизительно так:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  printf("Hello, world!\n");
  exit(0);
}
```

Вот только `printf(3)` — функция стандартной библиотеки C, а не операционной системы. Чем это плохо? — спросите вы. Да в общем всё нормально, но программируя на ассемблере вы наверное хотите взаимодействовать непосредственно с операционной системой, а не с библиотекой C. Это конечно же не значит, что из ассемблера нельзя вызывать функции библиотеки C. Просто мы пойдем более низкоуровневым путём.

Как вы уже наверное знаете, стандартный вывод (stdout), в который выводит данные `printf(3)`, является обычным файловым дескриптором, который заранее открывает для вас операционная система. Номер его дескриптора 1. Теперь нам на помощь придёт функция операционной системы `write(2)`.

```
WRITE(2)        Руководство программиста Linux        WRITE(2)

ИМЯ
        write - писать в файловый дескриптор

ОБЗОР
        #include <unistd.h>

        ssize_t write(int fd, const void *buf, size_t count);

ОПИСАНИЕ
        write пишет count байт в файл, на который ссылается файловый дескриптор fd
        из буфера, на который указывает buf.
```

А вот и сама программа:

```
#include <unistd.h>

int main(int argc, char* argv[])
{
  char str[] = "Hello, world!\n";
  write(1, str, sizeof(str) - 1);
  exit(0);
}
```

Почему sizeof(str) - 1? Потому что строка в C заканчивается нулевым байтом, а его нам печатать не нужно.

Теперь скопируйте следующий текст в файл `hello.s`. Файлы исходного кода на ассемблере имеют расширение `.s`.

```
.data                                   /* поместить следующее в сегмент данных         */

hello_str:                              /* наша строка                                  */
        .string "Hello, world!\n"

                                        /* длина строки                                 */
        .set hello_str_length, . - hello_str - 1

.text                                   /* поместить следующее в сегмент кода           */

.globl  main                            /* main -- глобальный символ, который видно
                                                  за пределами этого файла              */
.type   main, @function                 /* main -- функция (а не данные)                */


main:
        movl    $4, %eax                /* помещаем номер системного вызова write = 4
                                           в регистр %eax                               */

        movl    $1, %ebx                /* первый параметр -- в регистр %ebx
                                           номер файлового дескриптора stdout = 1       */

        movl    $hello_str, %ecx        /* второй параметр -- в регистр %ecx
                                           указатель на строку                          */

        movl    $hello_str_length, %edx /* третий параметр -- в регистр %edx
                                           длина строки                                 */

        int     $0x80                   /* вызываем прерывание 0x80                     */

        movl    $1, %eax                /* номер системного вызова exit = 1             */
        movl    $0, %ebx                /* передаём 0 как значение параметра            */
        int     $0x80                   /* вызываем exit(0)                             */
        .size   main, .-main            /* размер функции main                          */
```

Напоминаю, сейчас наша задача — скомпилировать первую программу. Подробное объяснение этого кода будет потом.

```
[user@host:~]$ gcc hello.s -o hello
[user@host:~]$
```

Если компиляция прошла успешно, GCC ничего не выводит на экран. Теперь запускаем нашу программу. Также проверяем, что она корректно завершилась со статусом 0.

```
[user@host:~]$ ./hello
Hello, world!
[user@host:~]$ echo $?
0
```

Теперь я советую прочитать главу про отладчик GDB. Он вам понадобится для исследования работы ваших программ. Возможно, сейчас вы не всё поймете, но эта глава специально расположена в конце, так как задумана больше как справочная, нежели обучающая. Для того, чтобы научиться работать с отладчиком, с ним нужно просто работать.

# Глава 3. Синтаксис ассемблера

Команды
Данные
Метки и прочие символы
Неинициализированные данные

## Команды

Команды ассемблера — это те инструкции, которые будет исполнять процессор. По сути, это самый низкий уровень программирования процессора. Каждая команда состоит из операции (что делать?) и операндов (аргументов). Операции мы будем рассматривать отдельно. А операнды у всех операций задаются в одном и том же формате. Операндов может быть от 0 (то есть нет вообще) до 3. В роли операнда могут выступать:

• Конкретное значение, известное на этапе компиляции — например, числовая константа или символ. Записываются при помощи знака $, например: `$0xf1`, `$10`, `$hello_str`. Эти операнды называются непосредственными.
• Регистр. Перед именем регистра ставится знак %, например: `%eax`, `%bx`, `%cl`.
• Указатель на ячейку в памяти (как он формируется и какой синтаксис записи — далее в этом разделе).
• Неявный операнд. Эти операнды не записываются непосредственно в исходном коде, а подразумеваются. Нет, компьютер не читает ваши мысли :) Просто некоторые команды всегда обращаются к определенным регистрам без явного указания, так как это входит в логику их работы. Такое поведение всегда описывается в документации.

### Важно

Если вы забудете знак $ когда записываете непосредственное числовое значение, компилятор будет интерпретировать его как абсолютный адрес. Это не вызовет ошибок компиляции, но скорее всего приведёт в Segmentation fault при выполнении.

Почти у каждой команды можно определить операнд-источник (команда читает его) и операнд-назначение (команда записывает в него результат). Общий синтаксис команды ассемблера такой:

```
операция источник, назначение
```

Для того, чтобы привести пример команды я, немного забегая наперед, расскажу об одной операции. Команда `mov источник , назначение` производит копирование `источника` в `назначение`. Возьмем строку из hello.s:

```
movl    $4, %eax                /* помещаем номер системного вызова write = 4
                                   в регистр %eax                              */
```

Как мы видим, источник — это непосредственное значение 4, а назначение — регистр `%eax`. Суффикс "l" в имени команды указывает на то, что ей следует работать с операндами длиной в 4 байта. Все суффиксы:

• "b" (от англ. byte) — 1 байт
• "w" (от англ. word) — 2 байта
• "l" (от англ. long) — 4 байта
• "q" (от англ. quad) — 8 байт

Таким образом, чтобы записать `$42` в регистр `%al` (а он имеет размер 1 байт):

```
movb  $42, %al
```

Важной особенностью всех команд является то, что они не могут работать с двумя операндами, находящимися в памяти. Хотя бы один из них следует сначала загрузить в регистр, а затем выполнять необходимую операцию.

Как формируется указатель на ячейку памяти? Синтаксис:

```
смещение(база, индекс, множитель)
```

Вычисленный адрес будет равен `база + индекс * множитель + смещение`. Множитель может равняться только 1, 2, 4 или 8. Например:

• (%ecx) адрес операнда находится в регистре `%ecx`. Этим способом удобно адресовать отдельные элементы в памяти: например, указатель на строку или указатель на int
• 4(%ecx) адрес операнда равен `%ecx` + 4. Удобно адресовать отдельные поля структур. Например, в `%ecx` адрес некоторой структуры, второй элемент которой находится "на расстоянии" 4 байта от её начала (или говорят "по смещению 4 байта")
• -4(%ecx) адрес операнда равен `%ecx` - 4
• foo(,%ecx,4) адрес операнда равен `foo` + `%ecx`*4, где `foo` — некоторый адрес. Удобно обращаться к элементам массива. Если `foo` — указатель на массив, элементы которого имеют размер 4 байта, то мы можем заносить в ecx номер элемента и таким образом обращаться к самому элементу

Ещё один важный нюанс: команды нужно помещать в секцию кода. Для этого перед командами нужно указать директиву `.text`. Вот так:

```
.text
        movl  $42, %eax
        ... ну и другие команды...
```

## Данные

Существуют директивы ассемблера, которые размещают в памяти данные, определенные программистом. Аргументы этих директив — список выражений, разделенных запятыми.

• `.byte` — размещает каждое выражение как 1 байт
• `.short` — 2 байта
• `.long` — 4 байта
• `.quad` — 8 байт

Например:

```
        .byte 0x10, 0xf5, 0x42, 0x55
        .long 0xaabbaabb
        .short -123, 456
```

Также существуют директивы для размещения в памяти строковых литералов:

• `.ascii "STR"` размещает строку "STR". Нулевых байтов не добавляет
• `.string "STR"` размещает строку "STR", после которой следует нулевой байт (как в языке C)
• У директивы `.string` есть синоним `.asciiz` (z от англ. zero — ноль, указывает на добавление нулевого байта)

Строка-аргумент этих директив может содержать стандартные escape-последовательности, которые вы использовали в C, например, "\n", "\r", "\t", "\\", "\"" и т. д.

Данные нужно помещать в секцию данных. Для этого перед данными нужно поместить директиву `.data`. Вот так:

```
.data
        .string "Hello, world\n"
        ... ещё много-много данных...
```

Если некоторые данные не предполагается изменять в ходе выполнения программы, их можно поместить в специальную секцию данных только для чтения при помощи директивы `.section .rodata`:

```
.section .rodata
        .string "program version 0.314"
```

Также приведу небольшую таблицу, которая сопоставляет типы данных в C на IA-32 и в ассемблере. Хочу заметить, что размер этих типов в C в других архитектурах может отличаться.

| Тип данных в C | Размер (sizeof), байт | Выравнивание, байт | Название |
|---|---|---|---|
| char | 1 | 1 | signed byte (байт со знаком) |
| signed char | | | |
| unsigned char | 1 | 1 | unsigned byte (байт без знака) |
| short | 2 | 2 | signed halfword (полуслово со знаком) |
| signed short | | | |
| unsigned short | 2 | 2 | unsigned halfword (полуслово без знака) |
| int | 4 | 4 | signed word (слово со знаком) |
| signed int | | | |
| long | | | |
| signed long | | | |
| enum | | | |
| unsigned int | 4 | 4 | unsigned word (слово без знака) |

| Тип данных в C | Размер (sizeof), байт | Выравнивание, байт | Название |
|---|---|---|---|
| unsigned long | | | |

Отдельных объяснений требует колонка "выравнивание". Выравнивание задано у каждого фундаментального типа данных (типа данных, которым процессор может оперировать непосредственно). Например, выравнивание word — 4 байта. Это значит, что данные типа word должны располагаться по адресу, кратному 4 (например, 0x00000100, 0x03284478). Архитектура рекомендует, но не требует выравнивания: доступ к невыровненным данным может быть медленнее, но принципиальной разницы нет и ошибки это не вызовет.

Для соблюдения выравнивания в распоряжении программиста есть директива `.p2align`.

```
        .p2align степень_двойки, заполнитель, максимум
```

Директива `.p2align` выравнивает текущий адрес до заданой границы. Граница выравнивания задаётся как степень числа 2: например, если вы указали .p2align 3 — следующее значение будет выровнено по 8-байтной границе. Для выравнивания размещается необходимое количество байт-заполнителей со значением `заполнитель`. Если для выравнивания требуется разместить более чем `максимум` байт-заполнителей, то выравнивание не выполняется.

Второй и третий аргумент являются необязательными.

Примеры:

```
.data
        .string "Hello, world\n"        /* мы вряд ли захотим считать, сколько символов занимает
    эта строка,
                                           и является ли следующий адрес выровненным */
        .p2align 2                      /* выравниваем по границе 4 байта для следующего .long */
        .long 123456
```

## Метки и прочие символы

Вы наверное заметили, что мы не присвоили имен нашим данным. Как же к ним обращаться? Очень просто — нужно поставить метку. Метка — это просто константа, значение которой — адрес.

```
hello_str:
        .string "Hello, world!\n"
```

Сама метка, в отличие от данных, места в памяти программы не занимает. Когда компилятор встречает в исходном коде метку, он запоминает текущий адрес и читает код дальше. В результате компилятор помнит все метки, и адреса, на которые они указывают. Программист может ссылаться на метки в своём коде. Существует специальная "метка", которая указывает на текущий адрес. Это метка "." (точка).

Значение метки как константы — это всегда адрес. А если вам нужна константа с каким-то другим значением? Тогда мы приходим к более общему понятию "символа". Символ — это просто некоторая константа. Причем он может быть определен в одном файле, а использован — в других.

Возьмём `hello.s` и скомпилируем его так:

```
[user@host:~]$ gcc -c hello.s
[user@host:~]$
```

Обратите анимание на параметр `-c`. Мы не компилируем исходный код в исполняемый файл, а лишь только в отдельный объектный файл `hello.o`. Теперь воспользуемся программой **nm(1)** :

```
[user@host:~]$ nm hello.o
00000000 d hello_str
0000000e a hello_str_length
```

```
00000000 T main
```

**nm(1)** выводит список символов в объектном файле. В первой колонке выводится значение символа, во второй — его тип, в третьей — имя. Возьмем символ `hello_str_length`. Это длина строки "Hello, world!\n". Значение символа четко определено и равно 0xe, об этом говорит тип "a" — Absolute value. А вот символ `hello_str` имеет тип "d" — значит он находится в секции данных (Data). Символ `main` находится в секции кода (Text section, тип "T"). А почему "a" была строчная буква, а "T" — заглавная? Если тип символа обозначен строчной буквой, значит это локальный символ, который видно только в пределах данного файла. Заглавная буква говорит о том, что символ глобальный и доступен другим модулям. Символ `main` мы сделали глобальным при помощи директивы `.globl main`.

Для создания нового символа используется директива `.set`. Синтаксис:

```
        .set символ, выражение
```

Например, определим символ `foobar` = 42:

```
        .set foobar, 42
```

Ещё пример из hello.s:

```
hello_str:                              /* наша строка                                  */
        .string "Hello, world!\n"

                                        /* длина строки                                 */
        .set hello_str_length, . - hello_str - 1
```

Сначала определяется символ `hello_str`, который содержит адрес строки. После этого мы определяем символ `hello_str_length`, который, судя по названию, содержит длину строки. Директива `.set` позволяет в качестве значения символа использовать арифметические выражения. Мы из значения текущего адреса (метка "точка") вычитаем адрес начала строки — получаем длину строки в байтах. Потом мы вычитаем ещё единицу, потому что директива `.string` добавляет в конце строки нулевой байт (а на экран мы его выводить не хотим).

## Неинициализированные данные

Часто требуется просто зарезервировать место в памяти для данных, без инициализации какими-то значениями. Например, у вас есть переменная, значение которой определяется параметрами командной строки. Действительно, вы вряд ли сможете дать ей какое-то осмысленное начальное значение, разве что 0. Такие данные называются неинциализированными и для них выделена специальная секция под названием `.bss`. В скомпилированной программе эта секция места не занимает. При загрузке программы в память секция неинициализированых данных будет заполнена нулевыми байтами.

Хорошо, но известные нам директивы размещения данных требуют указания инициализирующего значения. Поэтому для неинициализированных данных используются специальные директивы:

```
        .space количество_байт
        .space количество_байт, заполнитель
```

Директива `.space` резервирует `количество_байт` байт.

Также эту директиву можно использовать для размещения инициализированных данных, для этого существует параметр `заполнитель` — этим значением будет инициализирована память.

Например:

```
.bss
long_var_1:                     /* по размеру как .long */
```

```
        .space 4

buffer:                         /* какой-то буфер в 1024 байта */
        .space 1024

struct:                         /* какая-то структура размером 20 байт */
        .space 20
```

# Глава 4. Команды ассемблера

Команда mov
Команда lea
Команды для работы со стеком
Арифметика

Команда lea для арифметики
Команда loop
Команды сравнения и условные переходы. Безусловный переход
Произвольные циклы

Программа: поиск максимального элемента в массиве
Логическая арифметика
Подпрограммы

Программа: таблица умножения
Программа: вычисление факториала
Системные вызовы
Структуры

Программа: вывод размера файла
Программа: печать файла наоборот
Операции с цепочками данных

Пример: memcpy
Пример: strlen
Конструкция switch

Пример: интерпретатор языка brainfuck
Булевые выражения

# Команда mov

Синтаксис:

```
        mov источник, назначение
```

Команда `mov` производит копирование `источника` в `назначение`. Рассмотрим примеры:

```
/*
 * Это просто примеры использования команды mov,
 * ничего толкового этот код не делает
 */

.data
some_var:
        .long 0x00000072
other_var:
```

```
        .long 0x00000001, 0x00000002, 0x00000003

.text
.globl main
main:
        movl  $0x48, %eax               /* помещаем число 0x00000048 в %eax */

        movl  $some_var, %eax           /* помещаем в %eax значение метки some_var --
                                           то есть адрес числа в памяти
                                           например, у меня %eax стал равен 0x08049589 */

        movl  some_var, %eax            /* а теперь обращаемся к содержимому переменной
                                           в %eax теперь 0x00000072 */

        movl  other_var + 4, %eax       /* other_var указывает на 0x00000001
                                           размер одного long -- 4 байта
                                           значит, other_var + 4 указывает на 0x00000002
                                           в %eax теперь 0x00000002 */

        movl  $1, %ecx                  /* помещаем число 1 в %ecx */
        movl  other_var(,%ecx,4), %eax  /* помещаем в %eax первый (нумерация с нуля) элемент
    массива,
                                           пользуясь %ecx как индексным регистром */

        movl  $other_var, %ebx          /* в %ebx адрес other_var */
        movl  4(%ebx), %eax             /* обращаемся по адресу %ebx + 4
                                           в %eax снова 0x00000002 */

        movl  $other_var + 4, %eax      /* в %eax адрес, по которому расположен 0x00000002 */
        movl  $0x15, (%eax)             /* записываем по этому адресу число 0x00000015 */
```

Внимательно следите, когда вы загружаете адрес переменной, а когда обращаетесь к значению переменной по её адресу. Например:

```
        movl  other_var + 4, %eax       /* забыли знак $, в результате в %eax находится число
    0x00000002 */
        movl  $0x15, (%eax)             /* пытаемся записать по адресу 0x00000002 -> получаем
    Segmentaion fault */

        movl  0x48, %eax                /* забыли $, и пытаемся обратиться по адресу 0x00000048 ->
    Segmentaion fault */
```

## Команда lea

`lea` — мнемоническое от англ. Load Effective Address. Синтаксис:

```
        lea источник, назначение
```

Команда `lea` помещает адрес `источника` в `назначение`. Например:

```
.data
some_var:
        .long 0x00000072

.text
        leal  0x32, %eax                /* аналогично movl $0x32, %eax */
        leal  some_var, %eax            /* аналогично movl $some_var, %eax */

        leal  $0x32, %eax               /* вызовет ошибку при компиляции, так как $0x32 не является
    адресом */
        leal  $some_var, %eax           /* аналогично, ошибка компиляции */

        leal  4(%esp), %eax             /* помещаем в %eax адрес предыдущего элемента в стеке
                                           фактически, %eax = %esp + 4 */
```

## Команды для работы со стеком

Предусмотрено две специальные команды для работы со стеком, push (поместить в стек) и pop (достать из стека). Синтаксис:

```
push  источник
pop   назначение
```

При описании работы стека, я уже писал о принципе работы команды push и о принципе работы команды pop. Важный нюанс: push и pop работают только с операндами размером 4 и 2 байта. Если вы попробуете скомпилировать что-то вроде

```
pushb 0x10
```

GCC вам ответит, что:

```
[user@host:~]$ gcc test.s
test.s: Assembler messages:
test.s:14: Error: suffix or operands invalid for `push'
[user@host:~]$
```

Согласно ABI, в Linux стек выровнен по long. Сама архитектура этого не требует, это только соглашение между программами, но не рассчитывайте что другие библиотеки подпрограмм или операционная система захотят работать с невыровненным стеком. Что это всё значит? Если вы резервируете место в стеке, количество байт должно быть кратно размеру long, то есть 4. Например, вам нужно 2 байта в стеке под short. Но вы должны резервировать 4 байта для того, чтобы соблюдать выравнивание.

А теперь примеры:

```
.text
        pushl $0x10                     /* поместили в стек число 10 */
        pushl $0x20                     /* поместили в стек число 20 */
        popl  %eax                      /* достали 20 в %eax */
        popl  %ebx                      /* достали 10 в %ebx */

        pushl %eax
        popl  %ebx                      /* странный способ сделать movl %eax, %ebx */

        movl  $0x00000010, %eax
        pushl %eax                      /* положили в стек 4 байта*/
        popw  %ax                       /* достали 2 байта */
        popw  %bx                       /* и ещё 2 байта */
        /* в %ax находится 0x0010,
           в %bx находится 0x0000
           такой код сложен для понимания, его следует избегать */

        pushl %eax                      /* поместили %eax в стек -> %esp уменьшился на 4 */
        addl  $4, %esp                  /* увеличили %esp на 4, таким образом привели стек в
    исходное состояние */
```

Интересный вопрос: какое значение помещает в стек вот эта команда

```
pushl %esp
```

Если вы ещё раз посмотрите на алгоритм, то она должна поместить уже уменьшенное значение %esp. Однако в документации Intel [1] сказано, что в стек помещается такое значение %esp, каким оно было до выполнения команды — и она действительно работает именно так.

## Арифметика

Арифметических команд в нашем распоряжении довольно много. Синтаксис:

```
        inc   операнд
        dec   операнд

        add   источник, приёмник
        sub   источник, приёмник

        mul   множитель_1
```

Принцип работы:

```
inc: увеличивает операнд на 1
dec: уменьшает операнд на 1

add: приёмник = источник + приёмник
sub: приёмник = источник - приёмник
```

Команда `mul` имеет только один операнд. Второй сомножитель задаётся неявно. Он находится в регистре `%eax` и его размер выбирается в зависимости от суффикса команды (b, w или l). Место размещения результата также зависит от суффикса команды.

**Команда** — Второй сомножитель — Результат

**mulb** — %al — 16 бит: %ax

**mulw** — %ax — 32 бита: младшая часть в %ax, старшая в %dx

**mull** — %eax — 64 бита: младшая часть в %eax, старшая в %edx

Примеры:

```
.text
        movl  $72, %eax
        incl  %eax                      /* в %eax число 73 */
        decl  %eax                      /* в %eax число 72 */

        movl  $48, %eax
        addl  $16, %eax                 /* в %eax число 64 */

        movb  $5, %al
        movb  $5, %bl
        mulb  %bl                       /* в регистре %ax произведение %al * %bl = 25 */
```

Это скучно, — скажете вы. Хорошо. Давайте подумаем, какой будет результат выполнения следующего кода на C:

```
char x, y;
x = 250;
y = 14;
x = x + y;
printf("%d", (int) x);
```

Блльшинство сразу скажет, что результат (250 + 14 = 264) больше, чем может поместиться в одном байте. И что же напечатает программа? 8. Давайте рассмотрим, что происходит при сложении в двоичной системе.

```
    11111010       250
  + 00001110     + 14
  ----------      ---
  1 00001000      264
    |      |
    +------+
      = 8
```

Получается, что результат занимает 9 бит, а в переменную может поместиться только 8 бит. Это называется переполнение — перенос из старшего бита результата. В C переполнение не может быть перехвачено. Но в микропроцессоре эта ситуация регистрируется и её можно обработать. Когда происходит переполнение, устанавливается флаг cf. Команды условного прехода `jc` и `jnc` анализируют состояние этого флага. Команды условного перехода будут рассмотрены далее,

здесь эта информация приводится для полноты описания команд.

```
        movb  $0,   %ah                /* %ah = 0        */
        movb  $250, %al                /* %al = 250      */
        addb  $14,  %al                /* %al = %al + 14
                                          происходит переполнение, устанавливается флаг cf
                                          в %al число 8  */
        jnc   no_carry                 /* если переполнения не было, переходим на метку */
        movb  $1,   %ah                /* %ah = 1        */
no_carry:
        /* %ax = 264 = 0x0108 */
```

Этот код выдаёт правильную сумму в регистре `%ax` с учётом переполнения, если оно произошло. Попробуйте поменять числа в строках 2 и 3.

### Команда lea для арифметики

Для выполнения некоторых арифметических операций можно использовать команду `lea`. [2] Она вычисляет адрес своего операнда-источника и помещает этот адрес в операнд-назначение. Ведь она не производит чтение памяти по этому адресу, верно? А значит всё равно, что она будет вычислять: адрес или какие-то другие числа.

Вспомним, как формируется адрес операнда:

```
смещение(база, индекс, множитель)
```

Вычисленный адрес будет равен `база + индекс * множитель + смещение`.

Чем это нам удобно? Так мы можем получить команду с двумя операндами-источниками и одним результатом:

```
        movl  $10, %eax
        movl  $7, %ebx

        leal  5(%eax), %ecx            /* %ecx = %eax + 5 = 15 */
        leal  -3(%eax), %ecx           /* %ecx = %eax - 3 = 10 */
        leal  (%eax,%ebx), %ecx        /* %ecx = %eax + %ebx*1 = 17 */
        leal  (%eax,%ebx,2), %ecx      /* %ecx = %eax + %ebx*2 = 24 */
        leal  1(%eax,%ebx,2), %ecx     /* %ecx = %eax + %ebx*2 + 1 = 25 */
        leal  (,%eax,8), %ecx          /* %ecx = %eax*8 = 80 */
        leal  (%eax,%eax,2), %ecx      /* %eax = %eax + %eax*2 = %eax*3 = 30 */
        leal  (%eax,%eax,4), %ecx      /* %eax = %eax + %eax*4 = %eax*5 = 50 */
        leal  (%eax,%eax,8), %ecx      /* %eax = %eax + %eax*8 = %eax*9 = 90 */
```

Вспомните, что при сложении командой `add` результат записывается на место одного из слагаемых. Теперь, наверное, стало ясно главное преимущество `lea` в тех случаях, где её можно применить: она не перезаписывает операнды-источники. Как вы это сможете использовать — зависит только от вашей фантазии: прибавить константу к регистру и записать в другой регистр, сложить два регистра и записать в третий… Также `lea` можно применять для умножения регистра на 3, 5 и 9, как показано выше.

## Команда loop

Синтаксис:

```
        loop метка
```

Принцип работы:

• уменьшить значение регистра `%ecx` на 1

• если %ecx == 0, передать управление следующей за loop команде
• если %ecx != 0, передать управление на метку

Напишем программу для вычисления суммы чисел от 1 до 10 (конечно же, есть формула суммы арифметической прогрессии — но ведь это только пример).

```
.data
printf_format:
        .string "%d\n"

.text
.globl main
main:
        movl  $0, %eax                  /* в %eax будет результат, поэтому в начале обнулим его */
        movl  $10, %ecx                 /* 10 шагов цикла */

sum:
        addl  %ecx, %eax                /* %eax = %eax + %ecx */
        loop  sum

        /* %eax = 55, %ecx = 0 */

/*
 * следующий код выводит число в %eax на экран и завершает программу
 */
        pushl %eax
        pushl $printf_format
        call  printf
        addl  $8, %esp

        movl  $0, %eax
        ret
```

На C это бы выглядело так:

```
#include <stdio.h>

int main()
{
  int eax, ecx;
  eax = 0;
  ecx = 10;
  do
  {
    eax += ecx;
  } while(--ecx);
  printf("%d\n", eax);
}
```

## Команды сравнения и условные переходы. Безусловный переход

Команда loop неявно сравнивает регистр %ecx с нулём. Это довольно удобно для организации циклов, но часто циклы бывают намного сложнее, чем мы можем написать при помощи loop. Да и ещё нужен эквивалент конструкции if(){}. Вот команды, которые позволяют выполнять произвольные сравнения операндов:

```
        cmp операнд_2, операнд_1
```

Команда cmp выполняет вычитание операнд_1 - операнд_2 и устанавливает флаги. Результат вычитания нигде не запоминается.

## Важно

Обратите внимание на порядок операндов в записи команды: сначала второй, потом первый.

Сравнили, установили флаги, — и что дальше? А у нас есть целое семейство jump-команд, которые передают управление другим командам. Эти команды называются командами условного перехода. Каждой из них поставлено в соответствие условие, которое она проверяет. Синтаксис:

```
jcc метка
```

Команды `j _cc_` не существует, вместо `cc` нужно подставить мнемоническое обозначение условия.

**Мнемоника** — Английское слово — Смысл — Тип операндов

**e** — equal — равно — любые

**n** — not — отрицание — любые

**g** — greater — больше — без знака

**l** — less — меньше — без знака

**a** — above — больше — со знаком

**b** — below — меньше — со знаком

Таким образом, `je` проверяет, что операнды у команды сравнения были равны, `jl` проверяет, что `_операнд_1_ < _операнд_2_` и т. д. У каждой команды есть противоположная: просто добавляем букву "n":

• `je` — `jne`: равно — не равно
• `jg` — `jng`: больше — не больше

Теперь пример использования этих команд:

```
.text
        /* вот тут пропущен код, который получает некоторое
           значение в %eax
           пусть нас интересует случай, когда %eax == 15 */

        cmpl $15, %eax                  /* сравниваем */
        jne  not_equal:                 /* если не равно -- уходим на метку not_equal */

        /* тут мы окажемся только если переход не сработал,
           а значит %eax == 15 */

not_equal:
        /* а тут мы окажемся в любом случае */
```

Сравните с кодом на C:

```
if(eax == 15)
{
  /* тут мы окажемся только если переход не сработал,
     а значит %eax == 15 */
}
/* а тут мы окажемся в любом случае */
```

Кроме команд условного перехода, область применения которых ясна сразу, ещё существует команда безусловного перехода. Эта команда чем-то похожа на оператор `goto` языка C. Синтаксис:

```
jmp   адрес
```

Эта команда передаёт управление на `адрес` не проверяя никаких условий. Заметьте, то `адрес` может быть задан в виде непосредственного значения (метки), регистра или обращения к памяти.

## Произвольные циклы

Все инструкции для написания произвольных циклов я уже объяснил, поэтому сейчас я сначала приведу программу, а потом объяснение к ней. Почитайте её код и комментарии и попытайтесь разобраться, что она делает. Если сразу что-то непонятно — не страшно, сразу после исходного кода находится более подробное объяснение.

### Программа: поиск максимального элемента в массиве

Рассмотрим программу для нахождения максимального элемента массива. Сначала мы заносим в регистр %eax число 0. После этого мы сравниваем каждый элемент массива с текущим максимальным значением в %eax, и если этот элемент больше, то он становится текущим максимальным. После просмотра всего массива в %eax находится наибольший элемент.

```
.data
printf_format:
        .string "%d\n"

array:
        .long 10, 15, 148, 12, 151, 3, 72
array_end:

.text
.globl main
main:
        movl  $0, %eax                  /* в %eax будет результат
                                           в начале максимальное значение -- 0 */
        movl  $array, %ebx              /* в %ebx адрес текущего элемента массива */

loop_start:                             /* начало цикла */
        cmpl  %eax, (%ebx)              /* сравниваем текущий элемент массива и текущее
    максимальное значение в %eax */
        jbe   less                      /* если текущий элемент массива меньше или равен
    максимальному, обходим следующий код */
        movl  (%ebx), %eax              /* а вот если элемент массива больше максимального, значит
    он и есть новый максимум */
less:
        addl  $4, %ebx                  /* увеличиваем %ebx на размер одного элемента массива, 4
    байта */
        cmpl  $array_end, %ebx          /* сравниваем адрес текущего элемента и адрес конца массива
    */
        je    loop_end                  /* если они равны -- выходим из цикла */
        jmp   loop_start                /* иначе повторяем цикл снова */
loop_end:

/*
 * следующий код выводит число в %eax на экран и завершает программу
 */
        pushl %eax
        pushl $printf_format
        call  printf
        addl  $8, %esp

        movl  $0, %eax
        ret
```

Этот код соответствует приблизительно следующему на C:

```
#include <stdio.h>

int main()
{
  int array[] = { 10, 15, 148, 12, 151, 3, 72 };
  int *array_end = &array[sizeof(array) / sizeof(int)];
  int max = 0;
```

```
  int *p = array;

  do
  {
    if(*p > max)
    {
      max = *p;
    }
  } while(++p != array_end);

  printf("%d\n", max);
}
```

Возможно, такой способ обхода массива не очень привычен для вас. В C принято использовать переменную с номером текущего элемента, а не указатель на него. Никто не запрещает пойти этим же путём и на ассемблере:

```
.data
printf_format:
        .string "%d\n"

array:
        .long 10, 15, 148, 12, 151, 3, 72
array_end:

array_size:
        .long (array_end - array)/4     /* количество элементов массива */

.text
.globl main
main:
        movl  $0, %eax                  /* в %eax будет результат
                                           в начале максимальное значение -- 0 */
        movl  $0, %ecx                  /* начинаем просмотр с нулевого элемента */

loop_start:                             /* начало цикла */
        cmpl  %eax, array(,%ecx,4)      /* сравниваем текущий элемент массива и текущее
    максимальное значение в %eax */
        jbe   less                      /* если текущий элемент массива меньше или равен
    максимальному, обходим следующий код */
        movl  array(,%ecx,4), %eax      /* а вот если элемент массива больше максимального, значит
    он и есть новый максимум */
less:
        incl  %ecx                      /* увеличиваем на 1 номер текущего элемента */
        cmpl  array_size, %ecx          /* сравниваем номер текущего элемента и общее количество
    элементов */
        je    loop_end                  /* если они равны -- выходим из цикла */
        jmp   loop_start                /* иначе повторяем цикл снова */
loop_end:

/*
 * следующий код выводит число в %eax на экран и завершает программу
 */
        pushl %eax
        pushl $printf_format
        call  printf
        addl  $8, %esp

        movl $0, %eax
        ret
```

Рассматривая код этой программы, вы, наверное, уже поняли как создавать произвольный циклы в ассемблере, наподобие do{} while(); в C. Ещё раз повторю эту конструкцию, выкинув весь код, не относящийся к циклу:

```
loop_start:                             /* начало цикла */

        /* вот тут находится тело цикла */

        cmpl  ...                       /* что-то сравниваем для принятия решения о выходе из цикла
```

```
    */
        je    loop_end                  /* подбираем соответствующую команду условного перехода для
    выхода из цикла */
        jmp   loop_start                /* иначе повторяем цикл снова */
loop_end:
```

В C есть ещё один вид цикла, с проверкой условия перед входом в тело цикла: while(){}. Немного изменив предыдущий код, получаем следующее:

```
loop_start:                             /* начало цикла */
        cmpl  ...                       /* что-то сравниваем для принятия решения о выходе из цикла
    */
        je    loop_end                  /* подбираем соответствующую команду условного перехода для
    выхода из цикла */

        /* вот тут находится тело цикла */

        jmp   loop_start                /* переходим к проверке условия цикла */
loop_end:
```

Кто-то скажет: а ещё есть цикл for()! Но цикл

```
for(init; cond; incr)
{
  body;
}
```

эквивалентен такой конструкции:

```
init;
while(cond)
{
  body;
  incr;
}
```

Таким образом нам хватит и уже рассмотренных двух видов циклов.

## Логическая арифметика

Кроме обычных арифметических вычислений, мы можем выполнять и логические, то есть побитовые.

```
        and   источник, приёмник
        or    источник, приёмник
        xor   источник, приёмник
        not   операнд
        test  операнд_1, операнд_2
```

Команды `and`, `or` и `xor` ведут себя так же как и операторы языка C &, |, ^. Эти команды устанавливают флаги согласно результату.

Команда `not` инвертирует каждый бит операнда на противоположный, так же как и оператор языка C ~.

Команда `test` выполняет побитовое И над операндами, как и команда `and`, но в отличие от неё операнды не изменяет, а только устанавливает флаги. Её также называют командой логического сравнения, потому что с её помощью удобно проверять, установлены ли определённые биты. Например, так:

```
        testb $0b00001000, %al
        je    not_set
        /* нужные биты установлены */
not_set:
        /* биты не установлены */
```

Команду `test` можно применять для сравнения значения регистра с нулём:

```
        testl %eax, %eax
        je    is_zero
        /* %eax != 0 */
iz_zero:
        /* %eax == 0 */
```

Intel Optimization Manual рекомендует использовать `test` вместо `cmp` для сравнения регистра с нулём. [3]

Ещё следует упомянуть об одном трюке с `xor`. Как вы знаете, `_a_ XOR _a_ = 0`. Пользуясь этой особенностью, `xor` часто применяют для обнуления регистров:

```
        xorl  %eax, %eax
        /* теперь %eax == 0 */
```

Почему применяют `xor` вместо `mov`? Команда `xor` короче, а значит занимает меньше места в процессорном кеше, меньше времени тратится на декодирование и программа выполняется быстрее. Но эта команда устанавливает флаги. Поэтому, если вам нужно сохранить состояние флагов, применяйте `mov`. [4]

Иногда для обнуления регистра применяют команду `sub`. Помните, она тоже устанавливает флаги.

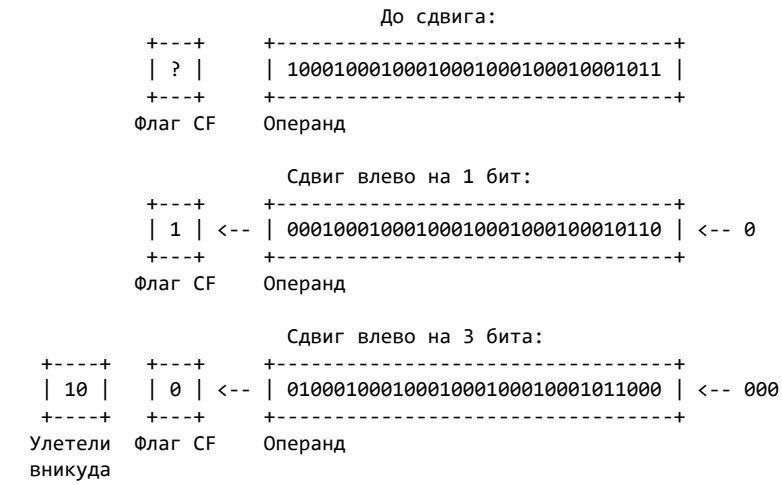
```
        subl  %eax, %eax
        /* теперь %eax == 0 */
```

К логическим командам также можно отнести команды сдвигов:

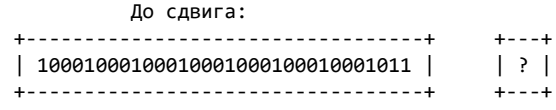
```
        /* Shift arithmetic left/Shift logical left */
        sal/shl количество_сдвигов, назначение

        /* Shift logical right */
        shr     количество_сдвигов, назначение

        /* Shift arithmetic right */
        sar     количество_сдвигов, назначение
```

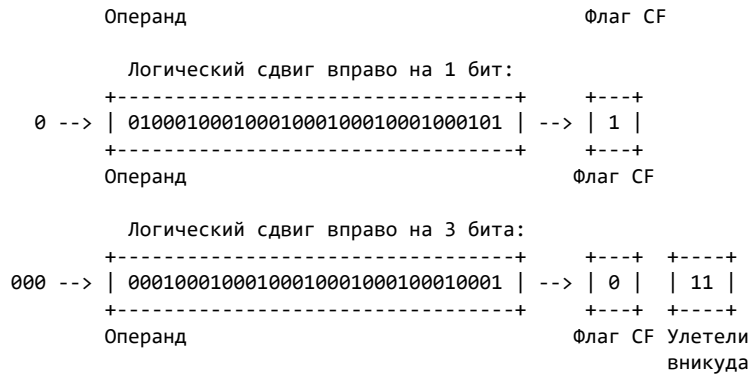
`количество_сдвигов` может быть задано непосредственным значением или находиться в регистре `%cl`. Учитываются только младшие 5 бит регистра `%cl` так что количество сдвигов может варьироваться в пределах от 0 до 31.

Принцип работы команды `shl`:



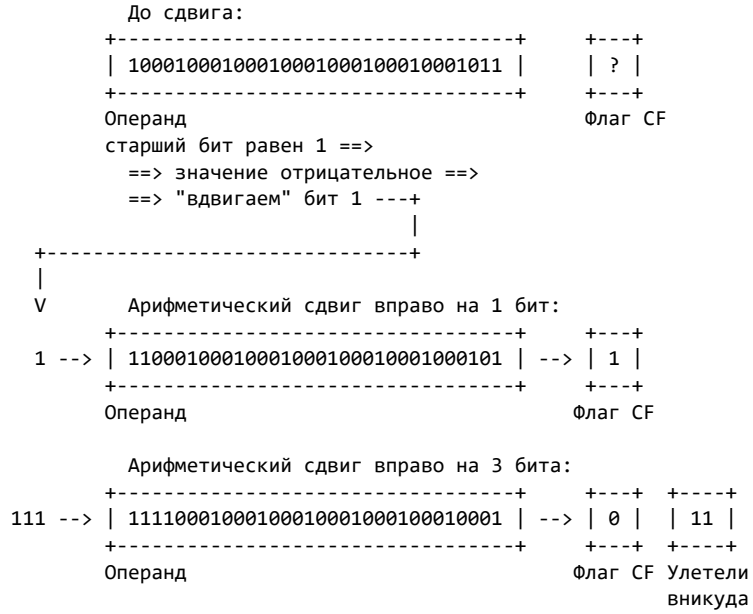

Принцип работы команды `shr`:






Эти две команды называются командами логического сдвига потому что они работают с операндом как с массивом бит. Каждый "выдвигаемый" бит попадает в флаг cf, причём с другой стороны операнда "вдвигается" бит 0. Таким образом, в регистре cf оказывается самый последний "выдвинутый" бит. Такое поведение вполне допустимо для работы с беззнаковыми числами, но числа со знаком будут обработаны неверно из-за того, что знаковый бит может быть потерян.

Для работы с числами со знаком существуют команды арифметического сдвига. Команды `shl` и `sal` выполняют полностью идентичные действия, так как при сдвиге влево знаковый бит не теряется (расширение знакового бита влево становится новым знаковым битом). Для сдвига вправо применяется команда `sar`. Она "вдвигает" слева знаковый бит исходного значения, таким образом сохраняя знак числа:



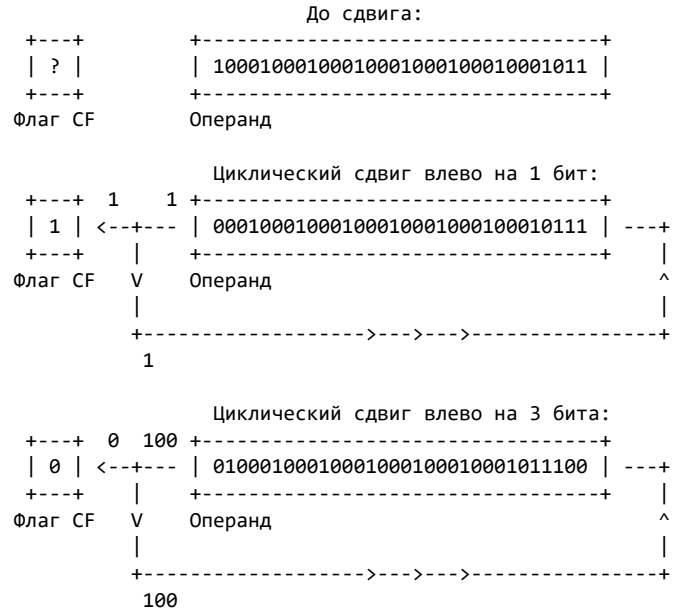

Многие программисты C знают об умножении и делении на степени двойки (2, 4, 8…) при помощи сдвигов. Этот трюк отлично работает и в ассемблере, используйте его для оптимизации.

Кроме сдвигов обычных, существуют циклические сдвиги:

```
/* Rotate right */
ror   количество_сдвигов, назначение

/* Rotate left */
rol   количество_сдвигов, назначение
```

Объясню на примере циклического сдвига влево на три бита: три старших ("левых") бита "выдвигаются" из регистра влево и "вдвигаются" в него справа. При этом в флаг CF записывается самый последний "выдвинутый" бит.

Принцип работы команды `rol`:

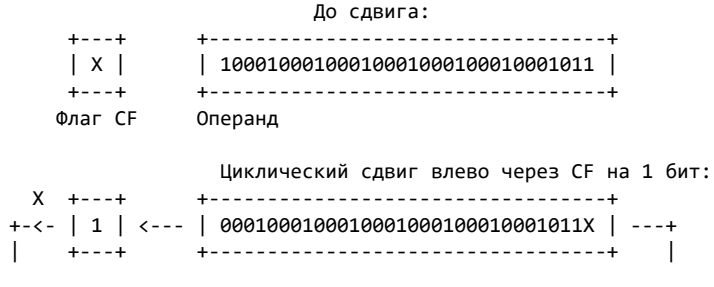


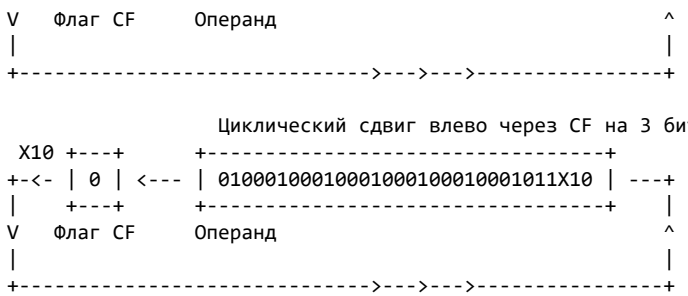

Принцип работы команды `ror`:


Существует ещё один вид сдвигов: циклический сдвиг через флаг CF. Эти команды рассматривают флаг CF как продолжение операнда.

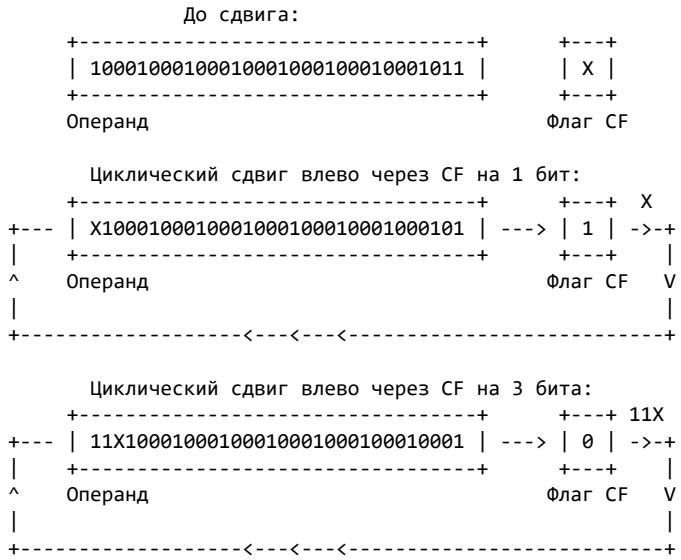
```
/* Rotate through carry right */
rcr   количество_сдвигов, назначение

/* Rotate through carry left */
rcl   количество_сдвигов, назначение
```

Принцип работы команды `rcl`:






Принцип работы команды `rcr`:




Эти сложные циклические сдвиги вам редко понадобятся в реальной работе. Но уже сейчас нужно знать что такие инструкции существуют, чтобы не изобретать велосипед потом. Ведь в языке C циклический сдвиг производится приблизительно так:

```
int main()
{
  int a = 0x11223344;
  int shift_count = 8;

  a = (a << shift_count) | (a >> (32 - shift_count));

  printf("%x\n", a);
}
```

## Подпрограммы

Под термином подпрограмма я буду подразумевать и функции (которые возвращают значение), и процедуры (которые void proc(...)). Подпрограммы нужны для достижения одной простой цели — избежать дублирования кода. В ассемблере есть две команды для организации подпрограмм.

```
call  метка
```

Используется для вызова подпрограммы, код которой находится по адресу `метка`. Принцип работы:

• Поместить в стек адрес следующей за `call` команды. Этот адрес называется адресом возврата.
• Передать управление на `метку`

Для возврата из подпрограммы используется команда `ret`

```
ret
```

```
        ret   число
```

Принцип работы:

• Извлечь из стека новое значение регистра `%eip` (то есть, передать управление на команду, расположенную по адресу из стека)
• Если команде передан операнд `число` , то `%esp` увеличивается на это `число`. Это необходимо для того, чтобы подпрограмма могла убрать из стека свои параметры.

Существует несколько способов передачи аргументов в подпрограмму.

• При помощи регистров. Перед вызовом подпрограммы вызывающий код помещает необходимые данные в регистры. У этого способа есть явный недостаток: число регистров ограничено, соответственно ограничено и максимальное число передаваемых параметров. Также если передать параметры почти во всех регистрах, подпрограмма будет вынуждена сохранять их в стек или память, так как ей может не хватить регистров для собственной работы. Несомненно, у этого способа есть и преимущество: доступ к регистрам очень быстрый.
• При помощи общей области памяти. Это похоже на глобальные переменные в C. Современные рекомендации написания кода (а часто и стандарты написания кода в больших проектах) запрещают этот метод. Он не поддерживает многопоточное выполнение кода. Он использует глобальные переменные неявным образом — смотря на опредение функции типа void func(void); невозможно сказать, какие глобальные переменные она изменяет и где ожидает свои параметры. Преимущества у этого метода вряд ли есть. Не используйте его без крайней необходимости.
• При помощи стека. Это самый популярный способ. Вызывающий код помещает аргументы в стек, а затем вызывает подпрограмму.

Рассмотрим передачу аргументов через стек подробнее. Предположим, нам нужно написать подпрограмму, принимающую три аргумента типа long (4 байта). Код:

```
sub:
        pushl %ebp
        movl  %esp, %ebp

        /* пролог закончен, можно начинать работу */

        subl  $8, %esp                          /* резервируем место для локальных переменных */


        movl  8(%ebp),  %eax                    /* что-то делаем с параметрами */
        movl  12(%ebp), %eax
        movl  16(%ebp), %eax


        /* эпилог */

        movl  %ebp, %esp
        popl  %ebp
        ret

main:
        pushl $0x00000010                       /* помещаем параметры в стек */
        pushl $0x00000020
        pushl $0x00000030
        call  sub                               /* вызываем подпрограмму */
        addl  $12, %esp
```

С вызовом всё ясно: помещаем аргументы в стек и даём команду `call`. А вот как в подпрограмме достать параметры из стека? Вспомним про регистр `%ebp`.

Мы сохраняем значение регистра `%ebp`, а затем записываем в него указатель на текущую вершину стека. Теперь у нас есть указатель на стек в известном состоянии. Сверху в стек можно помещать сколько угодно данных — но у нас останется доступ к параметрам. Часто эта последовательность команд называется "прологом" подпрограммы.

```
.                       .
.                       .
.                       .
+-----------------------+ 0x0000F040 <-- новое значение %ebp
| старое значение %ebp  |
+-----------------------+ 0x0000F044 <-- %ebp + 4
|    адрес возврата     |
+-----------------------+ 0x0000F048 <-- %ebp + 8
|      0x00000030       |
+-----------------------+ 0x0000F04C <-- %ebp + 12
|      0x00000020       |
+-----------------------+ 0x0000F050 <-- %ebp + 16
|      0x00000010       |
+-----------------------+ 0x0000F054
.                       .
.                       .
.                       .
```

Используя адрес в %ebp мы можем ссылаться на параметры:

```
 8(%ebp) = 0x00000030
12(%ebp) = 0x00000020
16(%ebp) = 0x00000010
```

Как видите, если идти от вершины стека в сторону аргументов, то мы будем встречать аргументы в обратном порядке по отношению к тому, как их туда поместили. Нужно сделать одно из двух: или помещать аргументы в обратном порядке (чтобы доставать их в прямом порядке), или учитывать обратный порядок аргументов в подпрограмме. В C принято при вызове помещать аргументы в обратном порядке. Так как операционная система Linux и большинство библиотек для неё написаны именно на C, я советую использовать "сишный" способ передачи аргументов и в ваших ассемблерных программах.

Подпрограмме могут понадобится собственные локальные переменные. Их принято держать в стеке. Для того, чтобы зарезервировать для них место, мы просто уменьшим регистр %esp на размер наших переменных. Предположим, что нам нужно 2 переменные типа long (4 байта), итого 2*4 = 8 байт. Мы уменьшим регистр %esp на 8. Теперь стек выглядит так:

```
.                         .
.                         .
.                         .
+-------------------------+ 0x0000F038 <-- %ebp - 8
| локальная переменная 2  |
+-------------------------+ 0x0000F03C <-- %ebp - 4
| локальная переменная 1  |
+-------------------------+ 0x0000F040 <-- %ebp
|  старое значение %ebp   |
+-------------------------+ 0x0000F044 <-- %ebp + 4
|     адрес возврата      |
+-------------------------+ 0x0000F048 <-- %ebp + 8
|       0x00000030        |
+-------------------------+ 0x0000F04C <-- %ebp + 12
|       0x00000020        |
+-------------------------+ 0x0000F050 <-- %ebp + 16
|       0x00000010        |
+-------------------------+ 0x0000F054
.                         .
.                         .
.                         .
```

Вы не можете делать никаких предположений о содержимом локальных переменных. Никто их для вас не инициализировал нулем. Можете для себя считать, что там находятся случайные значения.

При возврате из процедуры мы восстанавливаем старое значение %ebp из стека, потому что после возврата вызывающая функция вряд ли будет рада найти в регистре %ebp неизвестно что. Для этого, необходимо, чтобы старое значение %ebp было на вершине стека. Если подпрограмма что-то поместила в стек после старого %ebp, она должна это убрать. К счастью, мы не должны считать,

сколько байт мы поместили, сколько достали и сколько ещё осталось. Мы можем просто поместить значение регистра %ebp в регистр %esp, и стек станет точно таким же, как и после сохранения старого %ebp в начале подпрограммы. После этого команда ret возвращает управление вызывающему коду. Эта последовательность команд часто называется "эпилогом" подпрограммы.

## Важно

Сразу после того, как вы восстановили значение %esp в эпилоге, вы должны считать, что локальные переменные уничтожены. Хотя они ещё не перезаписаны, но они несомненно будут затёрты последующими командами push. Поэтому вы не должны сохранять указатели на локальные переменные дальше эпилога своей функции.

Остаётся одна маленькая проблема: в стеке всё ещё находятся аргументы для подпрограммы. Это можно решить одним из следующих способов:

• использовать команду ret с аргументом
• использовать необходимое число раз команду pop и выбросить результат
• увеличить %esp на размер всех помещенных в стек параметров

В C используется последний способ. Так как мы поместили в стек 3 значения типа long по 4 байта каждый, мы должны увеличить %esp на 12, что и делает команда addl сразу после call.

Заметьте, что не всегда обязательно выравнивать стек командой add. Если вы вызываете несколько подпрограмм подряд (но не в цикле!), то можно разрешить аргументам "накопиться" в стеке, а потом убрать их всех одной командой. Если ваша подпрограмма не содержит вызовов других подпрограмм в цикле и вы уверены, что оставшиеся аргументы в стеке не вызовут проблем переполнения стека, то аргументы можно не убирать вообще. Всё равно это сделает команда эпилога, которая восстанавливает %esp из %ebp. С другой стороны, если не уверены — лучше уберите аргументы, от 1 лишней команды программа медленнее не станет.

Строго говоря, все эти действия с %ebp не требуются. Вы можете использовать %ebp для хранения своих значений (никак не связанных со стеком), но тогда вам придётся обращаться к аргументам и локальным переменным через %esp или другие регистры, в которые вы поместите указатели. Трюк состоит в том, чтобы не изменять %esp после резервирования места для локальных переменных и аж до конца функции: так вы сможете использовать %esp на манер %ebp, как было показано выше. Не изменять %esp значит, что вы не сможете использовать push и pop (иначе все смещения переменных в стеке относительно %esp "поплывут"); вам понадобится создать необходимое число локальных переменных для хранения этих временных значений. С одной стороны, этот способ доступа к переменным немного сложнее, так как вы должны заранее просчитать, сколько места в стеке вам понадобится. С другой стороны, у вас появляется еще один свободный регистр %ebp. Так что если вы решите пойти этой дорогой, вы должны заранее продумать, сколько места для локальных переменных вам понадобится и дальше обращаться к ним через смещения относительно %esp.

И последнее: если вы хотите использовать вашу подпрограмму за пределами данного файла, не забудьте сделать её глобальной с помощью директивы .globl.

Посмотрим на код, который выводил содержимое регистра %eax на экран, вызывая функцию стандартной библиотеки C printf(3). Вы его уже видели в предыдущих программах, но там он был приведен без объяснений. Для справки привожу цитату из man:

```
PRINTF(3)            Linux Programmer's Manual            PRINTF(3)

NAME
       printf - formatted output conversion
SYNOPSIS
```

```
       #include <stdio.h>

       int printf(const char *format, ...);
```

```
.data
printf_format:
        .string "%d\n"

.text
        pushl %eax                      /* аргумент, подлежащий печати */
        pushl $printf_format            /* аргумент format */
        call  printf                    /* вызов printf() */
        addl  $8, %esp                  /* выровняли стек */
```

Обратите внимание на обратный порядок аргументов и очистку стека от аргументов.

## Важно

Значения регистров глобальны, вызывающая и вызываемая подпрограмма видят одни и те же регистры. Конечно же подпрограмма может изменять значения любых пользовательских регистров, но она обязана при возврате восстановить значения регистров `%ebp`, `%ebx`, `%esi`, `%edi` и `%esp`. Сохранение остальных регистров перед вызовом подпрограммы — задача программиста. Даже если вы заметили, что подпрограмма не изменяет какой-то регистр, это не повод его не сохранять. Ведь неизвестно как будут обстоять дела в следующей версии подпрограммы. Вы не должны делать каких-либо предположений о состоянии регистров на момент выхода из подпрограммы. Можете считать, что они содержат случайные значения.

Также внимания требует флаг df. При вызове подпрограмм флаг должен быть равен 0. Подпрограмма при возврате также должна установить флаг в 0. Коротко: если вам вдруг нужно установить этот флаг для какой-то операции, сбросьте его сразу как только он вам уже не нужен.

До этого момента мы обходились общим термином "подпрограмма". Но если подпрограмма — функция, она должна как-то передать возвращаемое значение. Это принято делать при помощи регистра `%eax`. Перед началом эпилога функция должна поместить в `%eax` возвращаемое значение.

## Программа: таблица умножения

Рассмотрим программу посложнее. Итак, программа для печати таблицы умножения. Размер таблицы умножения вводит пользователь. Нам понадобится вызвать функцию `scanf(3)` для ввода, `printf(3)` для вывода и организовать два вложенных цикла для вычислений.

```
.data
input_prompt:
        .string "enter size (1-255): "

scanf_format:
        .string "%d"

printf_format:
        .string "%5d "

printf_newline:
        .string "\n"

size:
        .long 0

.text
.globl main
main:
```

```
                                /* просим пользователя ввести размер таблицы */
        pushl $input_prompt     /* format */
        call  printf            /* вызов printf */

                                /* читаем размер таблицы в переменную size */
        pushl $size             /* указатель на переменную size */
        pushl $scanf_format     /* format */
        call  scanf             /* вызов scanf */

        addl  $12, %esp         /* выровняли стек одной командой сразу после двух функций */

        movl  $0, %eax          /* в регистре %ax команда mulb будет выдавать результат,
                                   но печатаем мы весь %eax
                                   поэтому два старших байта %eax должны быть нулевыми */

        movl  $0, %ebx          /* номер строки */

print_line:
        incl  %ebx              /* увеличиваем номер строки на 1 */
        cmpl  size, %ebx
        ja    print_line_end    /* если номер строки больше запрошенного размера --
    заканчиваем цикл */

        movl  $0, %ecx          /* номер колонки */

print_num:
        incl  %ecx              /* увеличиваем номер колонки на 1 */
        cmpl  size, %ecx
        ja    print_num_end     /* если номер колонки больше запрошенного размера --
    заканчиваем цикл */

        movb  %bl, %al          /* команда mulb ожидает второй операнд в %al */
        mulb  %cl               /* вычисляем %ax = %cl * %al */

        pushl %ebx              /* сохраняем используемые регистры перед вызовом printf */
        pushl %ecx

        pushl %eax              /* что печатать */
        pushl $printf_format    /* format */
        call  printf            /* вызов printf */
        addl  $8, %esp          /* выровняли стек */

        popl  %ecx              /* восстанавливаем регистры */
        popl  %ebx

        jmp   print_num         /* переходим в начало цикла */
print_num_end:

        pushl %ebx              /* сохраняем регистр */

        pushl $printf_newline   /* печатаем новую символ новой строки */
        call  printf
        addl  $4, %esp

        popl  %ebx              /* восстановили регистр */

        jmp   print_line        /* в начало цикла */
print_line_end:

        movl  $0, %eax          /* завершаем программу */
        ret
```

## Программа: вычисление факториала

Теперь напишем рекурсивную функцию для вычисления факториала. Она основана на следующей формуле:

0! = 1

n! = n * (n-1)!

```
  .data
  printf_format:
          .string "%d\n"

  .text
  /* int factorial(int) */
  factorial:
          pushl %ebp
          movl  %esp, %ebp

          /* достаём аргумент в %eax */
          movl  8(%ebp), %eax

          /* факториал 0 равен 1 */
          cmpl  $0, %eax
          jne   not_zero

          movl  $1, %eax
          jmp return
  not_zero:

          /* следующие 4 строки вычисляют:
             %eax = factorial(%eax - 1) */

          decl  %eax
          pushl %eax
          call  factorial
          addl  $4, %esp

          /* достаём в %ebx аргумент и вычисляем
             %eax = %eax * %ebx */

          movl  8(%ebp), %ebx
          mull  %ebx

          /* результат в паре %edx:%eax, но старшие 32 бита мы выбрасываем,
             так как они не помещаются в int */

  return:
          movl  %ebp, %esp
          popl  %ebp
          ret

  .globl main
  main:
          pushl %ebp
          movl  %esp, %ebp

          pushl $5
          call  factorial

          pushl %eax
          pushl $printf_format
          call  printf

          /* стек можно не выравнивать, это будет сделано во время эпилога */

          movl  $0, %eax                  /* завершаем программу */

          movl  %ebp, %esp
          popl  %ebp
          ret
```

Любой программист знает, что если существует очевидное итеративное (при помощи циклов) решение задачи, то ему следует отдавать предпочтение перед рекурсивным. Итеративный алгоритм нахождения факториала даже проще, чем рекурсивный; он следует из определения

факториала:

n! = 1 * 2 * … * n

Говоря проще, нужно перемножить все числа от 1 до n.

Функция на то и функция, что её можно заменить, при этом не изменяя вызывающий код. Для запуска следующего кода просто замените функцию из предыдущей программы вот этой новой версией:

```
factorial:
        movl  4(%esp), %ecx

        cmpl  $0, %ecx
        jne   not_zero

        movl  $1, %eax
        ret

not_zero:

        movl  $1, %eax
loop_start:
        mull  %ecx
        loop  loop_start

        ret
```

Что я сделал? Сначала я переписал рекурсию в виде цикла. Потом я увидел, что мне больше не нужен кадр стека, так как я не вызываю других функций и ничего не помещаю в стек. Поэтому я убрал пролог и эпилог, при этом регистр %ebp я не использую вообще. Если бы я его использовал, сначала я должен был бы сохранить его значение, и восстановить его перед возвратом.

В результате я так увлекся, что решил написать 64-битную версию этой функции. Она возвращает результат в паре %eax:%edx и может вычислить 20! = 2432902008176640000.

```
.data
printf_format:
        .string "%llu\n"

.text
/* long long int factorial(int) */
.type   factorial, @function
factorial:
        movl  4(%esp), %ecx

        cmpl  $0, %ecx
        jne   not_zero

        movl  $1, %eax
        ret

not_zero:

        movl  $1, %esi                  /* младшие 32 бита */
        movl  $0, %edi                  /* старшие 32 бита */
loop_start:
        movl  %esi, %eax                /* загружаем младшие биты для умножения */
        mull  %ecx                      /* %eax:%edx = младшие биты * %ecx */
        movl  %eax, %esi                /* записываем младшие биты обратно в %esi */

        movl  %edi, %eax                /* загружаем старшие биты */
        movl  %edx, %edi                /* записываем в %edi старшие биты предыдущего умножения,
                                           теперь результат умножения младших битов в %esi:%edi,
                                           а старшие биты в %eax для следующего умножения */
        mull  %ecx                      /* %eax:%edx = старшие биты * %ecx */
        addl  %eax, %edi                /* складываем полученный результат со старшими битами
    предыдущего умножения */
        loop  loop_start
```

```
        movl  %esi, %eax               /* результат возвращаем в паре %eax:%edx */
        movl  %edi, %edi

        ret
.size   factorial, .-factorial

.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        pushl $20
        call  factorial

        pushl %edi
        pushl %esi
        pushl $printf_format
        call  printf

        /* стек можно не выравнивать, это будет сделано во время эпилога */

        movl  $0, %eax                 /* завершаем программу */

        movl  %ebp, %esp
        popl  %ebp
        ret
```

Нам нужно умножить 64-битное число на 32-битное. Мы будем это делать как при умножении "в столбик":

```
             %edi       %esi
        *               %ecx
        ---------------------
        %edi*%ecx  %esi*%ecx
            A           |
           /|\          |
            +--<--<--<--+
           старших 32 бита
```

Но произведение `%esi * %ecx` не поместится в 32 бита, останется ещё старших 32 бита. Их мы должны прибавить к старшим 32-м битам результата. Приблизительно так вы это делаете на бумаге в десятичной системе:

```
          2  5            25 * 3 = 75
        *    3
          ----
            15
        + 6
          ----
          7  5
```

Задание: напишите программу-считалочку. Есть числа от 0 до m, которые располагаются по кругу. Счёт начинается с элемента 0. Каждый n-й элемент удаляют. Счёт продолжается с элемента, следующего за удалённым. Напишите программу, выводящую список вычеркнутых элементов. Подсказка: используйте `malloc(3)` для получения m+1 байт памяти и занесите в каждый байт число 1 при помощи `memset(3)`. Значение 1 означает, что элемент существует, значением 0 отмечайте удалённые элементы. При счете пропускайте удалённые элементы.

## Системные вызовы

Программа, которая не взаимодействует с внешним миром вряд ли может сделать что-то полезное. Вывести сообщение на экран, почитать данные из файла, установить сетевое соединение — это всё примеры действий, которые программа не может совершить без помощи операционной системы. В Linux пользовательский интерфейс ядра организован через системные вызовы. Системный вызов можно рассматривать как функцию, которую для вас выполняет операционная система.

Теперь наша задача состоит в том, чтобы разобраться, как происходит системный вызов. Каждый системный вызов имеет свой номер. Все они перечислены в файле `/usr/include/asm-i386/unistd.h`.

Системные вызовы считывают свои параметры из регисторв. Номер системного вызова нужно поместить в регистр `%eax`. Параметры помещаются в остальные регистры в таком порядке:

1. Первый — в `%ebx`
2. Второй — в `%ecx`
3. Третий — в `%edx`
4. Четвертый — в `%esi`
5. Пятый — в `%edi`
6. Шестой — в `%ebp`

Таким образом, используя все регистры общего назначения, можно передать максимум 6 параметров. Системный вызов производится вызовом прерывания 0x80. Такой способ вызова (с передачей параметров через регистры) называется fastcall. В других системах (например, *BSD) могут применяться другие способы вызова.

Следует отметить, что не следует использовать системные вызовы везде где только можно без особой необходимости. В разных версиях ядра порядок аргументов у некоторых системных вызовов отличается и это приводит к багам, которые довольно трудно найти. Поэтому стоит использовать функции стандартной библиотеки C, ведь их сигнатуры не изменяются, что обеспечивает портабельность кода на C. Почему бы нам не воспользоваться этим и не "заложить фундамент" переносимости наших ассемблерных программ? Только если вы пишете маленький участок самого нагруженного кода и для вас недопустим overhead, вносимый вызовом стандартной библиотеки C — только тогда стоит использовать системные вызовы напрямую.

В качестве примера можете посмотреть код программы Hello world.

## Структуры

Объявляя структуры в C вы не задумывались о том, как располагаются в памяти её элементы. В ассемблере понятия "структура" нет, зато есть "блок памяти", его адрес и смещение в этом блоке. Объясню на примере:

```
+------+------+------+------+
| 0x23 | 0x72 | 0x45 | 0x17 |
+------+------+------+------+
```

Пусть этот блок памяти размером 4 байта расположен по адресу 0x00010000. Это значит, что адрес байта 0x23 равен 0x00010000. Соответственно, адрес байта 0x72 равен 0x00010001. Говорят что байт 0x72 расположен по смещению 1 от начала блока памяти. Тогда байт 0x45 расположен по смещению 2, а байт 0x17 — по смещению 3. Таким образом, адрес элемента = базовый адрес + смещение.

Приблизительно так в ассемблере организована работа со структурами: к базовому адресу структуры прибавляется смещение, по которому находится нужный элемент. Теперь вопрос: как определить смещение? В C компилятор руководствуется следующими правилами:

• Вся структура должна быть выровнена так, как выровнен её элемент с наибольшим выравниванием.
• Каждый элемент находится по наименьшему следующему адресу с подходящим выравниванием. Если необходимо, для этого в структуру включается нужное число байт-заполнителей.
• Размер структуры должен быть кратен её выравниванию. Если необходимо, для этого в конец структуры включается нужное число байт-заполнителей.

Примеры (внизу указано смещение элементов в байтах; заполнители обозначены XX):

```
struct                  Выравнивание структуры: 1, размер: 1
{                       +----+
  char c;               | c  |
};                      +----+
                         0

struct                  Выравнивание структуры: 2, размер: 4
{                       +----+----+----+----+
  char c;               | c  | XX |    s    |
  short s;              +----+----+----+----+
};                       0         2

struct                  Выравнивание структуры: 4, размер: 8
{                       +----+----+----+----+----+----+----+----+
  char c;               | c  | XX   XX   XX |         i         |
  int i;                +----+----+----+----+----+----+----+----+
};                       0                   4

struct                  Выравнивание структуры: 4, размер: 8
{                       +----+----+----+----+----+----+----+----+
  int i;                |         i         | c  | XX   XX   XX |
  char c;               +----+----+----+----+----+----+----+----+
};                       0                   4

struct                  Выравнивание структуры: 4, размер: 12
{                       +----+----+----+----+----+----+----+----+----+----+----+----+
  char c;               | c  | XX   XX   XX |         i         |    s    | XX   XX |
  int i;                +----+----+----+----+----+----+----+----+----+----+----+----+
  short s;               0                   4                   8
};

struct                  Выравнивание структуры: 4, размер: 8
{                       +----+----+----+----+----+----+----+----+
  int i;                |         i         | c  | XX |    s    |
  char c;               +----+----+----+----+----+----+----+----+
  short s;               0                   4         6
};
```

Обратите внимание на два последние примера: элементы структур одни и те же, только расположены в разном порядке. Но размер структур получился разный!

## Программа: вывод размера файла

Напишем программу, которая выводит размер файла. Для этого потребуется вызвать функцию `stat(2)` и прочитать данные из структуры, которую она заполнит. man 2 stat:

```
STAT(2)                 Системные вызовы                STAT(2)

ИМЯ
       stat, fstat, lstat - получить статус файла

КРАТКАЯ СВОДКА
```

```
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *file_name, struct stat *buf);

ОПИСАНИЕ
       stat возвращает информацию о файле, заданном с помощью file_name и заполняет буфер buf.

       Все эти функции возвращают структуру stat, которая содержит такие поля:

              struct stat {
                  dev_t         st_dev;      /* устройство */
                  ino_t         st_ino;      /* inode */
                  mode_t        st_mode;     /* режим доступа */
                  nlink_t       st_nlink;    /* количество жестких ссылок */
                  uid_t         st_uid;      /* ID пользователя-владельца */
                  gid_t         st_gid;      /* ID группы-владельца */
                  dev_t         st_rdev;     /* тип устройства */
                                             /* (если это устройство) */
                  off_t         st_size;     /* общий размер в байтах */
                  unsigned long st_blksize;  /* размер блока ввода-вывода */
                                             /* в файловой системе */
                  unsigned long st_blocks;   /* количество выделенных блоков */
                  time_t        st_atime;    /* время последнего доступа */
                  time_t        st_mtime;    /* время последней модификации */
                  time_t        st_ctime;    /* время последнего изменения */
              };
```

Так, теперь осталось только вычислить смещение поля st_size… Но что это за типы — dev_t, ino_t? Какого они размера? Следует заглянуть в заголовочный файл и узнать, что обозначено при помощи typedef. Я сделал так:

```
[user@host:~]$ cpp /usr/include/sys/types.h | less
```

Далее, ищу в выводе препроцессора определение dev_t, нахожу:

```
typedef __dev_t dev_t;
```

Ищу __dev_t:

```
__extension__ typedef __u_quad_t __dev_t;
```

Ищу __u_quad_t:

```
__extension__ typedef unsigned long long int __u_quad_t;
```

Значит, sizeof(dev_t) = 8.

Мы бы могли и дальше продолжать искать, но в реальности всё немного по-другому. Если вы посмотрите на определение struct stat (cpp /usr/include/sys/stat.h | less), вы увидите поля с именами __pad1, __pad2, __unused4 и другие (зависит от системы). Эти поля не используются, они нужны для совместимости, и поэтому в man они не описаны. Так что самый верный способ не ошибиться — это просто попросить компилятор C посчитать это смещение для нас (вычитаем из адреса поля адрес структуры, получаем смещение):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  struct stat t;
  printf("sizeof = %d, offset = %d\n",
         sizeof(t),
         ((void *) &t.st_size) - ((void *) &t));
}
```

На моей системе программа напечатала "sizeof = 88, offset = 44". На вашей системе это значение может отличаться по описанным причинам. Теперь у нас есть все нужные данные об этой структуре, пишем программу:

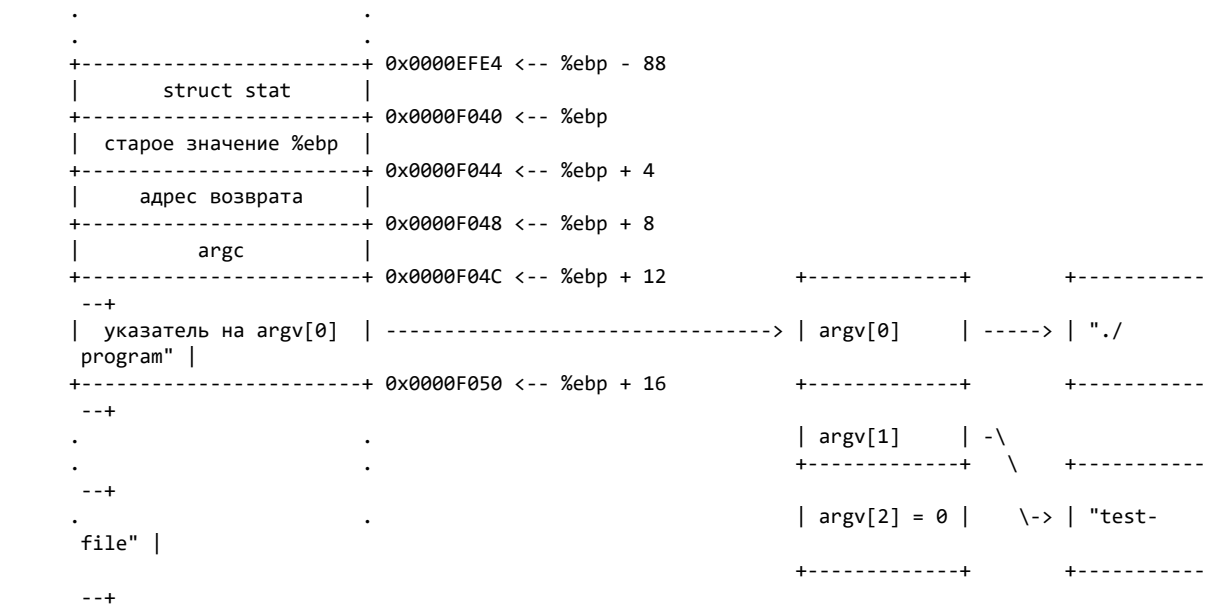
```
.data
str_usage:
        .string "usage: %s filename\n"

printf_format:
        .string "%u\n"

.text
.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        subl  $88, %esp                 /* выделили 88 байт под struct stat */

        cmpl  $2, 8(%ebp)               /* argc == 2? */
        je    args_ok
                                        /* нам передали не 2 аргумента, выводим usage */
        movl  12(%ebp), %ebx            /* помещаем в %ebx адрес массива argv  */
        pushl (%ebx)                    /* argv[0] */
        pushl $str_usage
        call  printf

        movl  $1, %eax                  /* выходим с кодом 1 */
        jmp   return

args_ok:
        leal  -88(%ebp), %ebx           /* помещаем адрес структуры в регистр %ebx */
        pushl %ebx

        movl  12(%ebp), %ecx            /* помещаем в %ecx адрес массива argv */
        pushl 4(%ecx)                   /* argv[1] -- имя файла */
        call  stat

        cmpl  $0, %eax                  /* stat() возвратил 0? */
        je    stat_ok

                                        /* stat() возвратил ошибку, вызываем perror(argv[1]) */
        movl  12(%ebp), %ecx
        pushl 4(%ecx)
        call  perror

        movl  $1, %eax
        jmp   return

stat_ok:
        pushl 44(%ebx)                  /* нужное нам поле по смещению 44 */
        pushl $printf_format
        call  printf

        movl  $0, %eax                  /* выходим с кодом 0 */

return:
        movl  %ebp, %esp
        popl  %ebp
        ret
```

Обратите внимание на обработку ошибок: если передано не 2 аргумента — выводим usage и выходим, если `stat(2)` вернул ошибку — выводим сообщение об ошибке и выходим.

Наверное, могут возникнуть некоторые сложности с пониманием, как расположены `argc` и `argv` в стеке. Допустим, вы запустили программу как

```
[user@host:~]$ ./program test-file
```

. .




Таким образом, в стек помещается два параметра: `argc` и указатель на первый элемент массива `argv[]`. Где-то в памяти расположен блок из трёх указателей: указатель на строку "./program", указатель на строку "test-file" и указатель NULL. Нам в стеке передали адрес этого блока памяти.

## Программа: печать файла наоборот

Напишем программу, которая читает со стандартного ввода всё до конца файла, а потом выводит введённые строки в обратном порядке. Для этого мы во время чтения будем помещать строки в связный список, а потом пройдем этот список в обратном порядке и напечатаем строки.

## Важно

Сохраните исходный код этой программы в файл с расширением `.S` — да, S в верхнем регистре.

```
.data
printf_format:
        .string "<%s>\n"

#define READ_CHUNK 128

.text

/* char *read_str(int *is_eof) */
read_str:
        pushl %ebp
        movl  %esp, %ebp

        pushl %ebx                      /* сохраняем регистры */
        pushl %esi

        movl  $0,   %ebx                /* прочитано байт */
        movl  $READ_CHUNK, %ecx         /* размер буфера */
        pushl %ecx
        call  malloc
        movl  %eax, %esi                /* указатель на начало буфера */
        decl  %ecx                      /* в конце должен быть нулевой байт, зарезервируем место
    для него */

        pushl stdin                     /* мы будем вызывать fgetc() всегда с этим аргументом */
1: /* read_start */
```

```
        call  fgetc                     /* читаем 1 символ */
        cmpl  $0xa, %eax                /* новая строка '\n'? */
        je    2f                        /* read_end */
        cmpl  $-1, %eax                 /* EOF? */
        je    4f                        /* eof_yes */
        movb  %al, (%esi,%ebx,1)        /* записываем прочитанный символ в буфер */
        incl  %ebx                      /* счётчик прочитанных байт */
        cmpl  %ecx, %ebx                /* буфер заполнен? */
        jne   1b                        /* read_start */

        addl  $READ_CHUNK, %ecx         /* увеличиваем размер буфера */
        pushl %ecx                      /* размер */
        pushl %esi                      /* указатель на буфер */
        call  realloc
        addl  $8, %esp                  /* убрали аргументы */
        movl  %eax, %esi                /* результат в %eax -- новый указатель */
        jmp   1b                        /* read_start */
2: /* read_end */

3: /* eof_no */
        movl  8(%ebp), %eax             /* *is_eof = 0 */
        movl  $0, (%eax)
        jmp   5f                        /* eof_end */
4: /* eof_yes */
        movl  8(%ebp), %eax             /* *is_eof = 1 */
        movl  $1, (%eax)
5: /* eof_end */

        movb  $0, (%esi,%ebx,1)         /* записываем в конец буфера '\0' */

        movl  %esi, %eax                /* результат в %eax */

        addl  $4, %esp                  /* убрали аргумент fgetc() */

        popl  %esi                      /* восстанавливаем регистры */
        popl  %ebx

        movl  %ebp, %esp
        popl  %ebp
        ret

/*
struct list_node
{
  struct list_node *prev;
  char *str;
};
*/

.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        subl  $4, %esp                  /* int is_eof; */

        movl  $0, %edi                  /* в %edi будем хранить указатель на предыдущую структуру */

1: /* read_start */
        leal  -4(%ebp), %eax            /* %eax = &is_eof; */
        pushl %eax
        call  read_str
        movl  %eax, %esi                /* указатель на прочитанную строку помещаем в %esi */

        pushl $8                        /* выделяем 8 байт под структуру */
        call  malloc

        movl  %edi, (%eax)              /* указатель на предыдущую структуру */
        movl  %esi, 4(%eax)             /* указатель на строку */

        movl  %eax, %edi                /* теперь эта структура -- предыдущая */
```

```
        addl  $8, %esp                  /* убрали аргументы */

        cmpl  $0, -4(%ebp)              /* is_eof == 0? */
        jne   2f
        jmp   1b
2: /* read_end */


3: /* print_start */
                                        /* теперь мы просматриваем список в обратном порядке,
                                           так что в %edi адрес текущей структуры */
        pushl 4(%edi)                   /* указатель на строку из текущей структуры */
        pushl $printf_format
        call  printf                    /* выводим на экран */

        addl  $4, %esp                  /* убрали из стека только $printf_format */
        call  free                      /* освободили память, которую занимает строка */

        pushl %edi                      /* указатель на структуру -- для освобождения памяти */
        movl  (%edi), %edi              /* заменяем указатель в %edi на следующий */
        call  free                      /* освобождаем память, которую занимает структура */

        addl  $8, %esp                  /* убрали аргументы */

        cmpl  $0, %edi                  /* адрес новой структуры == NULL? */
        je    4f
        jmp   3b
4: /* print_end */

        movl  $0, %eax                  /* выходим с кодом 0 */

return:
        movl  %ebp, %esp
        popl  %ebp
        ret
```

## Подсказка

Для того, чтобы послать с клавиатуры сигнал о конце файла, нажмите **Ctrl** - **D**.

```
[user@host:~]$ gcc print.S -o print
[user@host:~]$ ./print
aaa
bbbb
ccccc
^D<>
<ccccc>
<bbbb>
<aaa>
```

Заметьте, что мы ввели 4 строки: "aaa", "bbbb", "ccccc", "".

В этой программе я использовал некоторый новый синтаксис, но я надеюсь, что от этого программа не стала менее понятной. Во-первых, вы видите директиву препроцессора #define. Препроцессор C может быть использован для обработки исходного кода на ассемблере: нужно всего лишь использовать расширение файла с исходным кодом `.S`. А ещё я использовал метки-числа, причем некоторые из них повторяются в двух функциях. Почему бы не использовать текстовые метки как в предыдущих примерах? Можно, но они должны быть уникальными. Например, если бы я определил метку read_start и в функции read_str, и в main, GCC бы мне ответил:

```
[user@host:~]$ gcc -g print.S
print.S: Assembler messages:
print.S:85: Error: symbol `read_start' is already defined
```

Поэтому используя текстовые метки, приходится каждый раз придумывать уникальное имя. А можно использовать метки-числа, компилятор преобразует их в уникальные имена сам. Чтобы поставить метку, просто используйте любое положительное число в качестве имени. Чтобы сослаться на метку, которая определена ранее, используйте "Nb" (мнемоническое значение — backward), а чтобы сослаться на метку, которая определена дальше в коде, используйте "Nf" (мнемоническое значение — forward).

## Операции с цепочками данных

При обработке данных часто приходится иметь дело с цепочками данных. Цепочка, как подсказывает название, это массив данных: несколько переменных одного размера, расположенных друг за другом в памяти. В C вы использовали массив и индексную переменную, например argv[i]. Но в ассемблере для последовательной обработки цепочек есть специализированные команды. Синтаксис:

```
        lods
        stos
```

Странно, — скажет кто-то, — откуда эти команды знают, где брать данные и куда их записывать? Ведь у них и аргументов-то нет! Вспомните про регистры `%esi` и `%edi` и про их немного странные имена: "индекс источника" (source index) и "индекс приёмника" (destination index). Так вот, все цепочечные команды подразумевают, что в регистре `%esi` находится указатель на следующий необработанный элемент цепочки-источника, а в регистре `%edi` — указатель на следующий элемент цепочки-приёмника.

Направление просмотра цепочки задаётся флагом df: 0 — просмотр вперед, 1 — просмотр назад.

Итак, команда `lods` загружает элемент из цепочки-источника в регистр `%eax`/`%ax`/`%al` (размер регистра выбирается в зависимости от суффикса команды). После этого значение регистра `%esi` увеличивается или уменьшается (в зависимости от направления просмотра) на значение, равное размеру элемента цепочки.

Команда `stos` записывает содержимое регистра `%eax`/`%ax`/`%al` в цепочку-приёмник. После этого значение регистра `%edi` увеличивается или уменьшается (в зависимости от направления просмотра) на значение, равное размеру элемента цепочки.

Вот пример программы, которая работает с цепочечными командами. Конечно же, она занимается бестолковым делом, но в противном случае она была бы гораздо сложнее. Она увеличивает каждый байт строки str_in на 1, то есть заменяет a на b, b на c и так далее.

```
.data
printf_format:
        .string "%s\n"

str_in:
        .string "abc123()!@!777"
        .set str_in_length, .-str_in

.bss
str_out:
        .space str_in_length

.text
.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        movl  $str_in, %esi             /* цепочка-источник */
```

```
        movl  $str_out, %edi            /* цепочка-приёмник */

        movl  $str_in_length - 1, %ecx  /* длина строки без нулевого байта (нулевой байт не
    обрабатываем) */
1:
        lodsb                           /* загружаем байт из источника в %al */
        incb  %al                       /* производим какую-то операцию с %al */
        stosb                           /* сохраняем %al в приёмнике */
        loop  1b

        movsb                           /* копируем нулевой байт */

        /* важно: сейчас %edi указывает на конец цепочки-приёмника */

        pushl $str_out
        pushl $printf_format
        call  printf                    /* выводим на печать */

        movl  $0, %eax

        movl  %ebp, %esp
        popl  %ebp
        ret

[user@host:~]$ ./stringop
bcd234)*"A"888
[user@host:~]$
```

Но с цепочками мы часто выполняем довольно стандартные действия. Например, при копировании блоков памяти мы просто пересылаем байты из одной цепочки в другую, без обработки. При сравнении строк мы сравниваем элементы двух цепочек. При вычислении длины строки в C мы считаем байты до тех пор пока не встретим нулевой байт. Эти действия очень просты, но в тоже время используются очень часто, поэтому были введены следующие команды:

```
movs
cmps
scas
```

Размер элементов цепочки, которые обрабатывают эти команды, зависит от использованного суффикса команды.

Команда `movs` выполняет копирование одного элемента из цепочки-источника в цепочку-приёмник.

Команда `cmps` выполняет сравнение элемента из цепочки-источника и цепочки-приёмника (фактически, как и `cmp`, выполняет вычитание, `источник - приёмник` , результат никуда не записывается, но флаги устанавливаются).

Команда `scas` предназначена для поиска определённого элемента в цепочке. Она сравнивает содержимое регистра `%eax`/`%ax`/`%al` и содержимое элемента цепочки (выполняется вычитание `%eax`/`%ax`/`%al - элемент_цепочки` , результат не записывается, но флаги устанавливаются). Адрес цепочки должен быть помещён в регистр `%edi`.

После того, как эти команды выполнили своё основное действие, они увеличивают/уменьшают индексные регистры на размер элемента цепочки.

Подчеркну тот факт, что эти команды обрабатывают только один элемент цепочки. Таким образом, нужно организовать что-то вроде цикла для обработки всей цепочки. Для этих целей существуют префиксы команд:

```
rep
repe/repz
repne/repnz
```

Эти префиксы ставятся перед командой, например: `repe scas`. Префикс организовывает как бы цикл из одной команды, при этом с каждым шагом цикла значение регистра `%ecx` автоматически уменьшается на 1.

• rep повторяет команду пока %ecx не равен нулю.
• repe (или repz — то же самое) повторяет команду пока %ecx не равен нулю и установлен флаг zf. Анализируя значение регистра %ecx можно установить точную причину выхода из цикла: если %ecx равен нулю — значит, zf всегда был установлен и вся цепочка пройдена до конца, если %ecx больше нуля — значит, флаг zf в какой-то момент был сброшен.
• repne (или repnz — то же самое) повторяет команду пока %ecx не равен нулю и не установлен флаг zf.

Также следует указать команды для управления флагом df:

```
        cld
        std
```

cld (clear direction flag) сбрасывает флаг df.

std (set direction flag) устанавливает флаг df.

## Пример: memcpy

Вооружившись новыми знаниями попробуем заново изобрести функцию memcpy:

```
.data
printf_format:
        .string "%s\n"

str_in:
        .string "abc123()!@!777"
        .set str_in_length, .-str_in

.bss
str_out:
        .space str_in_length

.text

/* void *my_memcpy(void *dest, const void *src, size_t n); */

my_memcpy:
        pushl %ebp
        movl  %esp, %ebp

        pushl %esi
        pushl %edi

        movl  8(%ebp), %edi             /* цепочка-назначение */
        movl  12(%ebp), %esi            /* цепочка-источник */
        movl  16(%ebp), %ecx            /* длина */

        rep movsb

        movl  8(%ebp), %eax             /* возвращаем dest */

        popl  %edi
        popl  %esi

        movl  %ebp, %esp
        popl  %ebp
        ret

.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        pushl $str_in_length
```

```
        pushl $str_in
        pushl $str_out
        call  my_memcpy

        pushl $str_out
        pushl $printf_format
        call  printf

        movl  $0, %eax

        movl  %ebp, %esp
        popl  %ebp
        ret
```

Вы наверное будете удивлены если я вам скажу, что эта реализация `memcpy` всё равно не самая быстрая? Что ещё можно сделать, спросите вы. Ведь мы можем копировать данные не по одному байту, а целых 4 байта за раз при помощи `movsl`. Тогда у нас получается приблизительно такой алгоритм: копируем как можно больше данных блоками по 4 байта, после этого остаётся хвостик в 0, 1, 2 или 3 байта; этот остаток можно скопировать при помощи `movsb`. Поэтому нашу `memcpy` лучше переписать вот так:

```
/* void *my_memcpy(void *dest, const void *src, size_t n); */

my_memcpy:
        pushl %ebp
        movl  %esp, %ebp

        pushl %esi
        pushl %edi

        movl  8(%ebp), %edi             /* цепочка-назначение */
        movl  12(%ebp), %esi            /* цепочка-источник */
        movl  16(%ebp), %edx            /* длина */

        movl  %edx, %ecx
        shrl  $2, %ecx                  /* делим на 2^2 = 4, теперь в %ecx количество 4-байтных
    кусочков */
        rep movsl

        movl  %edx, %ecx
        andl  $3, %ecx                  /* $3 == $0b11, оставляем только два младших бита,
                                           то есть остаток от деления на 4 */
        jz    1f                        /* если результат 0, обходим цепочечную команду */
        rep movsb
1:

        movl  8(%ebp), %eax             /* возвращаем dest */

        popl  %edi
        popl  %esi

        movl  %ebp, %esp
        popl  %ebp
        ret
```

## Пример: strlen

Теперь `strlen`: нам нужно сравнить каждый байт цепочки с 0, остановиться когда найдём 0 и возвратить количество ненулевых байт. Как счетчик мы будем использовать регистр `%ecx`, который автоматически изменяют все префиксы. Но префиксы уменьшают счетчик и выходят из прекращают выполнение команды когда `%ecx` равен 0. Поэтому перед цепочечной командой мы поместим в `%ecx` число `0xffffffff` и этот регистр будет уменьшатся в ходе выполнения цепочечной команды. Результат у нас получится в дополнительном коде, поэтому мы используем команду `not` для инвертирования всех битов. И потом ещё уменьшим результат на 1 — нулевой байт.

```
.data
printf_format:
        .string "%u\n"

str_in:
        .string "abc123()!@!777"

.text

/* size_t my_strlen(const char *s); */

my_strlen:
        pushl %ebp
        movl  %esp, %ebp

        pushl %edi

        movl  8(%ebp), %edi             /* цепочка */

        movl  $0xffffffff, %ecx
        xorl  %eax, %eax                /* %eax = 0 */

        repne scasb

        notl  %ecx
        decl  %ecx

        movl  %ecx, %eax

        popl  %edi

        movl  %ebp, %esp
        popl  %ebp
        ret

.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        pushl $str_in
        call  my_strlen

        pushl %eax
        pushl $printf_format
        call  printf

        movl  $0, %eax

        movl  %ebp, %esp
        popl  %ebp
        ret
```

Как реализованы другие стандартные цепочечные функции можно посмотреть файле `/usr/src/linux/include/asm-i386/string.h`. Оттуда я взял все примеры для этого раздела.

В заключение обсуждения цепочечных команд хочу сказать следующее: не следует заново изобретать стандартные функции, как мы это только что сделали. Это всего лишь пример и объяснение как они работают. В реальных программах используйте цепочечные команды когда производите нестандартную обработку цепочек и цепочечные команды вам реально смогут помочь.

## Конструкция switch

Оператор switch языка C можно переписать на ассемблере несколькими способами. Рассмотрим несколько вариантов, какие могут быть значения у case:

• значения, из определённого маленького промежутка (все или почти все), например 23, 24, 25, 27, 29, 30
• значения, между которыми большие "расстояния" на числовой прямой, например 5, 15, 80, 3800
• комбинированый вариант: 35, 36, 37, 38, 39, 1200, 1600, 7000

Рассмотрим решение для первого случая. Вспомним, что команда `jmp` принимает адрес не только в виде непосредственного значения (метки), но как обращение к памяти. Значит, мы можем осуществлять переход на адрес, вычисленный в процессе выполнения. Теперь вопрос: как можно вычислить адрес? А нам не нужно ничего вычислять, мы просто поместим все адреса case-веток в массив. Пользуясь проверяемым значением как индексом массива, выбираем нужный адрес case-ветки. Таким образом, процессор всё вычислит за нас. Посмотрите на следующий код:

```
.data
printf_format:
        .string "%u\n"

.text
.globl main

main:
        pushl %ebp
        movl  %esp, %ebp

        movl  $1, %eax                  /* получаем в %eax некоторое интересующее нас значение */

                                        /* мы предусмотрели случаи только для 0, 1, 3, поэтому */
        cmpl  $3, %eax                  /* если %eax больше 3 (как беззнаковое), */
        ja    case_default              /* то переходим к default */

        jmp   *jump_table(,%eax,4)      /* переходим по адресу, содержащемуся в памяти jump_table +
    %eax*4 */

.section .rodata
        .p2align 4
jump_table:                             /* массив адресов                                  */
        .long case_0                    /* адрес этого элемента массива: jump_table + 0  */
        .long case_1                    /*                               jump_table + 4  */
        .long case_default              /*                               jump_table + 8  */
        .long case_3                    /*                               jump_table + 12 */
.text

case_0:
        movl $5, %ecx                   /* тело case-блока */
        jmp switch_end                  /* как break, переходим в конец switch */

case_1:
        movl $15, %ecx
        jmp switch_end

case_3:
        movl $35, %ecx
        jmp switch_end
```

```
case_default:
        movl $100, %ecx

switch_end:

        pushl %ecx                      /* выводим %ecx на экран, выходим */
        pushl $printf_format
        call  printf

        movl  $0, %eax

        movl  %ebp, %esp
        popl  %ebp
        ret
```

Этот код эквивалентен следующему коду на C:

```
#include <stdio.h>

int main()
{
  unsigned int a, c;

  a = 1;
  switch(a)
  {
    case 0:
      c = 5;
      break;

    case 1:
      c = 15;
      break;

    case 3:
      c = 35;
      break;

    default:
      c = 100;
      break;
  }

  printf("%u\n", c);
}
```

Смотрите: в секции `.rodata` (данные только для чтения) создаётся массив из 4 значений. Мы обращаемся к нему как к обычному массиву, индексируя его по `%eax`: `jump_table(,%eax,4)`. Но зачем перед этим стоит звёздочка? Она обозначает, что мы хотим перейти по адресу, содержащемуся в памяти по адресу `jump_table(,%eax,4)`. (Если бы её не было — мы бы начали исполнять данные как код.)

Заметьте, что тут нам понадобились значения 0, 1, 3, которые укладываются в маленький промежуток [0; 3]. Так как для значения 2 не предусмотрено особой обработки, в массиве адресов `jump_table` индексу 2 соответствует `case_default`. Перед тем, как сделать `jmp`, нужно обязательно проверить, что проверяемое значение входит в наш промежуток, и если не входит — то перейти на default. Если вы этого не сделаете, то когда попадётся значение за пределами массива, программа в лучшем случае получит Segmentation fault, а в худшем (если рядом с этим масивом вдресов в памяти окажется еще один массив адресов) — код продолжит исполнение вообще непонятно где.

Теперь рассмотрим случай, когда значения для веток case находятся на большом расстоянии друг от друга. Очевидно, что способ с массивом сдресов не подходит, иначе массив бы занимал большое количество памяти и содержал в основном адреса ветки default. В этом случае лучшее, что может сделать программист — это выразить switch как последовательное сравнение со всеми перечисленными значениями. Если значений довольно много, то придётся применить немного

логики: приблизительно прикинуть, какие ветки будут исполняться чаще всего, и отсортировать их в таком порядке в коде. Это нужно для того, чтобы наиболее часто исполняемые ветки испольнялись после маленького числа сравнений. Допустим, у нас есть варианты 5, 38, 70 и 1400, причём 70 будет появляться чаще всего:

```
.data
printf_format:
        .string "%u\n"

.text
.globl main

main:
        pushl %ebp
        movl  %esp, %ebp

        movl  $70, %eax                 /* получаем в %eax некоторое интересующее нас значение */

        cmpl  $70, %eax
        je    case_70

        cmpl  $5, %eax
        je    case_5

        cmpl  $38, %eax
        je    case_38

        cmpl  $1400, %eax
        je    case_1400

case_default:
        movl  $100, %ecx
        jmp   switch_end

case_5:
        movl  $5, %ecx
        jmp   switch_end

case_38:
        movl  $15, %ecx
        jmp   switch_end

case_70:
        movl  $25, %ecx
        jmp   switch_end

case_1400:
        movl  $35, %ecx

switch_end:

        pushl %ecx

        pushl $printf_format
        call  printf

        movl  $0, %eax

        movl  %ebp, %esp
        popl  %ebp
        ret
```

Единственное, на что хочется обратить внимание — это расположение ветки default: если все сравнения оказались ложными, то код default выполняется "автоматически".

Наконец, третий, комбинированый, вариант. Путь варианты 35, 36, 37, 39, 1200, 1600 и 7000. Тогда мы видим промежуток [35; 39] и ещё три числа. Тогда код будет выглядеть приблизительно так:

```
        movl  $1, %eax                  /* получаем в %eax некоторое интересующее нас значение */
```

```
        cmpl  $35, %eax
        jb    case_default

        cmpl  $39, %eax
        ja    switch_compare

        jmp   *jump_table-35(,%eax,4)

.section .rodata
        .p2align 4
jump_table:
        .long case_35
        .long case_36
        .long case_37
        .long case_default
        .long case_39
.text

switch_compare:
        cmpl  $1200, %eax
        jmp   case_1200

        cmpl  $1600, %eax
        jmp   case_1600

        cmpl  $7000, %eax
        jmp   case_7000

case_default:
        /* ... */
        jmp   switch_end

case_35:
        /* ... */
        jmp   switch_end

        ... ещё код ...
switch_end:
```

Снова-таки, имеет смысл переставить некоторые части этого кода, если вы заранее знаете, какие значения вам придётся обрабатывать чаще всего.

## Пример: интерпретатор языка brainfuck

brainfuck (далее сокращённо bf) — это эзотерический язык программирования, то есть, язык, предназначенный не для практического применения, а придуманный как головоломка, как задача, которая заставляет программиста думать нестандартно. Команды bf управляют массивом целых чисел с неограниченным набором ячеек, есть понятие текущей ячейки.

• Команды < и > дают возможность перемещаться по массиву на одну ячейку влево и, соответственно, вправо.
• Команды + и - увеличивают и соответственно, уменьшают, содержимое текущей ячейки на 1.
• Команда . выводит содержимое текущей ячейки на экран как один символ; команда , читает один символ и помещает его в текущкю ячейку.
• Команды циклов [ и ] должны всегда находиться в парах и соблюдать правила вложенности (как скобки в математических выражениях). Команда [ сравнивает значение текущей ячейки с 0: если оно равно 0, то выполняется команда, следующая за соответствующей ], если не 0, то просто выполняется следующая команда. Команда ] передаёт управление на соответствующую ].
• Остальные символы в коде программы являются комментариями и их следует пропускать.

В начальном состоянии все ячейки содержат значение 0, а текущей является крайняя левая ячейка.

Вот несколько программ с объяснениями:

```
+>+>+           Устанавливает первые три ячейки в 1
[-]             Обнулаяет текущкю ячейку
[>>>+<<<-]      Перемещает значение текущей ячейки в ячейку, расположеную "тремя шагами правее"
```

Интерпретация программы состоит из двух шагов: загрузка программы и собственно исполнение. Во время загрузки следует проверить корректность программы (соответствие []) и расположить код программы в памяти в удобном для выполнения виде. Для этого каждой команде присвоен номер операции, начиная с 0 — для того, чтобы можно было выполнять операции при помощи помощи перехода по массиву адресов, как в switch.

Большинство программ на bf содержат последовательности одинаковых команд < > + -, которые можно выполнять не по одной, а все сразу. Например, выполняя код +++++ можно выполнить пять раз увеличение на 1, или один раз увеличение на 5. Таким образом, довольно простыми средствами можно сильно оптимизировать выполнение программы.

Вот программы, которые вызовут ошибки загрузки программы:

```
[       No matching ']' found for a '['
]       No matching '[' found for a ']'
```

А эти программы вызовут ошибки выполнения:

```
<       Memory underflow
+[>+]   Memory overflow
```

## Булевые выражения

Рассмотрим такой код на языке C:

```
if(((a > 5) && (b < 10)) || (c == 0))
{
  do_something();
}
```

В принципе, булевое выражение можно вычислять как обычное арифметическое, то есть, по такой схеме:

- `a > 5`
- `b < 10`
- `(a > 5) && (b < 10)`
- `b == 0`
- `((a > 5) && (b < 10)) || (c == 0)`

Такой способ вычисления называется полным. Можем ли мы вычислить значение этого выражения быстрее? Смотрите, если `c == 0`, то всё выражение будет иметь значение true в любом случае, независимо от `a` и `b`. А вот если `c != 0`, то приходится поверять значения `a` и `b`. Таким образом, наш код (фактически) превращается в такой:

```
if(c == 0)
{
  goto do_it;
}
if((a > 5) && (b < 10))
{
  goto do_it;
}
goto dont_do_it;

do_it:
  do_something();
dont_do_it:
```

В принципе, можно пойти дальше: если `a <= 5`, нас не интересует сравнение `b < 10`: всё равно выражение равно false.

```
if(c == 0)
{
  goto do_it;
}
if(a > 5)
{
  if(b < 10)
  {
    goto do_it;
  }
}
goto dont_do_it;

do_it:
  do_something();

dont_do_it:
```

Такой способ вычисления выражений называется сокращённым или "ленивым" (от англ. lazy evaluation), потому что позволяет вычислить выражение, не проверяя всех входящих в него подвыражений. Можно вывести такие формальные правила:

• Если у оператора OR хотя бы один операнд имеет значение true, всё выражение имеет значение true
• Если у оператора AND хотя бы один операнд имеет значение false, всё выражение иммет значение false

В принципе, ленивое вычисление булевых выражений помогает написать более быстрый (а часто и более простой) код. С другой стороны, возникают проблемы, если одно из подвыражений при вычислении вызывает стороние эффекты (англ. side effetcs), например вызов функции:

```
if(foo() || (c == 0))
{
  do_something();
}
```

Если мы используем ленивое вычисление и оказывается, что `c == 0`, то функция `foo()` вызвана не будет, потому что от её разельтата значение выражения уже не зависит. Хорошо это или плохо зависит от конкретной ситуации, но без сомнения, способ выполнения такого кода становится неочевидным.

Теперь переёдем к тому, как это реализовывается на ассемблере. Начнём с полного вычисления:

```
cmpl  $5, a
/* так, а что дальше? */
```

Действительно, нам нужно сохранить результат сравнения в переменную. Из команд, анализирующих флаги мы знаем только семейство `j _cc_`, но они нам не подходят. Кроме `j _cc_` существует семейство `set _cc_`. Они проверяют состояние флагов точно так же, как и `j _cc_`. На основе флагов операнд устанавливается в 1, если проверяемое условие `cc` истинно, и в 1 если усливие ложно.

```
setcc операнд
```

Требуется заметить, что команды `set _cc_` работают только с операндами (регистры и память) размером один байт.

Тогда полное вычисление будет выглядеть так:

```
cmpl  $5, a
seta  %al
cmpl  $10, b
```

```
        setb  %bl
        andb  %bl, %al
        cmpl  $0, c
        sete  %bl
        orb   %bl, %al
        jz    is_false
is_true:
        ...
is_false:
        ...
```

Обратите внимание, что команда `or` устанавливает флаги и нам не нужно отдельно сравнивать `%al` с нулём.

---

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual, 4.1 Instructions (N-Z), PUSH

[2] Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.3 Using LEA

[3] Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.7 Compares

[4] Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.6 Clearing Registers and Dependency Breaking Idioms

# Глава 5. Отладчик GDB

Целью отладки программы является удаление багов из программы. Для этого вам скорее всего понадобится исследовать состояние переменных во время выполнения, а также и сам процесс выполнения (какие условия сработали и т. д.). Тут отладчик — наш первый помощник. Конечно же, в C достаточно много возможностей отладки без непосредственной остановки программы: от простого `printf(3)` до специальных систем ведения логов по сети и syslog. В ассемблере такие методы тоже применимы, но вы можете захотеть посмотреть состояния регисторв, дамп оперативной памяти и другие вещи, которые гораздо удобнее сделать в интерактивном отладчике. В общем, если вы пишете на ассемблере, то без отладчика вы вряд ли обойдётесь.

Вы можете начать отладку с определиня точки останова (breakpoint), если вы уже приблизительно знаете, какой участок кода хотите исследовать. Этот способ используется больше всего: ставим точку останова, запускам программу и проходим её выполнение по шагам, попутно исследуя необходимые переменные и регистры. Вы также можете просто запустить прогорамму под отладчиком и поймать момент, когда она вылетает с Segmentation fault — так можно узнать, какая инструкция пытается получить доступ к памяти, рассмотреть по-подробнее, что это за переменная и т. д. Теперь можно исследовать этот код ещё раз, пройти его по шагам, поставив точку останова чуть раньше момента сбоя.

Начнём с простого. Возьмём программу hello world и скомпилируем её с отладочной информацией при помощи ключа `-g`:

```
[user@host:~]$ gcc -g hello.s -o hello
[user@host:~]$
```

Запускаем gdb:

```
[user@host:~]$ gdb ./hello
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
```

```
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/lib/tls/
    libthread_db.so.1".

(gdb)
```

GDB запустился, загрузил вашу программу, вывел на экран приглашение (gdb) и ждёт команд. Мы хотим пройти программу "по шагам" (single-step mode). Для этого мы должны указать команду, на которой программа должна остановиться. Можно указать подпрограмму — тогда остановка будет перед началом исполнения инструкций этой подпрограммы. Ещё можно указать имя файла и номер строки.

```
(gdb) b main
Breakpoint 1 at 0x8048324: file hello.s, line 17.
(gdb)
```

b — сокращение от break. Все команды в GDB можно сокращать, если это не создаёт двусмысленных расшифровок. Запускам программу командой run. Эта же команда используется для перезапуска сначала уже запущенной программы.

```
(gdb) r
Starting program: /tmp/hello

Breakpoint 1, main () at hello.s:17
17              movl    $4, %eax                /* помещаем номер системного вызова write = 4
Current language:  auto; currently asm
(gdb)
```

GDB остановил программу и ждёт команд. Вы видите команду вашей программы, которая будет выполнена следующей, имя функции, которая сейчас исполняется, имя файла и номер строки. Для пошагового исполнения у нас есть две команды: step (сокращённо s) и next (сокращённо n). Команда step производит выполнение программы с заходом в подпрограммы. Конмада next выполняет пошагово только инструкции текущей подпрограммы.

```
(gdb) n
20              movl    $1, %ebx                /* первый параметр -- в регистр %ebx
(gdb)
```

Итак, инструкция на строке 17 выполнена, мы ожидаем, что в регистре %eax находится число 4. Для вывода на экран различных выражений используется команда print (сокращённо p). В отличие от команд ассемблера, GDB в записи регистров использует знак $ вместо %. Посмотрим, что в регистре %eax:

```
(gdb) p $eax
$1 = 4
(gdb)
```

Действительно 4! :) GDB нумерует все выведенные выражения. Сейчас мы видим первое выражение ($1), которое равно 4. Теперь к этому выражению можно обращаться по имени. Также можно производить простые вычисления:

```
(gdb) p $1
$2 = 4
(gdb) p $1 + 10
$3 = 14
(gdb) p 0x10 + 0x1f
$4 = 47
(gdb)
```

Пока мы игрались с командой print, мы уже забыли, какая инструкция исполняется следующей. Команда info line выводит информацию об указанной строке кода. Без аргументов выводит информацию о текущей строке кода.

```
(gdb) info line
Line 20 of "hello.s" starts at address 0x8048329 <main+5> and ends at 0x804832e <main+10>.
(gdb)
```

Команда list (сокращённо l) выводит на экран исходный код вашей программы. В качестве аргументов ей можно передать:

• номер_строки — номер строки в текущем файле
• файл :номер_строки — номер строки в указаном файле
• имя_функции — имя функции, если нет неоднозначности
• файл :имя_функции — имя функции в указаном файле
• *адрес — адрес в памяти, по которому расположена необходимая инструкция

Если передавать один аргумент, то команда list выведет 10 строк исходного кода вокруг этого места. Передавая два аргумента вы указываете строку начала и строку конца листинга.

```
(gdb) l main
12                                                      за пределами этого файла             */
13      .type   main, @function                 /* main -- функция (а не данные)             */
14
15
16      main:
17              movl    $4, %eax                /* помещаем номер системного вызова write = 4
18                                                 в регистр %eax                               */
19
20              movl    $1, %ebx                /* первый параметр -- в регистр %ebx
21                                                 номер файлового дескриптора stdout = 1       */
(gdb) l *$eip
0x8048329 is at hello.s:20.
15
16      main:
17              movl    $4, %eax                /* помещаем номер системного вызова write = 4
18                                                 в регистр %eax                               */
19
20              movl    $1, %ebx                /* первый параметр -- в регистр %ebx
21                                                 номер файлового дескриптора stdout = 1       */
22
23              movl    $hello_str, %ecx        /* второй параметр -- в регистр %ecx
24                                                 указатель на строку                          */
(gdb) l 20, 25
20              movl    $1, %ebx                /* первый параметр -- в регистр %ebx
21                                                 номер файлового дескриптора stdout = 1       */
22
23              movl    $hello_str, %ecx        /* второй параметр -- в регистр %ecx
24                                                 указатель на строку                          */
25
(gdb)
```

Запомните эту команду: list *$eip. С её помощью вы всегда можете просмотреть исходный код вокруг инструкции, которая сейчас выполняется. Выполняем нашу программу дальше:

```
(gdb) n
23              movl    $hello_str, %ecx        /* второй параметр -- в регистр %ecx
(gdb) n
26              movl    $hello_str_length, %edx /* третий параметр -- в регистр %edx
(gdb)
```

Не правда ли, утомительно каждый раз нажимать n? Если просто нажать Enter, GDB повторит последнюю команду:

```
(gdb)
29              int     $0x80                   /* вызываем прерывание 0x80                     */
(gdb)
Hello, world!
31              movl    $1, %eax                /* номер системного вызова exit = 1             */
(gdb)
```

Ещё одна удобная команда, о которой стоит знать — это info registers. Конечно же, её можно сократить до i r. Ей можно передать параметр — список регистров, которые необходимо напечатать. Например, находясь в защищённом режиме, нам вряд ли будут интересны значения сегментных регистров.

```
(gdb) info registers
eax            0xe      14
ecx            0x804955c        134518108
edx            0xe      14
ebx            0x1      1
esp            0xbfabb55c       0xbfabb55c
ebp            0xbfabb5a8       0xbfabb5a8
esi            0x0      0
edi            0xb7f6bcc0       -1208566592
eip            0x804833a        0x804833a <main+22>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) info registers eax ecx edx ebx esp ebp esi edi eip eflags
eax            0xe      14
ecx            0x804955c        134518108
edx            0xe      14
ebx            0x1      1
esp            0xbfabb55c       0xbfabb55c
ebp            0xbfabb5a8       0xbfabb5a8
esi            0x0      0
edi            0xb7f6bcc0       -1208566592
eip            0x804833a        0x804833a <main+22>
eflags         0x246    [ PF ZF IF ]
(gdb)
```

Так, а кроме регистров у нас ещё ведь есть память, и частный случай памяти — стек. Как их посмотреть? Команда x/формат адрес отображает содержимое памяти, расположенной по адресу в заданном формате. Формат это (в таком порядке) количество элементов, буква формата, и размер элемента. Буквы формата: o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) and s(string). Размер: b(byte), h(halfword), w(word), g(giant, 8 bytes). Например, напечатаем 14 символов строки hello_str:

```
(gdb) x/14c &hello_str
0x804955c <hello_str>:  72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 44 ','  32 ' '  119 'w'
0x8049564 <hello_str+8>:        111 'o' 114 'r' 108 'l' 100 'd' 33 '!'  10 '\n'
(gdb)
```

То же самое, только в шестнадцатеричном виде:

```
(gdb) x/14xb &hello_str
0x804955c <hello_str>:  0x48    0x65    0x6c    0x6c    0x6f    0x2c    0x20    0x77
0x8049564 <hello_str+8>:        0x6f    0x72    0x6c    0x64    0x21    0x0a
(gdb)
```

Напечатаем 8 верхних слов (4 байта) из стека (для "погружения в стек" читаем слева направо и сверху вниз):

```
(gdb) x/8xw $esp
0xbfd8902c:     0xb7e14ea8      0x00000001      0xbfd890a4      0xbfd890ac
0xbfd8903c:     0x00000000      0xb7f2dff4      0x00000000      0xb7f53cc0
(gdb)
```

Было бы хорошо, если бы GDB отображал значение какого-то выражения автоматически. Это делает команда display/формат выражение. Если в формате будет указан размер, то принцип действия аналогичен x. Если размер не указан, то команда ведёт себя как print.

```
(gdb) display/4xw $esp
1: x/4xw $esp
0xbf8fdb9c:     0xb7e4dea8      0x00000001      0xbf8fdc14      0xbf8fdc1c
(gdb) display/x $eax
2: /x $eax = 0xe
(gdb) n
32              movl    $0, %ebx                /* передаём 0 как значение параметра           */
```

```
2: /x $eax = 0x1
1: x/4xw $esp
0xbf8fdb9c:     0xb7e4dea8      0x00000001      0xbf8fdc14      0xbf8fdc1c
(gdb)

frame

si, ni

display/i $eip
display/12i $eip-7

(gdb) define ir
Type commands for definition of "ir".
End with a line saying just "end".
>info registers
>end

(gdb) define test
Redefine command "test"? (y or n) y
Type commands for definition of "test".
End with a line saying just "end".
>x/x $esp
>x/x $esp+4
>x/x $esp+8
>end
(gdb) test
0xbfeba95c:     0xb7e8aea8
0xbfeba960:     0x00000001
0xbfeba964:     0xbfeba9d4
```

# Глава 6. Ссылки

Книги и спецификации:

- http://www.intel.com/products/processor/manuals/
- http://download.savannah.gnu.org/releases/pgubook/
- http://www.drpaulcarter.com/pcasm/

Программы:

- http://ald.sourceforge.net/

Tutorials and FAQs:

- http://la.kmv.ru/intro/Assembly-Intro.html
- http://web.cecs.pdx.edu/~bjorn/CS200/linux_tutorial/
- http://docs.cs.up.ac.za/programming/asm/derick_tut/
- http://www.unknownroad.com/rtfm/gdbtut/
- http://asm.sourceforge.net/resources.html
- http://en.wikibooks.org/wiki/X86_Assembly
- http://en.wikipedia.org/wiki/X86

А также:

- info gas
- info gdb

Электропочта автора - gribozavr□gmail.com