

# ОБРАЗ МЫШЛЕНИЯ – ДИЗАССЕМБЛЕР IDA Pro

## ТОМ I ОПИСАНИЕ ФУНКЦИЙ ВСТРОЕННОГО ЯЗЫКА IDA Pro

### Аннотация

Подробный справочник по функциям встроенного языка, интерфейсу и архитектуре дизассемблера IDA Pro 4.01 с уточнением особенностей младших версий.

Показывает приемы эффективного использования IDA Pro для исследования зашифрованного кода, π-кода, самомодифицирующегося кода и кода, защищенного антиотладочными приемами.

Ориентирован на системных программистов средней и высокой квалификации в совершенстве владеющих языком ассемблера микропроцессоров серии Intel 80x86 и работающих с операционными системами фирмы Microsoft.

### Введение. Об этой книге

Цель этой книги – частично компенсировать информационный голод, окутывающий один из популярнейших дизассемблеров современности – IDA Pro. Сведения, содержащиеся в документации, подаваемой вместе с этим продуктом, весьма отрывочные и безнадежно устаревшие.

Самостоятельное же освоение IDA Pro требует значительных усилий, длительного времени и постоянных консультаций с ее разработчиками. Появлению этой книги предшествовал большой объем работы, проделанной автором. Первоначально когда замышлялось написать книгу, обобщающую достижения современного реинженеринга, планировалось скоромное издание страниц максимум в пятьсот, но в течение работы над проектом выяснилось: описание одних лишь функций встроенного языка IDA Pro значительно превышает этот объем. Поэтому весь материал пришлось разбить на три тома – «Описание функций встроенного языка IDA Pro», «Приемы эффективной работы с IDA Pro» и «Технологии дизассемблирования».

Функции ядра IDA находят применение не только во встроенном языке, – сама IDA активно использует их для дизассемблирования, а большинство пунктов меню эквивалентны соответствующим функциям ядра. Поэтому, любую операцию можно выполнить не только последовательными нажатиями «горячих» клавиш, но и совокупностью команд встроенного языка.

Эта книга так же рассказывает и об архитектуре, затрагивая вопросы внутреннего устройства IDA Pro без понимания которых полноценная работа с дизассемблера невозможна.

**ОБРАЩЕНИЕ АВТОРА К ЧИТАТЕЛЮ:** когда писались первые строки этой книги ее автор еще не обладал тем опытом, который необходим для написания справочной литературы подобного типа. В результате, из-под пера вылезло нечто ужасное, и все пришлось переписывать заново.

К сожалению, сроки издания нельзя бесконечно оттягивать (читатели нервничают, издатель сердится) и в том издании, что вы держите сейчас в руках, «доведены до ума» лишь десять первых глав из двадцати, а остальные даны в первоизданном варианте.

Автор просит читателя извинить его за такую ситуацию и, положив руку на сердце, обещает, что в следующем издании (если только одно будет это следующее издание – это ж от читателей зависит) все огрехи будут исправлены.

Крис Касперски.  
февраль 2001  
Серверный Кавказ.

## Версии IDA Pro

Дизассемблер IDA Pro относится к интенсивно развивающимся продуктам, – постоянные совершенствования и бесконечные изменения, вносимые разработчиками, породили множество версий, из которых наибольшее распространение получили 3.84, 3.84b, 3.85, 4.0, а некоторые до сих пор предпочитают использовать IDA 3.6.

К сожалению, даже близкие версии плохо совместимы между собой – прототипы и поведение функций встроенного языка находится в постоянном изменении, затрудняя создание переносимых скриптов.

Приводимый в книге материал в основном рассчитан на IDA Pro 4.0.1, но в ряде случаев оговариваются особенности поведения и других версий дизассемблера.

Существуют три различных пакета поставки дизассемблера – **стандартная** (*IDA Pro Standard*), **прогрессивная** (*IDA Pro Advanced*) и **демонстрационная** (*IDA Pro Demo*). Отличие между IDA Pro Standard и IDA Pro Advanced заключается в количестве поддерживаемых процессоров, полный перечень которых можно найти в прилагаемой к IDA документации. Демонстрационный пакет представляет собой усеченный вариант полнофункционального продукта и обладает рядом существенных ограничений: никакие процессоры кроме семейства Intel 80x86 и типы файлов за исключением win32 PE не поддерживаются; в поставку входят сигнатуры всего лишь двух компиляторов – Microsoft Visual C++ 6.0 и Borland C++ Builder; функция сохранения результатов работы заблокирована, а максимальное время продолжительности одного сеанса работы с дизассемблером ограничено.

В каждый пакет поставки (за исключением демонстрационного) входят две ипостаси – одна **графическая** под Windows-32 (в дальнейшем обозначаемая как IDAG) и три **консольных** для MS-DOS, OS/2 и Windows-32. В демонстрационный пакет входит лишь одна графическая ипостась. Обе ипостаси обладают сходными функциональными возможностями, поэтому в книге будет описана лишь одна из них, скомпилированная для исполнения в среде Windows-32 (в дальнейшем обозначаемая как IDAW)

Покупка IDA Pro Standard или IDA Pro Advanced дает право на бесплатное приобретение IDA SDK (*Software Development Kit*) – программного пакета, позволяющего пользователю самостоятельно разрабатывать процессорные модули, файловые загрузчики и плагины (внешние самостоятельные модули). Это неограниченно расширяет возможности IDA Pro, позволяя анализировать любой код, для какого бы микропроцессора и операционной системы он ни предназначался. SDK различных версий не совместимы друг с другом и созданные пользователем модули могут не работать в другой версии IDA Pro, поэтому, прежде чем переходить на очередную версию IDA Pro, настоятельно рекомендуется убедиться в сохранении работоспособности всех скриптов, модулей и плагинов и т.д.

**Рисунок 1 "ida.console.view"** Так выглядит консольная ипостась IDA Pro 4.01

**Рисунок 2 "ida.gui.view"** Так выглядит графическая ипостась IDA Pro 4.01

**Рисунок 3 "ida.gui.view.4.14.bmp"** Так выглядит графическая ипостась IDA Pro 4.14 Demo

## Кратное введение в дизассемблирование

Одним из способов изучения программ в отсутствии исходных текстов является **дизассемблирование**, – перевод двоичных кодов процессора в удобочитаемые мнемонические инструкции. С первого взгляда кажется: ничего сложного в такой операции нет, и один дизассемблер не будет сильно хуже любого другого. На самом же деле, ассемблирование – однонаправленный процесс с потерями, поэтому автоматическое восстановление исходного текста **невозможно**.

Одна из фундаментальных проблем дизассемблирования заключается в синтаксической неотличимости констант от адресов памяти (сегментов и смещений). Потребность распознавания смещений объясняется необходимостью замены конкретных адресов на метки, действительное смещение которых определяется на этапе ассемблирования программы.

Сказанное можно проиллюстрировать следующим примером: рассмотрим исходную программу (а). При ассемблировании смещение строки s0, загружаемое в регистр s0 заменяется его конкретным значением, в данном случае равным 108h, отчего, команда “MOV DX, offset s0” приобретает вид “MOV DX, 108h”. Это влечет за собой потерю информации – теперь уже нельзя однозначно утверждать как выглядел исходный текст, т.к. ассемблирование “...offset s0” и “...108h” дает одинаковый результат, т.е. функция ассемблирования **не инъективна**<sup>1</sup>.

Если все машинные инструкции исходного файла, перевести в соответствующие им символьные мнемоники (назовем такую операцию **простым синтаксическим дизассемблированием**), в результате получится (б). Легко видеть – программа сохраняет работоспособность лишь до тех пор, пока выводимая строка располагается по адресу 108h. Если модификация кода программы (с) нарушает такое равновесие, на экране вместо ожидаемого приветствия появляется мусор – теперь выводимая строка находится по адресу 0x10C, но в регистр DX по прежнему загружается прежнее значение ее смещения – 0x108 (d).

<pre>mov    ah,9 mov    dx, offset s0 int    21h ret s0     DB 'Hello,World!',0Dh,0Ah,'\$'</pre>	→	<pre>mov    ah,9 mov    dx,0108h int    21h ret s0 DB 'Hello,World!',0Dh,0Ah,'\$'</pre>	→	<pre>mov    ah,09 mov    dx,0108h int    21h xor    ax,ax int    16h ret s0 DB 'Hello,World!',0Dh,0Ah,'\$'</pre>
(a) Исходная программа		(b) Дизассемблированная программа		(c) Модифицированная программа

<pre>:0100 start :0100 :0102 :0105 :0107 :0109 :010B :010B aHelloWorld :010C end</pre>	<pre>proc    near mov     ah, 9 mov     dx, 108h int     21h xor     ax, ax int     16h retn db     'Hello,World!',0Dh,0Ah,'\$' start</pre>
(d) Неработоспособный результат	

Аналогичная проблема возникает и переводе с одного языка на другой – фраза «это ключ» в зависимости от ситуации может быть переведена и как “this is key”, и “this is clue”, и “this is switch”... Для правильного перевода мало простого словаря - подстрочечника, необходимо еще понимать о чем идет речь, т.е. осмысливать переводимый текст.

Человек легко может определить, что содержимое регистра DX в данном случае

<sup>1</sup> Функция  $f(x) = y$  называется инъективной, если уравнение  $f(y) = x$ , имеет только один корень и, соответственно, наоборот.

является именно смещением, а не чем ни будь иным, поскольку, его ожидает функция 0x9 прерывания 0x21. Но дизассемблеру для успешной работы мало знать одних прототипов системных и библиотечных функций, – он должен еще уметь отслеживать содержимое регистров, а, следовательно, «понимать» команды микропроцессора. Создание такого дизассемблера (часто называемого **контекстным**) очень сложная инженерная задача, тесно граничащая с искусственным интеллектом, которая на сегодняшний день еще никем не решена. Существует более или менее удачные разработки, но ни одна из них не способна генерировать 100%-работоспособные листинги.

Например, путь в исходной программе имелся фрагмент (а), загружающий в регистр AX смещение начала таблицы, а в регистр BX индекс требуемого элемента. Ассемблер, заменяя оба значения константами (b), создает неразрешимую задачу, – легко видеть, что один из регистров содержит смещение, а другой индекс, но как узнать какой именно?

MOV AX,offset Table	BB 00 02		MOV AX,0010
MOV BX,200h ; Index	01 D8		MOV BX,0200
ADD AX,BX	→ 8B 07	→	ADD AX,BX
MOV AX,[BX]	B8 10 00		MOV AX,Word ptr [BX]

(a) Исходная программа (b) Машинный код (c) Дизассемблированный текст

Другая фундаментальная проблема заключается в невозможности определения границ инструкций синтаксическим дизассемблером. Путь в исходной программе (а) имелась директива выравнивания кода по адресам кратным четырем, тогда ассемблер (b) вставит в этом месте несколько произвольных символов (как правило нулей), а дизассемблер «не зная» об этом, примет их за часть инструкций, в результате чего сгенерирует ни на что ни годный листинг (с).

JMP Label	00: E90100
<b>Align 4</b>	<b>03: 00</b>
Label: XOR AX,AX	04: 33 C0
RET	06: C3

(a) Исходная программа (b) Машинный код

00: E9 01 00	jmp 04
03: 00 33	add [bp][di],dh
05: C0 C3	rol bl,-070;

(c) Дизассемблированный текст

Контекстные дизассемблеры частично позволяют этого избежать, поскольку, способны распознавать типовые способы передачи управления, но если программист использует регистровые переходы, дизассемблеру придется эмулировать выполнение программы, для определения значений регистров в каждой точке программы. Это не только технически сложная, но и ресурсоемкая задача, решение которой еще предстоит найти.

Дизассемблирование – творческий процесс, развивающий интуицию и абстрактное мышление, возможно, даже особый вид искусства, позволяющего каждому проявить свою индивидуальность. На сегодняшний день не существует ни одного полностью автоматического дизассемблера, способного генерировать безупречно работоспособный листинг и доводить полученный ими результат до готовности приходится человеку. Таким образом, встает вопрос о механизмах взаимодействия человека с дизассемблером.

По типу реализации интерфейса взаимодействия с пользователем, существующие дизассемблеры можно разделить на две категории – **автономные** и **интерактивные**. Автономные дизассемблеры требуют от пользования задания всех указаний до начала дизассемблирования и не позволяют вмешиваться непосредственно в сам процесс. Если же конечный результат окажется неудовлетворительным, пользователь либо вручную правит полученный листинг, либо указывает дизассемблеру на его ошибки и повторяет всю

процедуру вновь и вновь, порой десятки раз! Такой способ общения человека с дизассемблером непродуцируемо и неудобно, но его легче запрограммировать.

Интерактивные дизассемблеры обладают развитым пользовательским интерфейсом, благодаря которому приобретают значительную гибкость, позволяя человеку «вручную» управлять разбором программы, помогая автоматическому анализатору там, где ему самому не справиться – отличать адреса от констант, определять границы инструкций и т.д.

Примером автономного дизассемблера является SOURCER, а интерактивного – IDA. Преимущество SOURCER-а заключается в простоте управления, в то время как работа с IDA требует высокой квалификации и навыков системного программирования. Неопытные пользователи часто предпочитают SOURCER, лидирующий среди других дизассемблеров, на небольших проектах. Но он очень плохо справляется с анализом большого, порядка нескольких мегабайт, заковыристого файла, а с шифрованным или π-кодом не справляется вообще! И тогда на помощь приходит IDA, которая, будучи виртуальной программируемой машиной, может абсолютно **все** – стоит лишь разработать и ввести соответствующий скрипт. А как это сделать и рассказывает настоящая книга.

## Первые шаги с IDA Pro

С легкой руки Дениса Ричи повелось начинать освоение нового языка программирования с создания простейшей программы “Hello, World!”, -- и здесь не будет нарушена эта традиция. Оценим возможности IDA Pro следующим примером (для совместимости с книгой рекомендуется откомпилировать его с помощью Microsoft Visual C++ 6.0 вызовом “cl.exe first.cpp” в командной строке):

```
#include <iostream.h>
void main()
{
    cout<<"Hello, Sailor!\n";
}
a) исходный текст программы first.cpp
```

Компилятор сгенерирует исполняемый файл размером почти в 40 килобайт, большую часть которого займет служебный, стартовый или библиотечный код! Попытка дизассемблирования с помощью таких дизассемблеров как W32DASM (или аналогичных ему) не увенчается быстрым успехом, поскольку над полученным листингом размером **в пятьсот килобайт** (!) можно просидеть не час и не два. Легко представить сколько времени уйдет на серьезные задачи, требующие изучения десятков мегабайт дизассемблированного текста.

Попробуем эту программу дизассемблировать с помощью IDA. Если все настройки оставить по умолчанию, после завершения анализа экран (в зависимости от версии) должен выглядеть следующим образом:

**Рисунок 4 “0x000.bmp” Так выглядит результат работы консольной версии IDA Pro 3.6**

**Рисунок 5 “0x001.bmp” Так выглядит результат работы консольной версии IDA Pro 4.0**

**Рисунок 6 “0x002.bmp” Так выглядит результат работы графической версии IDA Pro 4.0**

С версии 3.8<sup>2</sup> в IDA появилась поддержка «сворачивания» (*Collapsed*) функций. Такой прием значительно упрощает навигацию по тексту, позволяя убрать с экрана не интересные в данный момент строки. По умолчанию все библиотечные функции сворачиваются автоматически.

Развернуть функцию можно подведя к ней курсор и нажав <+> на дополнительной цифровой клавиатуре, расположенной справа. Соответственно, клавиша <-> предназначена для сворачивания.

По окончании автоматического анализа файла "first.exe", IDA переместит курсор к строке ".text:00401B2C" – точке входа в программу. Среди начинающих программистов широко распространено заблуждение, якобы программы, написанные на Си, начинают выполняться с функции "main", но в действительности это не совсем так. На самом деле сразу после загрузки файла управление передается на функцию "Start", вставленную компилятором. Она подготавливает глобальные переменные \_osver (билд), \_winmajor (старшая версия операционной системы), \_winminor (младшая версия операционной системы), \_winver (полная версия операционной системы), \_\_argc (количество аргументов командной строки), \_\_argv (массив указателей на строки аргументов), \_environ (массив указателей на строки переменных окружения); инициализирует кучи (heap); вызывает функцию main, а после возвращения управления завершает процесс с помощью функции Exit.

Наглядно продемонстрировать инициализацию переменных, совершаемую стартовым кодом, позволяет следующая программа.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int a;
    printf(">Версия OS:\t\t\t%d.%d\n\
>Билд:\t\t\t%d\n\
>Количество аргументов:\t%d\n", \
    _winmajor, _winminor, _osver, __argc);
    for (a=0; a<__argc; a++)
        printf(">\tАргумент\t\t\t%02d:\t\t%s\n", a+1, __argv[a]);
    a=!a-1;
    while(_environ[++a]) ;
    printf(">Количество переменных окружения:%d\n", a);
    while(a) printf(">\tПеременная\t%d:\t\t%s\n", a, _environ[--a]);
}
```

a) исходный текст программы CRt0.demo.c

Прототип функции main как будто указывает, что приложение не принимает ни каких аргументов командной строки, но результат работы программы доказывает обратное и на машине автора выглядит так (приводится в сокращенном виде):

```
>Версия OS: 5.0
>Билд: 2195
>Количество аргументов: 1
> Аргумент 01: CRt0.demo
>Количество переменных окружения: 30
> Переменная 29: windir=C:\WINNT
>...
```

b) результат работы программы CRt0.demo.c

Очевидно, нет никакой необходимости анализировать стандартный стартовый код

---

<sup>2</sup> А может и чуточку раньше

приложения, и первая задача исследователя – найти место передачи управления на функцию main. К сожалению, гарантированное решение этой задачи требует полного анализа содержимого функции “Start”. У исследователей существует множество хитростей, но все они базируются на особенностях реализации конкретных компиляторов<sup>3</sup> и не могут считаться универсальными.

Рекомендуется изучить исходные тексты стартовых функций популярных компиляторов, находящиеся в файлах CRt0.c (Microsoft Visual C) и c0w.asm (Borland C) – это упростит анализ дизассемблерного листинга.

Ниже, в качестве иллюстрации, приводится содержимое стартового кода программы “first.exe”, полученное в результате работы W32Dasm:

```
//***** Program Entry Point *****  
:00401B2C 55          push ebp  
:00401B2D 8BEC        mov ebp, esp  
:00401B2F 6AFF        push FFFFFFFF  
:00401B31 6870714000  push 00407170  
:00401B36 68A8374000  push 004037A8  
:00401B3B 64A100000000 mov eax, dword ptr fs:[00000000]  
:00401B41 50          push eax  
:00401B42 64892500000000 mov dword ptr fs:[00000000], esp  
:00401B49 83EC10      sub esp, 00000010  
:00401B4C 53          push ebx  
:00401B4D 56          push esi  
:00401B4E 57          push edi  
:00401B4F 8965E8      mov dword ptr [ebp-18], esp
```

Reference To: KERNEL32.GetVersion, Ord:0174h

```
|  
:00401B52 FF1504704000 Call dword ptr [00407004]  
:00401B58 33D2       xor edx, edx  
:00401B5A 8AD4       mov dl, ah  
:00401B5C 8915B0874000 mov dword ptr [004087B0], edx  
:00401B62 8BC8       mov ecx, eax  
:00401B64 81E1FF000000 and ecx, 000000FF  
:00401B6A 890DAC874000 mov dword ptr [004087AC], ecx  
:00401B70 C1E108     shl ecx, 08  
:00401B73 03CA       add ecx, edx  
:00401B75 890DA8874000 mov dword ptr [004087A8], ecx  
:00401B7B C1E810     shr eax, 10  
:00401B7E A3A4874000 mov dword ptr [004087A4], eax  
:00401B83 6A00       push 00000000  
:00401B85 E8D91B0000 call 00403763  
:00401B8A 59         pop ecx  
:00401B8B 85C0       test eax, eax  
:00401B8D 7508       jne 00401B97  
:00401B8F 6A1C       push 0000001C  
:00401B91 E89A000000 call 00401C30  
:00401B96 59         pop ecx
```

Referenced by a (U)nconditional or (C)onditional Jump at Address:

---

<sup>3</sup> Например, Microsoft Visual C всегда, независимо от прототипа функции main передает ей три аргумента – указатель на массив указателей переменных окружения, указатель на массив указателей аргументов командной строки и количество аргументов командной строки, а все остальные функции стартового кода принимают меньшее количество аргументов

```

|:00401B8D(C)
|
:00401B97 8365FC00          and dword ptr [ebp-04], 00000000
:00401B9B E8D70C0000        call 00402877

```

Reference To: KERNEL32.GetCommandLineA, Ord:00CAh

```

|
:00401BA0 FF1560704000        Call dword ptr [00407060]
:00401BA6 A3E49C4000        mov dword ptr [00409CE4], eax
:00401BAB E8811A0000        call 00403631
:00401BB0 A388874000        mov dword ptr [00408788], eax
:00401BB5 E82A180000        call 004033E4
:00401BBA E86C170000        call 0040332B
:00401BBF E8E1140000        call 004030A5
:00401BC4 A1C0874000        mov eax, dword ptr [004087C0]
:00401BC9 A3C4874000        mov dword ptr [004087C4], eax
:00401BCE 50                push eax
:00401BCF FF35B8874000        push dword ptr [004087B8]
:00401BD5 FF35B4874000        push dword ptr [004087B4]
:00401BDB E820F4FFFF        call 00401000
:00401BE0 83C40C          add esp, 0000000C
:00401BE3 8945E4          mov dword ptr [ebp-1C], eax
:00401BE6 50                push eax
:00401BE7 E8E6140000        call 004030D2
:00401BEC 8B45EC          mov eax, dword ptr [ebp-14]
:00401BEF 8B08          mov ecx, dword ptr [eax]
:00401BF1 8B09          mov ecx, dword ptr [ecx]
:00401BF3 894DE0          mov dword ptr [ebp-20], ecx
:00401BF6 50                push eax
:00401BF7 51                push ecx
:00401BF8 E8AA150000        call 004031A7
:00401BFD 59                pop ecx
:00401BFE 59                pop ecx
:00401BFF C3                ret

```

а) стартовый код программы "first.exe", полученный дизассемблером W32Dasm

Иначе выглядит результат работы IDA, умеющей распознавать библиотечные функции по их сигнатурам (приблизительно по такому же алгоритму работает множество антивирусов). Поэтому, способности дизассемблера тесно связаны с его версией и полной комплект поставки – далеко не все версии IDA Pro в состоянии работать с программами, сгенерированными современными компиляторами. (Перечень поддерживаемых компиляторов можно найти в файле "%IDA%/SIG/list").

```

00401B2C start      proc near
00401B2C
00401B2C var_20        = dword ptr -20h
00401B2C var_1C        = dword ptr -1Ch
00401B2C var_18        = dword ptr -18h
00401B2C var_14        = dword ptr -14h
00401B2C var_4         = dword ptr -4
00401B2C
00401B2C          push    ebp
00401B2D          mov     ebp, esp
00401B2F          push    0FFFFFFFFh
00401B31          push    offset stru_407170
00401B36          push    offset __except_handler3
00401B3B          mov     eax, large fs:0

```



```

00401B41      push    eax
00401B42      mov     large fs:0, esp
00401B49      sub     esp, 10h
00401B4C      push    ebx
00401B4D      push    esi
00401B4E      push    edi
00401B4F      mov     [ebp+var_18], esp
00401B52      call   ds:GetVersion
00401B58      xor     edx, edx
00401B5A      mov     dl, ah
00401B5C      mov     dword_4087B0, edx
00401B62      mov     ecx, eax
00401B64      and     ecx, 0FFh
00401B6A      mov     dword_4087AC, ecx
00401B70      shl     ecx, 8
00401B73      add     ecx, edx
00401B75      mov     dword_4087A8, ecx
00401B7B      shr     eax, 10h
00401B7E      mov     dword_4087A4, eax
00401B83      push    0
00401B85      call   __heap_init
00401B8A      pop     ecx
00401B8B      test   eax, eax
00401B8D      jnz    short loc_401B97
00401B8F      push    1Ch
00401B91      call   sub_401C30      ; _fast_error_exit
00401B96      pop     ecx
00401B97
00401B97 loc_401B97:      ; CODE XREF: start+61↑j
00401B97      and     [ebp+var_4], 0
00401B9B      call   __ioinit
00401BA0      call   ds:GetCommandLineA
00401BA6      mov     dword_409CE4, eax
00401BAB      call   __crtGetEnvironmentStringsA
00401BB0      mov     dword_408788, eax
00401BB5      call   __setargv
00401BBA      call   __setenvp
00401BBF      call   __cinit
00401BC4      mov     eax, dword_4087C0
00401BC9      mov     dword_4087C4, eax
00401BCE      push    eax
00401BCF      push    dword_4087B8
00401BD5      push    dword_4087B4
00401BDB      call   sub_401000
00401BE0      add     esp, 0Ch
00401BE3      mov     [ebp+var_1C], eax
00401BE6      push    eax
00401BE7      call   _exit
00401BEC ; -----
00401BEC
00401BEC loc_401BEC:      ; DATA XREF: _rdata:00407170↓o
00401BEC      mov     eax, [ebp-14h]
00401BEF      mov     ecx, [eax]
00401BF1      mov     ecx, [ecx]
00401BF3      mov     [ebp-20h], ecx
00401BF6      push    eax

```

```

00401BF7      push    ecx
00401BF8      call   __XcptFilter
00401BFD      pop     ecx
00401BFE      pop     ecx
00401BFF      retn
00401BFF start  endp ; sp = -34h
b) стартовый код программы "first.exe", полученный дизассемблером IDA Pro 4.01

```

С приведенным примером IDA Pro успешно справляется, о чем свидетельствует стока "Using FLIRT signature: VC v2.0/4.x/5.0 runtime" в окне сообщений

### Рисунок 7 "0x003" Загрузка библиотеки сигнатур

Дизассемблер сумел определить имена всех функций вызываемых стартовым кодом, за исключением одной, расположенной по адресу 0x0401BDB. Учитывая передачу трех аргументов и обращение к `_exit`, после возвращения функцией управления, можно предположить, что это `main` и есть.

Перейти по адресу 0x0401000 для изучения содержимого функции `main` можно несколькими способами – прокрутить экран с помощью стрелок управления курсором, нажать клавишу <G> и ввести требуемый адрес в появившемся окне диалога, но проще и быстрее всего воспользоваться встроенной в IDA Pro системой навигации. Если подвести курсор в границы имени, константы или выражения и нажать <Enter>, IDA автоматически перейдет на требуемый адрес.

В данном случае требуется подвести к строке "sub\_401000" (аргументу команды `call`) и нажать на <Enter>, если все сделано правильно, экран дизассемблера должен выглядеть следующим образом:

```

00401000 ; ----- S U B R O U T I N E -----
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 sub_401000 proc near          ; CODE XREF: start+AF↓p
00401000     push   ebp
00401001     mov    ebp, esp
00401003     push   offset aHelloSailor ; "Hello, Sailor!\n"
00401008     mov    ecx, offset dword_408748
0040100D     call  ??6ostream@@QAEAAV0@PBD@Z ; ostream::operator<<(char const *)
00401012     pop    ebp
00401013     retn
00401013 sub_401000 endp

```

Дизассемблер сумел распознать строковую переменную и дал ей осмысленное имя "aHelloSailor", а в комментарии, расположенном справа, для наглядности привел оригинальное содержимое "Hello, Sailor!\n". Если поместить курсор в границы имени "aHelloSailor": и нажать <Enter>, IDA автоматически перейдет к требуемой строке:

```

00408040 aHelloSailor  db 'Hello, Sailor!',0Ah,0 ; DATA XREF: sub_401000+3↑o

```

Выражение "DATA XREF: sub\_401000+3↑o" называется перекрестной ссылкой и свидетельствует о том, что в третьей строке процедуры `sub_401000`, произошло обращение к текущему адресу по его смещению ("o" от `offset`), а стрелка, направленная вверх, указывает на относительное расположение источника перекрестной ссылки.

Если в границы выражения "sub\_401000+3" подвести курсор и нажать <Enter>, IDA Pro перейдет к следующей строке:

```

00401003     push   offset aHelloSailor ; "Hello, Sailor!\n"

```

Нажатие клавиши <Esc> отменяет предыдущее перемещение, возвращая курсор в

исходную позицию. (Аналогично команде “back” в web-браузере). Смещение строки “Hello, Sailor!\n”, передается процедуре “??6ostream@@QAEAAV0@PBD@Z”, представляющей собой оператор “<<” языка C++. Странное имя объясняется ограничениями, наложенными на символы, допустимые в именах библиотечных функций. Поэтому, компиляторы автоматически преобразуют (**замангляют**) такие имена в “абракадабру”, пригодную для работы с линкером, и многие начинающие программисты даже не догадываются об этой скрытой “кухне”.

Для облегчения анализа текста, IDA Pro в комментариях отображает «правильные» имена, но существует возможность заставить ее везде показывать незамангленные имена. Для этого необходимо в меню “Options” выбрать пункт “Demangled names” и в появившемся окне диалога переместить радио кнопку на “Names”, после этого вызов оператора “<<” станет выглядеть так:

```
0040100D      call    ostream::operator<<(char const *)
```

На этом анализ приложения “first.cpp” можно считать завершенным. Для полноты картины остается переименовать функцию “sub\_401000” в main. Для этого необходимо подвести курсор к строке 0x0401000 (началу функции) и нажать клавишу <N>, в появившемся диалоге ввести “main”. Конечный результат должен выглядеть так:

```
00401000 ; ----- S U B R O U T I N E -----
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 main      proc near          ; CODE XREF: start+AF↓p
00401000         push    ebp
00401001         mov     ebp, esp
00401003         push    offset aHelloSailor ; "Hello, Sailor!\n"
00401008         mov     ecx, offset dword_408748
0040100D         call   ostream::operator<<(char const *)
00401012         pop     ebp
00401013         retn
00401013 main      endp
```

Для сравнения результат работы W32Dasm выглядит следующим образом (ниже приводится лишь содержимое функции main):

```
:00401000 55                push ebp
:00401001 8BEC             mov ebp, esp

Possible StringData Ref from Data Obj ->"Hello, Sailor!"
|
:00401003 6840804000      push 00408040
:00401008 B948874000      mov ecx, 00408748
:0040100D E8AB000000      call 004010BD
:00401012 5D                pop ebp
:00401013 C3                ret
```

Другое важное преимущество IDA – способность дизассемблировать зашифрованные программы. В демонстрационном примере ??? “/SRC/Crypt.com” использовалась статическая шифровка, часто встречающаяся в “конвертных” защитах. Этот простой прием полностью “ослепляет” большинство дизассемблеров. Например, результат обработки файла “Crypt.com” SOURCER-ом выглядит так:

```
Crypt      proc    far

7E5B:0100          start:
7E5B:0100  83 C6 06    add     si,6
7E5B:0103  FF E6      jmp     si                ;*
                                           ;*No entry point to code

7E5B:0105  B9 14BE    mov     cx,14BEh
7E5B:0108  01 AD 5691 add     ds:data_1e[di],bp    ; (7E5B:5691=0)
7E5B:010C  80 34 66    xor     byte ptr [si],66h   ; 'f'
```

```

7E5B:010F 46          inc    si
7E5B:0110 E2 FA          loop  $-4          ; Loop if cx > 0

7E5B:0112 FF E6          jmp    si          ;*
                                           ;* No entry point to code

7E5B:114 18 00          sbb   [bx+si],al
7E5B:116 D2 6F DC          shr   byte ptr [bx-24h],cl ; Shift w/zeros fill
7E5B:119 6E 67 AB 47 A5 2E db 6Eh, 67h,0ABh, 47h,0A5h, 2Eh
7E5B:11F 03 0A 0A 09 4A 35 db 03h, 0Ah, 0Ah, 09h, 4Ah, 35h
7E5B:125 07 0F 0A 09 14 47 db 07h, 0Fh, 0Ah, 09h, 14h, 47h
7E5B:12B 6B 6C 42 E8 00 00 db 6Bh, 6Ch, 42h, E8h, 00h, 00h
7E5B:131 59 5E BF 00 01 57 db 59h, 5Eh, BFh, 00h, 01h, 57h
7E5B:137 2B CE F3 A4 C3     db 2Bh, CEh, F3h, A4h, C3h

Crypt                                endp

```

SOURCER половину кода вообще не смог дизассемблировать, оставив ее в виде дампа, а другую половину дизассемблировал неправильно! Команда “JMP SI” в строке :0x103 осуществляет переход по адресу :0x106 (значение регистра SI после загрузки com файла равно 0x100, поэтому после команды “ADD SI,6” регистр SI равен 0x106). Но следующая за “JMP” команда расположена по адресу 0x105! В исходном тексте в это место вставлен байт-пустышка, сбивающий дизассемблер с толку.

```

Start:
ADD     SI,6
JMP     SI
DB      0B9h          ;
LEA     SI,_end      ; На начало зашифрованного фрагмента

```

SOURCER не обладает способностью предсказывать регистровые переходы и, встретив команду “JMP SI” продолжает дизассемблирование, молчаливо предполагая, что команды последовательно расположены вплотную друг к другу. Существует возможность создать файл определений, указывающий, что по адресу:0x105 расположен байт данных, но подобное взаимодействие с пользователем очень неудобно.

Напротив, IDA изначально проектировалась как дружественная к пользователю интерактивная среда. В отличие от SURCER-подобных дизассемблеров, IDA не делает никаких молчаливых предположений, и при возникновении затруднений обращается за помощью к человеку. Поэтому, встретив регистровый переход по неизвестному адресу, она прекращает дальнейший анализ, и результат анализа файла “Crypt.com” выглядит так:

```

seg000:0100 start          proc near
seg000:0100                add     si, 6
seg000:0103                jmp     si
seg000:0103 start          endp
seg000:0103
seg000:0103 ; -----
seg000:0105                db 0B9h ; !
seg000:0106                db 0BEh ; -
seg000:0107                db 14h ;
seg000:0108                db 1 ;
seg000:0109                db 0ADh ; i
seg000:010A                db 91h ; N
...

```

Необходимо помочь дизассемблеру, указав адрес перехода. Начинающие пользователи в этой ситуации обычно подводят курсор к соответствующей строке и нажимают клавишу <C>, заставляя IDA дизассемблировать код с текущей позиции до конца функции. Несмотря на кажущуюся очевидность, такое решение ошибочно, ибо по-прежнему остается неизвестным куда указывает условный переход в строке :0x103 и откуда код, расположенный по адресу :0x106 получает управление.

Правильное решение – добавить перекрестную ссылку, связывающую строку :0x103, со строкой :0x106. Для этого необходимо в меню “View” выбрать пункт “Cross references” и в появившемся окне диалога заполнить поля “from” и “to” значениями

seg000:0103 и seg000:0106 соответственно.

После этого экран дизассемблера должен выглядеть следующим образом (в IDA версии 4.01.300 содержится ошибка, и добавление новой перекрестной ссылки не всегда приводит к автоматическому дизассемблированию):

```
seg000:0100          public start
seg000:0100 start    proc near
seg000:0100          add     si, 6
seg000:0103          jmp     si
seg000:0103 start    endp
seg000:0103
seg000:0103 ; -----
seg000:0105          db 0B9h ; !
seg000:0106 ; -----
seg000:0106
seg000:0106 loc_0_106:          ; CODE XREF: start+3↑u
seg000:0106          mov     si, 114h
seg000:0109          lodsw
seg000:010A          xchg  ax, cx
seg000:010B          push  si
seg000:010C
seg000:010C loc_0_10C:          ; CODE XREF: seg000:0110↓j
seg000:010C          xor   byte ptr [si], 66h
seg000:010F          inc   si
seg000:0110          loop  loc_0_10C
seg000:0112          jmp   si
seg000:0112 ; -----
seg000:0114          db 18h ;
seg000:0115          db 0 ;
seg000:0116          db 0D2h ; T
seg000:0117          db 6Fh ; o
...
```

Поскольку IDA Pro не отображает адреса-приемника перекрестной ссылки, то рекомендуется выполнить это самостоятельно. Такой прием улучшит наглядность текста и упростит навигацию. Если повести курсор к строке :0x103 нажать клавишу <:>, введя в появившемся диалоговом окне любой осмысленный комментарий (например “переход по адресу 0106”), то экран примет следующий вид:

```
seg000:0103          jmp     si          ; Переход по адресу 0106
```

Ценность такого приема заключается в возможности быстрого перехода по адресу, на который ссылается “JMP SI”, - достаточно лишь подвести курсор к числу “0106” и нажать <Enter>. Важно соблюдать правильность написания – IDA Pro не распознает шестнадцатеричный формат ни в стиле Си (0x106), ни в стиле MASM\TASM (0106h).

Что представляет собой число “114h” в строке :0x106 – константу или смещение? Чтобы узнать это, необходимо проанализировать следующую команду – “LODSW”, поскольку ее выполнение приводит к загрузке в регистр AX слова, расположенного по адресу DS:SI, очевидно, в регистр SI заносится смещение.

```
seg000:0106          mov     si, 114h
seg000:0109          lodsw
```

Однократное нажатие клавиши <O> преобразует константу в смещение и дизассемблируемый текст станет выглядеть так:

```
seg000:0106          mov     si, offset unk_0_114
seg000:0109          lodsw
...
seg000:0114 unk_0_114          db 18h ;          ; DATA XREF: seg000:0106↑o
seg000:0115          db 0 ;
seg000:0116          db 0D2h ; T
seg000:0117          db 6Fh ; o
...
```

IDA Pro автоматически создала новое имя “unk\_0\_114”, ссылающееся на переменную неопределенного типа размером в **байт**, но команда “LODSW” загружает в регистр AX **слово**, поэтому необходимо перейти к строке :0144 и дважды нажать <D> пока экран не станет выглядеть так:

```
seg000:0114 word_0_114      dw 18h                ; DATA XREF: seg000:0106↑o
seg000:0116                db 0D2h ; T
```

Но что именно содержится в ячейке “word\_0\_144”? Понять это позволит изучение следующего кода:

```
seg000:0106                mov     si, offset word_0_114
seg000:0109                lodsw
seg000:010A                xchg  ax, cx
seg000:010B                push  si
seg000:010C                ;
seg000:010C loc_0_10C:      ; CODE XREF: seg000:0110↓j
seg000:010C                xor   byte ptr [si], 66h
seg000:010F                inc   si
seg000:0110                loop  loc_0_10C
```

В строке :0x10A значение регистра AX помещается в регистр CX, и затем он используется командой “LOOP LOC\_010C” как счетчик цикла. Тело цикла представляет собой простейший расшифровщик – команда “XOR” расшифровывает один байт, на который указывает регистр SI, а команда “INC SI” перемещает указатель на следующий байт. Следовательно, в ячейке “word\_0\_144” содержится количество байт, которые необходимо расшифровать. Подведя к ней курсор, нажатием клавиши <N> можно дать ей осмысленное имя, например “BytesToDecrypt”.

После завершения цикла расшифровщика встречается еще один безусловный регистровый переход.

```
seg000:0112                jmp    si
```

Чтобы узнать куда именно он передает управление, необходимо проанализировать код и определить содержимое регистра SI. Часто для этой цели прибегают к помощи отладчика – устанавливают точку останова в строке 0x112 и дождавшись его «всплытия» просматривают значения регистров. Специально для этой цели, IDA Pro поддерживает генерацию map-файлов, содержащих символьную информацию для отладчика. В частности, чтобы не заучивать численные значения всех «подопытных» адресов, каждому из них можно присвоить легко запоминаемое символьное имя. Например, если подвести курсор к строке “seg000:0112”, нажать <N> и ввести “BreakHere”, отладчик сможет автоматически вычислить обратный адрес по его имени.

Для создания map-файла в меню “File” необходимо кликнуть по «Produce output file» и в развернувшемся подменю выбрать «Produce MAP file» или вместо всего этого нажать на клавиатуре «горячую» комбинацию «Shift-F10». Независимо от способа вызова на экран должно появиться диалоговое окно следующего вида. Оно позволяет выбрать какого рода данные будут включены в map-файл – информация о сегментах, имена автоматически сгенерированные IDA Pro (такие как, например, “loc\_0\_106”, “sub\_0x110” и т.д.) и «размангленные» (т.е. приведенные в читабельный вид) имена. Подробнее о сегментах рассказывается в главе «Сегменты и селекторы», об авто генерируемых и замангленных именах - в главе «Настойки IDA Pro”.

Содержимое полученного map-файла должно быть следующим:

Start	Stop	Length	Name	Class
00100H	0013BH	0003CH	seg000	CODE
			Address	Publics by Value

```

0000:0100      start
0000:0112      BreakHere
0000:0114      BytesToDecrypt
Program entry point at 0000:0100

```

Такой формат поддерживают большинство отладчиков, в том числе и популярнейший Soft-Ice, в поставку которого входит утилита “msym”, запускаемая с указанием имени конвертируемого map-файла в командной строке. Полученный sym-файл необходимо разместить в одной директории с отлаживаемой программой, загружаемой в загрузчик **без указания расширения**, т.е., например, так “WLDR Crypt”. В противном случае символьная информация не будет загружена!

Затем необходимо установить точку останова командой “brx BreakHere” и покинуть отладчик командой “x”. Спустя секунду его окно вновь появиться на экране, извещая о достижении процессором контрольной точки. Посмотрев на значения регистров, отображаемых по умолчанию вверху экрана, можно выяснить, что содержимое SI равно 0x12E.

С другой стороны, это же значение можно вычислить «в уме», не прибегая к отладчику. Команда MOV в строке 0x106 загружает в регистр SI смещение 0x114, откуда командой LODSW считывается количество расшифровываемых байт – 0x18, при этом содержимое SI увеличивается на размер слова – два байта. Отсюда, в момент завершения цикла расшифровки значение SI будет равно  $0x114+0x18+0x2 = 0x12E$ .

Вычислив адрес перехода в строке 0x112, рекомендуется создать соответствующую перекрестную ссылку (from: 0x122; to: 0x12E) и добавить комментарий к строке 0x112 (“Переход по адресу 012E”). Создание перекрестной ссылки автоматически дизассемблирует код, начиная с адреса seg000:012E и до конца файла.

```

seg000:012E loc_0_12E:          ; CODE XREF: seg000:0112u
seg000:012E      call  $+3
seg000:0131      pop   cx
seg000:0132      pop   si
seg000:0133      mov  di, 100h
seg000:0136      push di
seg000:0137      sub  cx, si
seg000:0139      repe movsb
seg000:013B      retn

```

Назначение команды “CALL \$+3” (где \$ обозначает текущее значение регистра указателя команд IP) состоит в заталкивании в стек содержимого регистра IP, откуда впоследствии оно может быть извлечено в любой регистр общего назначения. Необходимость подобного трюка объясняется тем, что в микропроцессорах серии Intel 80x86 регистр IP не входит в список непосредственно адресуемых и читать его значение могут лишь команды, изменяющие ход выполнения программы, в том числе и call.

Для облегчения анализа листинга можно добавить к стокам 0x12E и 0x131 комментарий – “MOV CX, IP”, или еще лучше – сразу вычислить и подставить непосредственное значение – “MOV CX,0x131”.

Команда “POP SI” в строке 0x132 снимает слово из стека и помещает его в регистр SI. Прокручивая экран дизассемблера вверх в строке 0x10B можно обнаружить парную ей инструкцию “PUSH SI”, заносящую в стек смещение первого расшифровываемого байта. После этого становится понятным смысл последующих команд “MOV DI, 0x100\SUB CX,SI\REPE MOVSB”. Они перемещают начало расшифрованного фрагмента по адресу, начинающегося со смещения 0x100. Такая операция характерна для «конвертных» защит, накладывающихся на уже откомпилированный файл, который перед запуском должен быть размещен по своим «родным» адресам.

Перед началом перемещения в регистр CX заносится длина копируемого блока, вычисляемая путем вычитания смещения первого расшифрованного байта от смещения второй команды перемещающего кода. В действительности, истинная длина на три байта

короче и по идее от полученного значения необходимо вычесть три. Однако, такое несогласование не нарушает работоспособности, поскольку содержимое ячеек памяти, лежащих за концом расшифрованного фрагмента, не определено и может быть любым.

Пара команд "0x136:PUSH DI" и "0x13B:RETN" образуют аналог инструкции "CALL DI" – "PUSH" заталкивает адрес возврата в стек, а "RETN" извлекает его оттуда и передает управление по соответствующему адресу. Зная значение DI (оно равно 0x100) можно было бы добавить еще одну перекрестную ссылку ("from:0x13B; to:0x100") и комментарий к строке :0x13B – "Переход по адресу 0x100", но ведь к этому моменту по указанным адресам расположен совсем другой код! Поэтому, логически правильнее добавить перекрестную ссылку "from:0x13B; to:0x116" и комментарий "Переход по адресу 0x116".

Сразу же после создания новой перекрестной ссылки IDA попытается дизассемблировать зашифрованный код, в результате чего получится следующее:

```

seg000:0116 loc_0_116:                ; CODE XREF: seg000:013Bu
seg000:0116                shr                byte ptr [bx-24h], cl
seg000:0119                outsb
seg000:011A                stos               word ptr es:[edi]
seg000:011C                inc                di
seg000:011D                movsw
seg000:011E                add                cx, cs:[bp+si]
seg000:0121                or                 cl, [bx+di]
seg000:0123                dec                dx
seg000:0124                xor                ax, 0F07h
seg000:0127                or                 cl, [bx+di]
seg000:0129                adc                al, 47h
seg000:0129;-----
seg000:012B                db                 6Bh ; k
seg000:012C                db                 6Ch ; l
seg000:012D                db                 42h ; B
seg000:012E;-----

```

Непосредственное дизассемблирование зашифрованного кода невозможно – предварительно его необходимо расшифровать. Подавляющее большинство дизассемблеров не могут модифицировать анализируемый текст налету и до загрузки в дизассемблер исследуемый файл должен быть полностью расшифрован. На практике, однако, это выглядит несколько иначе – прежде чем расшифровывать необходимо выяснить алгоритм расшифровки, проанализировав доступную часть файла. Затем выйти из дизассемблера, тем или иным способом расшифровать «секретный» фрагмент, вновь загрузить файл в дизассемблер (причем предыдущие результаты дизассемблирования окажутся утеряны) и продолжить его анализ до тех пор, пока не встретится еще один зашифрованный фрагмент, после чего описанный цикл «выход из дизассемблера – расшифровка – загрузка - анализ» повторяется вновь.

Достоинство IDA заключается в том, что она позволяет выполнить ту же задачу значительно меньшими усилиями, никуда не выходя из дизассемблера. Это достигается за счет наличия механизма виртуальной памяти, подробно описанного в главе «Виртуальная память». Если не вдаваться в технические тонкости, упрощенно можно изобразить IDA в виде «прозрачной» виртуальной машины, оперирующей с физической памятью компьютера. Для модификации ячеек памяти необходимо знать их адрес, состоящий из пары чисел – сегмента и смещения.

Слева каждой строки указывается ее смещение и имя сегмента, например "seg000:0116". Узнать базовый адрес сегмента по его имени можно, открыв окно «Сегменты» выбрав в меню «View» пункт «Segments».

Name	Start	End	Align	Base	Type	Cls	32es	ss	ds			
seg000	00000100	0000013C	byte	<u>1000</u>	pub	CODE	N	FFFF	FFFF	1000	00010100	0001013C

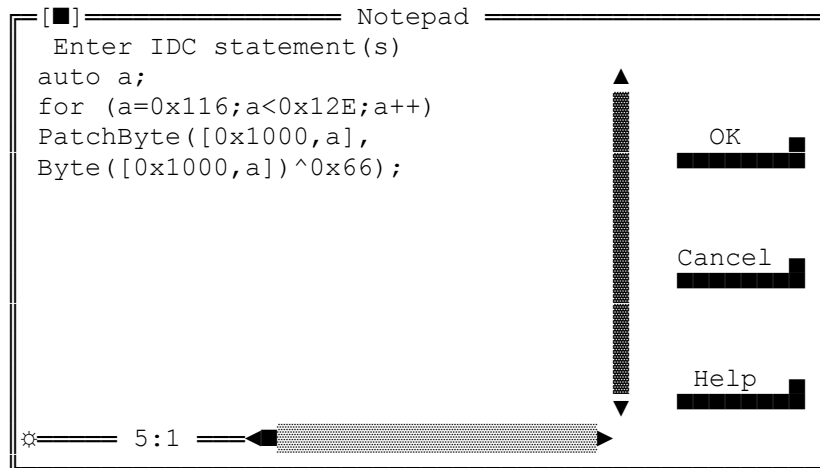


## Рисунок 8 Окно «Сегменты»

Искомый адрес находится в столбце “Base” и для наглядности на приведенной копии экрана выделен жирным шрифтом. Обратится к любой ячейке сегмента поможет конструкция “[segment:offset]”, а для чтения и модификации ячеек предусмотрены функции Byte и PatchByte соответственно. Их вызов может выглядеть, например, так: a=Byte([0x1000,0x100]) – читает ячейку, расположенную по смещению 0x100 в сегменте с базовым адресом 0x1000; PatchByte([0x1000,0x100],0x27) – присваивает значение 0x27 ячейке памяти, расположенной по смещению 0x100 в сегменте с базовым адресом 0x1000. Как следует из названия функций, они манипулируют с ячейками размером в один байт. Существуют так же функции, манипулирующие целыми словами, подробнее о них можно прочесть в главе «Виртуальная память».

Знания этих двух функций вполне достаточно для написания скрипта - расшифровщика при условии, что читатель знаком с языком Си. Реализация IDA-Си не полностью поддерживается стандарта – подробнее об этом рассказывается в главе «Язык скриптов IDA-Си», здесь же достаточно заметить, что IDA не позволяет разработчику задавать тип переменной и определяет его автоматически по ее первому использованию, а объявление осуществляется ключевым словом “auto”. Например, “auto MyVar, s0” объявляет две переменных – MyVar и s0.

Для создания скрипта необходимо нажать комбинацию клавиш <Shift-F2> или выбрать в меню “File” пункт “IDC Command” и в появившемся окне диалога ввести исходный текст программы:



## Рисунок 9 Встроенный редактор скриптов

```

auto a;
for (a=0x116;a<0x12E;a++)
PatchByte ([0x1000,a],Byte ([0x1000,a])^0x66);
  
```

а) исходный текст скрипта - расшифровщика

**Пояснение:** как было показано выше алгоритм расшифровщика сводится к последовательному преобразованию каждой ячейки зашифрованного фрагмента операцией XOR 0x66, (см. ниже – выделено жирным шрифтом)

```

seg000:010C      xor     byte ptr [si], 66h
seg000:010F      inc     si
  
```

```
seg000:0110                                loop    loc_0_10C
```

Сам же зашифрованный фрагмент начинается с адреса `seg000:0x116` и продолжается вплоть до `seg000:0x12E`. Отсюда – цикл расшифровки на языке Си выглядит так: *for (a=0x116;a<0x12E;a++) PatchByte([0x1000,a],Byte([0x1000,a]^0x66));*

В зависимости от версии IDA для выполнения скрипта необходимо нажать либо <Enter> (версия 3.8x и старше), либо <Ctrl-Enter> в более ранних версиях. Если все сделано правильно, после выполнения скрипта экран дизассемблера должен выглядеть так (b).

Возможные ошибки – несоблюдение регистра символов (IDA к этому чувствительна), синтаксические ошибки, базовый адрес вашего сегмента отличается от `0x1000` (еще раз вызовите окно «Сегменты» чтобы узнать его значение). В противном случае необходимо подвести курсор к строке “`seg000:0116`”, нажать клавишу <U> для удаления результатов предыдущего дизассемблирования зашифрованного фрагмента и затем клавишу <C> для повторного дизассемблирования расшифрованного кода.

```
seg000:0116 loc_0_116:                        ; CODE XREF: seg000:013Bu
seg000:0116                mov     ah, 9
seg000:0118                mov     cx, 108h
seg000:011B                int     21h                ; DOS - PRINT STRING
seg000:011B                ; DS:DX -> string terminated by "$"
seg000:011D                retn
seg000:011D ; -----
seg000:011E                db     48h ; H
seg000:011F                db     65h ; e
seg000:0120                db     6Ch ; l
seg000:0121                db     6Ch ; l
seg000:0122                db     6Fh ; o
seg000:0123                db     2Ch ; ,
seg000:0124                db     53h ; S
seg000:0125                db     61h ; a
seg000:0126                db     69h ; i
seg000:0127                db     6Ch ; l
seg000:0128                db     6Fh ; o
seg000:0129                db     72h ; r
seg000:012A                db     21h ; !
seg000:012B                db     0Dh ;
seg000:012C                db     0Ah ;
seg000:012D                db     24h ; $
seg000:012E ; -----
```

b) результат работы скрипта расшифровщика

Цепочку символов, расположенную начиная с адреса “`seg000:011E`” можно преобразовать в удобочитаемый вид, подведя к ней курсор и нажав клавишу “<A>”. Теперь экран дизассемблера будет выглядеть так:

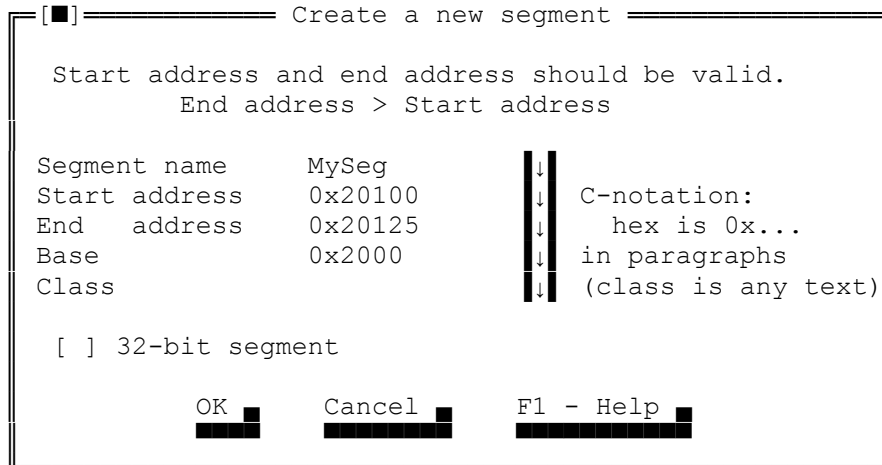
```
seg000:0116 loc_0_116:                        ; CODE XREF: seg000:013Bu
seg000:0116                mov     ah, 9
seg000:0118                mov     cx, 108h
seg000:011B                int     21h                ; DOS - PRINT STRING
seg000:011B                ; DS:DX -> string terminated by "$"
seg000:011D                retn
seg000:011D ; -----
seg000:011E aHelloSailor db 'Hello,Sailor!',0Dh,0Ah,'$'
seg000:012E ; -----
```

c) создание ASCII-строки

Команда “`MOV AH,9`” в строке `:0116` подготавливает регистр AH перед вызовом прерывания `0x21`, выбирая функцию вывода строки на экран, смещение которой заносится следующей командой в регистр DX. Т.е. для успешного ассемблирования листинга необходимо заменить константу `0x108` соответствующим смещением. Но ведь выводимая строка на этапе ассемблирования (до перемещения кода) расположена совсем в другом месте! Одно из возможных решений этой проблемы заключается в создании нового

сегмента с последующим копированием в него расшифрованного кода – в результате чего достигается эмуляция перемещения кода работающей программы.

Для создания нового сегмента можно выбрать в меню «View» пункт «Segments» и в раскрывшемся окне нажать клавишу <Insert>. Появится диалог следующего вида (см. рис. 10):



**Рисунок 10 IDAC: Создание нового сегмента**

Подробнее о значении каждого из полей рассказано в главе «Сегменты и селекторы», а здесь не рассматривается.

**Пояснение:** Базовый адрес сегмента может быть любым если при этом не происходит перекрытия сегментов *seg000* и *MySeg*; начальный адрес сегмента задается так, чтобы смещение первого байта было равно *0x100*; разница между конечным и начальным адресом равна длине сегмента, вычислить которую можно вычитанием смещения начала расшифрованного фрагмента от смещения его конца –  $0x13B - 0x116 = 0x25$ .

Скопировать требуемый фрагмент в только что созданный сегмент можно скриптом следующего содержания.

```
auto a;
for (a=0x0;a<0x25;a++) PatchByte([0x2000,a+0x100],Byte([0x1000,a+0x116]));
```

а) исходный текст скрипта - копировщика

Для его ввода необходимо вновь нажать <Shift-F2>, при этом предыдущий скрипт будет утерян (IDA позволяет работать не более чем с один скриптом одновременно). После завершения его работы экран дизассемблера будет выглядеть так:

```
MySeg:0100 MySeg            segment byte public '' use16
MySeg:0100                assume cs:MySeg
MySeg:0100                ;org 100h
MySeg:0100                assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
MySeg:0100                db 0B4h ; |
MySeg:0101                db 9 ;
MySeg:0102                db 0EAh ; ||
MySeg:0103                db 8 ;
MySeg:0104                db 1 ;
MySeg:0105                db 0CDh ; =
MySeg:0106                db 21h ; !
MySeg:0107                db 0C3h ; |
MySeg:0108                db 48h ; H
MySeg:0109                db 65h ; e
```

```

MySeg:010A          db  6Ch ; l
MySeg:010B          db  6Ch ; l
MySeg:010C          db  6Fh ; o
MySeg:010D          db  2Ch ; ,
MySeg:010E          db  53h ; S
MySeg:010F          db  61h ; a
MySeg:0110          db  69h ; i
MySeg:0111          db  6Ch ; l
MySeg:0112          db  6Fh ; o
MySeg:0113          db  72h ; r
MySeg:0114          db  21h ; !
MySeg:0115          db  0Dh ;
MySeg:0116          db  0Ah ;
MySeg:0117          db  24h ; $
MySeg:0117 MySeg   ends

```

b) результат работы скрипта-копировщика

Теперь необходимо создать перекрестную ссылку “from:seg000:013B; to:MySeg:0x100”, преобразовать цепочку символов в удобочитаемую строку, подведя курсор к строке MySeg:0108 и нажав клавишу <A>. Экран дизассемблера должен выглядеть так:

```

MySeg:0100 loc_1000_100:          ; CODE XREF: seg000:013Bu
MySeg:0100          mov     ah, 9
MySeg:0102          mov     dx, 108h
MySeg:0105          int     21h          ; DOS - PRINT STRING
MySeg:0105          ; DS:DX -> string terminated by "$"
MySeg:0107          retn
MySeg:0107 ;
MySeg:0108 aHelloSailorS db 'Hello,Sailor!',0Dh,0Ah
MySeg:0108          db '$'
MySeg:0118 MySeg   ends

```

c) результат дизассемблирования скопированного фрагмента

Результатом всех этих операций стало совпадение смещения строки со значением, загружаемым в регистр DX (в тексте они выделены жирным шрифтом). Если подвести курсор к константе “108h” и нажать клавишу <Ctrl-O> она будет преобразована в смещение:

```

MySeg:0102          mov     dx, offset aHelloSailorS ; "Hello,Sailor!\r\n$u"
MySeg:0105          int     21h          ; DOS - PRINT STRING
MySeg:0105          ; DS:DX -> string terminated by "$"
MySeg:0107          retn
MySeg:0107 ;
MySeg:0108 aHelloSailorS db 'Hello,Sailor!',0Dh,0Ah ; DATA XREF: MySeg:0102o

```

d) преобразование константы в смещение

Полученный листинг удобен для анализа, но все еще не готов к ассемблированию, хотя бы уже потому, что никакой ассемблер не в состоянии зашифровать требуемый код. Конечно, эту операцию можно выполнить вручную, после компиляции, но IDA позволит проделать то же самое не выходя из нее и не прибегая к помощи стороннего инструментария.

Демонстрация получится намного нагляднее, если в исследуемый файл внести некоторые изменения, например, добавить ожидание клавиши на выходе. Для этого можно прибегнуть к интегрированному в IDA ассемблеру, но прежде, разумеется, необходимо несколько «раздвинуть» границы сегмента MySeg, дабы было к чему дописывать новый код.

Выберете в меню “View” пункт “Segments” и в открывшемся окне подведите курсор к строке “MySeg”. Нажатие <Ctrl-E> открывает диалог свойств сегмента, содержащий среди прочих полей конечный адрес, который и требуется изменить. Не обязательно указывать точное значение – можно «растянуть» сегмент с небольшим запасом от предполагаемых изменений.

Если попытаться добавить к программе код “XOR AX,AX; INT 16h” он неминуемо

затрет начало строки "Hello, Sailor!", поэтому, ее необходимо заблаговременно передвинуть немного «вниз» (т.е. в область более старших адресов), например, с помощью скрипта следующего содержания «for(a=0x108;a<0x11A;a++) PatchByte([0x2000,a+0x20],Byte([0x2000,a]));».

**Пояснение:** объявление переменной a для краткости опущено (сами должны понимать, не маленькие :-), длина строки, как водится, берется с запасом, чтобы не утомлять себя лишними вычислениями и перемещение происходит справа налево, поскольку исходный и целевой фрагменты заведомо не пересекаются.

Подведя к курсор к строке :0128 нажатием <A> преобразуем цепочку символов к удобно-читаемому виду; подведем курсор к строке :0102 и, выбрав в меню "Edit" пункт "Path program", "Assembler", введем команду "MOV DX,128h", где «128h» - новое смещение строки, и тут же преобразуем его в смещение нажатием <Ctrl-O>.

Вот теперь можно вводить новый текст – переместив курсор на инструкцию "ret", вновь вызовем ассемблер и введем "XOR AX,AX<ENTER>INT 16h<Enter>RET<Enter><Esc>". На последок рекомендуется произвести «косметическую» чистку – уменьшить размер сегмента до необходимого и переместить строку "Hello, Sailor" вверх, прижав ее вплотную к коду.

**Пояснение:** удалить адреса, оставшиеся при уменьшении размеров сегмента за его концом можно взводом флажка "Disable Address" в окне свойств сегмента, вызываемом нажатием <Alt-S>

Если все было сделано правильно конечный результат должен выглядеть как показано ниже:

```

seg000:0100 ; File Name   : F:\IDAN\SRC\Crypt.com
seg000:0100 ; Format       : MS-DOS COM-file
seg000:0100 ; Base Address: 1000h Range: 10100h-1013Ch Loaded length: 3Ch
seg000:0100
seg000:0100
seg000:0100 ; =====
seg000:0100
seg000:0100 ; Segment type: Pure code
seg000:0100 seg000      segment byte public 'CODE' use16
seg000:0100                assume cs:seg000
seg000:0100                org 100h
seg000:0100                assume es:nothing, ss:nothing, ds:seg000, fs:nothing, gs:nothing
seg000:0100
seg000:0100 ; ----- S U B R O U T I N E -----
seg000:0100
seg000:0100
seg000:0100                public start
seg000:0100 start      proc near
seg000:0100                add     si, 6
seg000:0103                jmp     si             ; ĩãðãõĩã ĩĩ äãðãñó 0106
seg000:0103 start      endp
seg000:0103
seg000:0103 ; -----
seg000:0105                db 0B9h ; !
seg000:0106 ; -----
seg000:0106                mov     si, offset BytesToDecrypt
seg000:0109                lodsw
seg000:010A                xchg   ax, cx
seg000:010B                push  si
seg000:010C
seg000:010C loc_0_10C:                ; CODE XREF: seg000:0110j
seg000:010C                xor     byte ptr [si], 66h
seg000:010F                inc     si
seg000:0110                loop  loc_0_10C
seg000:0112
seg000:0112 BreakHere:                ; ĩãðãõĩã ĩĩ äãðãñó 012E
seg000:0112                jmp     si
seg000:0112 ; -----
seg000:0114 BytesToDecrypt dw 18h                ; DATA XREF: seg000:0106o
seg000:0116 ; -----

```

```

seg000:0116
seg000:0116 loc_0_116:                ; CODE XREF: seg000:013Bu
mov     ah, 9
seg000:0118 mov     cx, 108h                ; "Hello,Sailor!\r\n$"
seg000:011B int     21h                    ; DOS - PRINT STRING
seg000:011D retn                                ; DS:DX -> string terminated by "$"
seg000:011D ; -----
seg000:011E aHelloSailor db 'Hello,Sailor!',0Dh,0Ah,'$' ; DATA XREF: seg000:0118o
seg000:012E ; -----
seg000:012E
seg000:012E loc_0_12E:                ; CODE XREF: seg000:0112u
call    $+3
seg000:0131 pop     cx
seg000:0132 pop     si
seg000:0133 mov     di, 100h
seg000:0136 push    di
seg000:0137 sub     cx, si
seg000:0139 repe movsb
seg000:013B retn
seg000:013B seg000 ends
seg000:013B
MySeg:0100 ; -----
MySeg:0100 ; =====
MySeg:0100
MySeg:0100 ; Segment type: Regular
MySeg:0100 MySeg segment byte public '' usel6
MySeg:0100 assume cs:MySeg
MySeg:0100 ;org 100h
MySeg:0100 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
MySeg:0100
MySeg:0100 loc_1000_100:                ; CODE XREF: seg000:013Bu
MySeg:0100 mov     ah, 9
MySeg:0102 mov     dx, offset aHelloSailor_0 ; "Hello,Sailor!\r\n$"
MySeg:0105 int     21h                    ; DOS - PRINT STRING
MySeg:0105 retn                                ; DS:DX -> string terminated by "$"
MySeg:0107 xor     ax, ax
MySeg:0109 int     16h                    ; KEYBOARD - READ CHAR FROM BUFFER, WAIT IF EMPTY
MySeg:0109 retn                                ; Return: AH = scan code, AL = character
MySeg:010B ; -----
MySeg:010C aHelloSailor_0 db 'Hello,Sailor!',0Dh,0Ah,'$' ; DATA XREF: MySeg:0102o
MySeg:010C MySeg ends
MySeg:010C
MySeg:010C
MySeg:010C end start

```

#### а) окончательно дизассемблированный текст

Структурно программа состоит из следующих частей – расшифровщика, занимающего адреса seg000:0x100 – seg000:0x113, переменной размером в слово, содержащей количество расшифровываемых байт, занимающей адреса seg000:0x114-seg000:0x116, исполняемого кода программы, занимающего целиком сегмент MySeg и загрузчика, занимающего адреса seg000:0x12E-seg000:0x13B. Все эти части должны быть в перечисленном порядке скопированы в целевой файл, причем исполняемый код программы необходимо предварительно зашифровать, произведя над каждым его байтом операцию XOR 0x66.

Ниже приведен пример скрипта, автоматически выполняющего указанные действия. Для его загрузки достаточно нажать <F2> или выбрать в меню “File” пункт “Load file”, “IDC file”.

```

// Компилятор для файла Crypt
//
static main()
{
auto a, f;

// Открывается файл Crtypt2.com для записи в двоичном режиме

```

```

f=fopen("crypt2.com","wb");

// В файл Crypt2 копируется расшифровщик
for (a=0x100;a<0x114;a++) fputc(Byte([0x1000,a]),f);
// Определяется и копируется в файл слово, содержащее число
// байтов для расшифровки
fputc( SegEnd([0x2000,0x100]) - SegStart([0x2000,0x100]),f);
fputc(0,f);

// Копируется и налету шифруется расшифрованный фрагмент
for (a=SegStart([0x2000,0x100]);a!=SegEnd([0x2000,0x100]);a++)
fputc(Byte(a) ^ 0x66,f);

// Дописывается загрузчик
for (a=0x12E;a<0x13C;a++)
fputc(Byte([0x1000,a]),f);

// Закрывается файл.
fclose(f);
}

```

а) исходный код скрипта-компилятора

Подробное объяснение каждой функции, встретившийся в скрипте, можно найти в главах «Сегменты и селекторы», «Файловый ввод-вывод» и т.д.

Выполнение скрипта приведет к созданию файла “Crypt2.com”, запустив который можно убедиться в его работоспособности – он выводит строку на экран и, дождавшись нажатия любой клавиши, завершает свою работу.

Огромным преимуществом такого подхода является «сквозная» компиляция файла, т.е. дизассемблированный листинг в действительности не ассемблировался! Вместо этого из виртуальной памяти байт-за-байтом читалось оригинальное содержимое, которое за исключением модифицированных строк доподлинно идентично исходному файлу. Напротив, повторное ассемблирование практически никогда не позволяет добиться полного сходства с дизассемблируемым файлом.

IDA – очень удобный инструмент для модификации файлов, исходные тексты которых утеряны или отсутствуют; она практически единственный дизассемблер, способный анализировать зашифрованные программы, не прибегая к сторонним средствам; она обладает развитым пользовательским интерфейсом и удобной системой навигации по исследуемому тексту; она дает может справиться с любой мыслимой и немислимой задачей...

...но эти, и многие другие возможности, невозможно реализовать в полной мере, без владения языком скриптов, что и подтвердил приведенный выше пример.

## Язык скриптов IDA Си

IDA поддерживает Си-подобный скрип-язык, в целом напоминающий Си Керигана и Ричи, но значительно упрощенный, не поддерживающий типов, массивов, указателей и в том числе не обладающий возможностью отладки приложений. Язык скриптов изначально задумывался как средство выполнения простейших операций, укладываемых в десяток-другой строк кода, а для «серьезных» задач предусмотрен механизм **плагинов** – внешних модулей, написанных на Borland C++ или Microsoft Visual C++ и подключаемых к IDA по мере необходимости. Впрочем, в подавляющем большинстве случаев нет никакой нужды прибегать к плагинам и вполне можно обойтись встроенным скрипт-языком.

Ниже приводится краткое описание языка скриптов IDA Си, рассчитанное на читателя знакомого и владеющего «классическим» Си.

### Консоль

Простейшие скрипты могут быть набраны «не отходя от кассы» в диалоге, вызываемом нажатием <Shif-F2>, в дальнейшем именуемым **консолью**. Достоинство этого

метода – оперативность, а недостатки - жесткое ограничение максимально допустимого размера вводимого текста (порядка одного килобайта) и невозможность работать более чем с одним скрипом одновременно.

Более сложные скрипты как правило создаются в отдельном файле, загружаемый нажатием <F2> или заданием ключа “-Sfilename” в командной строке, причем между “-S” и именем файла не должно присутствовать символа пробела.

Результаты работы скрипта (и служебные сообщения самой IDA) выводятся в специальную область экрана, напоминающую собой бесконечную телетайпную ленту, условно именуемую «консолью». Поскольку одна консоль связана с **вводом** (исходного текста скрипта), а другая с **выводом** (результатов работы), никакого противоречия не возникает и по контексту легко понять о какой именно консоли идет речь.

**Рисунок 11 0x017 IDAC: Область вывода результатов работы скрипта «консоль»**

**Рисунок 12 0x018 IDAG: Область вывода результатов работы скрипта «консоль»**

### Функции, объявление функций, аргументы функции, возвращаемое значение

IDA поддерживает функции двух типов – встроенные (**build-in**) функции и функции, определяемые пользователем. Объявление пользовательских функций происходит следующим образом – перед именем функции указывается ключевое слово “**static**”, а за именем в круглых скобках через запятую перечисляются аргументы (если они есть) без указания их типа, например, так:

```
static MyFunc()                static MyOtherFunc(x,y,s0)
{                               {
    // тело функции;           // тело функции;
}
```

**Ограничения:** максимально допустимая длина имени функции равна 16 символам; возможность организовывать вложенные функции отсутствует.

Консоль не поддерживает объявления функций, сообщая при этом о синтаксической ошибке, функции могут быть объявлены только в idc-файлах.

Все функции, объявленные как “static” (т.е. все функции вообще, поскольку, локальные функции IDA не поддерживает) будучи однажды загруженными, резидентно остаются в памяти на протяжении всего сеанса работа с дизассемблером и в любой момент доступны для вызова из консоли или других программ.

Например, если создать idc-файл следующего содержания: «*static Demo(s0) {Warning(s0);}*», загрузить его нажатием <F2>, нажать <Shift-F2> для вызова консоли и набрать нечто наподобие “*Demo(“Hello, IDA!”);*” появится диалоговое окно, выводящее указанную строку на экран.

static-функции не могут быть выгружены из памяти, но могут перекрываться повторным объявлением. Встроенные в IDA функции перекрыть нельзя.

Если в тексте скрипта объявлена функция main(), она автоматически выполняется при его загрузке.

Возврат значения функции осуществляется посредством “return” с последующим указанием имени переменной или константы, например, так:

```
static MyFunc(x,y)            staic MyOtherFunc()
{                               {
    return x-y;                return “Hello, IDA Pro\\n”;
```

Пустой “return”, равно как и его отсутствие, возвращает нулевое значение (пустую строку).



## Объявление переменных, типы переменных, преобразования переменных

Для объявления переменных используется ключевое слово **auto** за которым последовательно перечисляются декларируемые переменные, отделенные друг от друга запятой. Инициализация при объявлении не поддерживается.

Например:

```
auto x;                auto x, y, z;  
а) верно
```

```
auto x=1;  
б) неверно
```

Все переменные, объявленные таким образом, локальные; поддержка глобальных переменных, доступных всем функциям отсутствует. Вместо этого IDA поддерживает **виртуальные массивы**, доступные всем функциям всех загруженных скриптов одновременно (см. главу ??? «Виртуальные массивы»).

Имя переменной не должно совпадать ни с одним зарезервированным ключевым словом. Зарезервированные ключевые слова в IDA те же самые, что и в Си.

IDA поддерживает два типа переменных – переменные типа **строка** и переменные типа **число**.

Строковые переменные ограничены максимальной длиной в 255 символов и заканчиваются нулем. Над ними доступны три операции – **присвоение**, **контекция** (слияние) и **сравнение**.

Переменные типа число в свою очередь делятся на два подтипа: **long** – 32-разрядное знаковое целое и **float** – число с плавающей запятой до 25 разрядов. Над ними обоими доступны операции – **присвоение**, **сравнение**, **сложение**, **вычитание**, **умножение**, **деление**. Над целочисленными переменными доступны битовые операции - **циклический сдвиг** вправо и влево, битовое **И**, битовое **НЕ**, битовое **ИЛИ**, битовое **НЕТ** и битовое **ИЛИ-ИСКЛЮЧАЮЩЕЕ-И**.

Преобразования типов переменных осуществляются автоматически и происходят в следующих случаях:

- если левая переменная – строка, то и правая переменная преобразуется в строку;
- если левая переменная – длинное целое, а правая – строка, правая переменная преобразуется в длинное целое;
- если одна из переменных имеет тип float, все остальные переменные преобразуются в тип float;
- если над переменной выполняются одна из битовых операций, она преобразуется в длинное целое

Преобразование «строка → длинное целое» осуществляется по следующей схеме: если левая часть строки представляет собой число, записанное в десятичной нотации, результат преобразования равен этому числу, в противном случае – нулю.

Примеры:

```
auto s0, s1, s2, s3;  
s0="1234"; s1="1234mystring", "0x1234", "MyString";  
Message(">%d, %d, %d, %d\n", s0, s1, s2, s3);
```

```
>1234, 1234, 0, 0
```

Преобразование «длинное целое → строка» осуществляется путем дописывания завершающего нуля к преобразуемому значению.

Примеры:

```
auto a,b;
a=0x21;b=0x22232425;
Message(">%s,%s\n",a,b);

> !,%$#"
```

Преобразование «строка → float» осуществляется аналогично преобразованию «строка → длинное целое», включая дробную часть; если же преобразование невозможно, результат равен нулю.

Например:

```
auto s0,s1;
s0="1.1"; s1="MyString";
Message(">%f,%f\n",s0,s1);

> 1.1, 0.
```

Преобразование «float → строка» в отличие от преобразования «длинное целое → строка» осуществляет действительно корректное преобразование, сравните:

```
auto f;
f="1.2";
Message(">%s\n",f);

> 1.2

auto a;
a=0x21;
Message(">%s\n",a);

>!
```

При сложении двух строковых переменных происходит их контакция (слияние); при вычитании одной строки из другой левый операнд игнорируется, а над каждым символом правого выполняется операция NEG (дополнение до нуля); при делении и умножении строк друг на друга они преобразуются к длинному целому. Операции вычитания, деления и умножения строк недокументированны и по-разному могут вести себя в зависимости от версии IDA. Рекомендуется воздержаться от использования их в собственных программах.

Смешивание операндов различных типов заставляет внимательно относиться к порядку расстановки их выражении – результат может зависеть от перемены мест слагаемых, например:

```
auto x,s0;x=1;s0="3h";
Message(">%x\n",x+s0);

>4

auto x,s0;x=1;s0="3h";
Message(">%x\n",s0+x);

>3
```

**Пояснение:** в первом случае левая переменная – длинное целое, поэтому, строка “3h” преобразуется к длинному целому, в результате чего получается: 1+3=4; во втором случае левая переменная – строка, поэтому, длинное целое #1 преобразуется к строке, в результате чего получается: “3h”+“\x1” = “3h\x1”, но затем “3h\x1” вновь преобразуется к длинному целому, ибо того требует спецификатор “%x”, а “3h\x1” = #3. Так что от перемены мест слагаемых сумма меняется!

Смещение типов – частый источник трудноуловимых ошибок, и лучше использовать явное преобразование вызовом следующих функций:

- **long ( переменная )** // Преобразует переменную в длинное целое
- **char ( переменная )** // Преобразует переменную в строку

- `float( переменная )` // Преобразует переменную в тип float

Например:

```
auto x, s0; x=1; s0="3h";           auto x, s0; x=1; s0="3h";
Message(">%d\n", long(s0)+x);      Message(">%d\n", x+long(s0));

>4                                 >4
```

На этот раз от перемены мест слагаемых сумма не изменяется!

## Директивы

IDA поддерживает следующие стандартные директивы препроцессора:

- **#define**
- **#undef**
- **#include**
- **#error**
- **#ifdef#ifndef#else#endif**

**Внимание:** консоль не поддерживает никаких директив препроцессора – их можно использовать только в IDC-файлах.

**Замечание:** для использования определений, констант, приводимых в этой книге по ходу описания функций, в исходный код скрипта необходимо включить директиву **#include <idc.idc>**.

Вызов функций возможен и без включения файла `<idc.idc>` в исходный листинг – это необходимо только для использования определений, например, таких, как `BADADDR`.

До вызова консоли IDA самостоятельно подключает к ней этот файл, делая доступными все определения.

## Предписания

IDA поддерживает следующие стандартные конструкции, изменяющие нормальный ход выполнения программы:

- `if, else;`
- `for;`
- `while, do, break, continue;`
- `return`

**Замечание:** цикл `for (expr1; expr2; expr3 ) statement` в отличие от стандартного языка Си не поддерживает более одного счетчика.

## Математические и битовые операторы

Поддерживаются следующие математические операторы – сложение: “+”, вычитание: “-“, умножение: “\*”, деление “/”, приращение на единицу “++”. Операторы “+=” и “-=” не поддерживаются.

Поддерживаются следующие битовые операторы – битовое **И**: “&”, битовое **ИЛИ**: “|”, битовое **НЕТ**: “!”, битовое **ИЛИ-ИСКЛЮЧАЮЩЕЕ-И**: “^”.

## Массивы

Встроенной языковой поддержки массивов, наподобие той, что присутствует в Си, у IDA нет.

## **ВИРТУАЛЬНАЯ ПАМЯТЬ**

### **Архитектура виртуальной памяти**

В отличие от многих других дизассемблеров, IDA работает не с исследуемым файлом, а с его образом, загруженным в **виртуальную память**. IDA эмулирует загрузчик операционной системы, благодаря чему образ загруженного в дизассемблер файла идентичен образу того же файла, загруженного операционной системой.

IDA использует плоскую 32-разрядную модель виртуальной памяти, предоставляя в распоряжение пользователя немногим менее четырех гигабайт адресного пространства. Наибольший возможный адрес равен 0xFF000000 и для удобства определен в файле <idc.idc> через константу **MAXADDR**. Попытка задания больших адресов приведет к ошибке.

Адресное пространство виртуальной памяти может быть разбито на один или более сегментов (см. главу «Сегменты и селекторы»), однако, подавляющее большинство встроенных функций IDA ожидают не сегментных, а **линейных** адресов.

Адресное пространство виртуальной памяти не непрерывно – в нем могут присутствовать «выключенные» адреса, попытка обращения к которым приведет к ошибке. IDA не предоставляет функций, позволяющих манипулировать «включением» - «выключением» адресов. Эта операция может быть выполнена только косвенно – адреса «включаются» при загрузке бинарного файла и создании нового сегмента, а выключаются при его удалении (см. описание функции SegCreate).

С каждым «включенным» адресом виртуальной памяти связана специальная структура данных, включающая в себя 8-разрядную **ячейку** виртуальной памяти и 24-разрядное поле атрибутов этой ячейки.

Атрибуты, в частности, указывают следует ли отображать ячейку как инструкцию или как данные, в каком виде следует представлять операнды и т.д. Назначение каждого бита атрибутов подробно описано в главах «Элементы», «Типы элементов», «Операнды» и «Объекты». Поле атрибутов доступно не только для косвенного (посредством вызова встроенных в IDA Функций), но и непосредственного чтения и изменения. Однако, разработчики IDA настоятельно рекомендуют избегать непосредственной модификации, поскольку, назначение тех или иных битов атрибутов в последующих версиях дизассемблера может измениться!

Ячейка может иметь как **инициализированное**, так и **неинициализированное** значение. Попытка чтения неинициализированной ячейки возвращает ошибку. Определить инициализирована ячейка или нет можно по состоянию младшего бита поля атрибутов – если ячейка инициализирована он равен одному, а если неинициализированная – нулю.

32-разрядное целое, содержащее ячейку и связанные с ней атрибуты, называется **флагом** виртуальной памяти (см. рис. 13).

32	16	8	0
атрибуты			ячейка

**Рисунок 13** Строение флага виртуальной памяти

Флаги виртуальной памяти хранятся в **виртуальном массиве**, подробнее об организации которого рассказано в главе «Виртуальные массивы».

Сами же виртуальные массивы хранятся в **страничной** физической памяти. Совокупность страниц физической страничной памяти в документации и контекстовой помощи IDA называется виртуальной памятью. Такая терминологическая путаница требует постоянного уточнения о чем собственно идет речь – образе загруженного файла или виртуальном массиве, поэтому, во избежание двусмысленного понимания, здесь и далее память, хранящая виртуальные массивы, будет называться страничной, а память, хранящая образ загруженного файла – виртуальной.

### Технические детали:

Виртуальное адресное пространство представляет собой совокупность индексов всех существующих элементов виртуального массива, а «выключенные» адреса являются отсутствующими индексами виртуального массива.

Виртуальный массив располагается в файле \*.id1, формат заголовка которого приведен в таблице 1.

смещение	длина	значение
0x0	0x4	сигнатура “Va4” – “Virtual Array version 4”
0x4	0x2 (Word)	количество непрерывных регионов памяти
0x6	0x2 (Word)	количество страниц всего (одна страница равна 4 кб)
0x8	0x4 (long)	индекс (он же линейный адрес) первого кванта региона
0xC	0x4 (long)	индекс (он же линейный адрес) последнего кванта региона
0x10	0x4 (long)	смещение индекса первого кванта региона в файле *.id1
+0x4	0x4 (long)	индекс (он же линейный адрес) первого кванта следующего региона
+0x4	0x4 (long)	индекс (он же линейный адрес) последнего кванта региона
+0x4	0x4 (long)	смещение индекса первого кванта следующего региона в файле *.id1
...	...	...

**Таблица 1 структура файла \*.id1**

Если открыть файл “tutor.id1” в любом шестнадцатеричном редакторе (**внимание, это необходимо сделать до выхода из IDA, т.к. при выходе он будет автоматически удален**), его начало должно выглядеть так:

```
00000000: 56 61 34 00 02 00 03 00 | 66 06 01 00 78 06 01 00  Va4  ● ♥ f▲☉ x▲☉
00000010: 98 39 00 00 77 07 01 00 | 89 07 01 00 DC 5D 00 00  III9 w•☉ Й•☉ ■]
```

??? #Художнику – используя файл 0x019\_о выделить указанные ниже элементы графически (кружочком) со стрелочкой, указывающей на его отношение к нижеследующему описанию, причем левую часть (т.е. указание самого значения поля, отделенную двоеточием) затем удалить.

“Va4”:                   сигнатура, позволяющая распознать тип файла  
00 02:                   два непрерывных адресных пространства 0x10666 – 0x10677 и  
0x10777 и 0x10788  
00 03:                   три страницы виртуальной памяти израсходовано (размер страницы  
4 кб).  
66 06 01 00:           начальный адрес первого непрерывного регион виртуальной памяти  
78 06 01 00:           конечный адрес первого непрерывного региона виртуальной памяти  
98 39 00 00:           смещение, по которому хранится первый регион виртуальной  
памяти в этом файле

77 07 01 00: начальный адрес второго непрерывного региона виртуальной памяти  
 89 07 01 00: конечный адрес второго непрерывного региона виртуальной памяти  
 DC 5D 00 00: смещение, по которому хранится второй регион виртуальной памяти  
 в этом файле

По смещению 0x003998 (помним об обратном порядке байтов) в файле "tutor.id1" находится следующее содержимое:

```
00003998: 68 21 00 00|45 01 00 00|4C 01 00 00|4C 01 00 00 H! e l l
000039A8: 4F 01 00 00|0C 01 00 00|00 01 00 00|69 01 00 00 o , o I
000039B8: 64 01 00 00|61 01 00 00|00 01 00 00|70 01 00 00 D A o P
000039C8: 52 01 00 00|4F 01 00 00|21 01 00 00|20 01 00 00 r o ! o
000039D8: 0D 01 00 00|FF 01 00 00|00 00 00 00|00 00 00 00 J o o
```

Это и есть флаги виртуальной памяти, содержащие, ячейки и атрибуты виртуальной памяти, читая которые через каждый четвертый байт, можно разобрать "Hello, IDA Pro!"

### Архитектура страничной памяти

Размещение виртуальной памяти в дисковом файле требует некоторого количества оперативной памяти компьютера под кэш-буфера, поскольку побайтовый обмен с диском крайне непроизводителен.

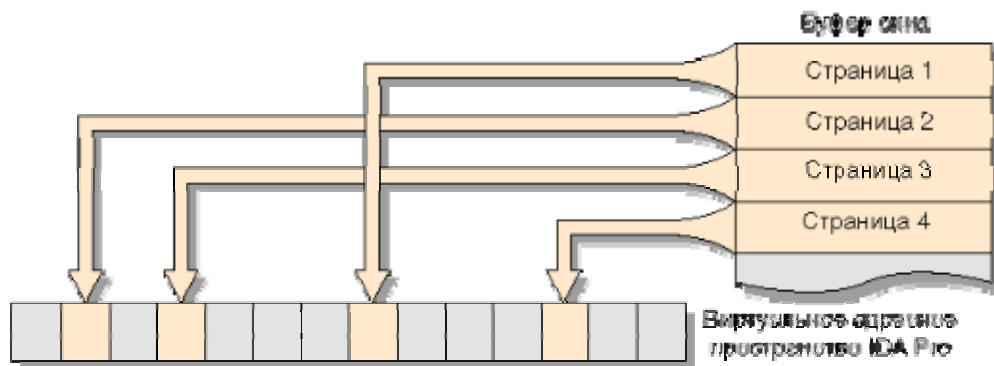
Кэш IDA имеет страничную организацию. Вся виртуальная память разбита на множество блоков одинакового размера, называемых **страницами**. Если происходит обращение к одной ячейке виртуальной памяти, страница, в которой эта ячейка расположена, загружается в оперативную память целиком. Соответственно, модификация одной ячейки виртуальной памяти влечет перезапись всей страницы.

Разбивка на страницы осуществляется маскированием n младших битов целевых адресов, по этой причине размер страницы всегда представляет собой степень двойки. Если размер региона, выделенный под кэш буфер (так же называемый **окном**), превышает размер одной страницы, IDA может загружать в оперативную память несколько страниц одновременно. Соответственно размер окна должен быть кратен размеру страницы.

Если окно целиком заполнено, но требуется загрузить еще одну страницу, IDA просматривает буфер на предмет поиска не модифицированной страницы к которой дольше всего не было обращения. А отыскав такую, замещает ее прочитанной с диска. В противном случае (если все страницы со времени последней загрузки были модифицированы), IDA «сбрасывает» на диск самую старую страницу и только затем замещает ее новой.

Во избежание потерь информации при сбоях питания, зависаниях дизассемблера и т.д., предусмотрен автоматический «сброс» модифицированных страниц через определенные промежутки времени, длительность которых задается значением поля "AUTOSAVE" файла <idatui.cfg> и <idagui.cfg> - консольной и графической версий соответственно. По умолчанию буфера сохраняются после совершения пользователем любых действий или по истечении пяти минут, при этом в окне сообщений появится поясняющая надпись «Flushing buffers, please wait...ok»

**Замечание:** IDA не учитывает «популярность» страницы (т.е. частоту ее использования или количество обращений), а только время последнего обращения (чтения или записи).



??? #Художнику – перерисовать рисунок!

**Рисунок 14 Окно страничной памяти**

Увеличение размера страниц увеличивает вероятность того, что очередной запрашиваемый байт окажется уже загруженным в оперативную память, но в то же время, повышает накладные расходы на сброс модифицированных страниц и уменьшает количество страниц в окне. Поскольку, размер окна сверху ограничен объемом доступной оперативной памяти, возникает задача вычисления оптимального соотношения между размер и количестве страниц.

Размеры и количество страниц задаются полями **“VPAGES”** и **“VPAGESIZE”** файла `<ida.cfg>` соответственно. Если `VPAGES == 0`, IDA пытается самостоятельно вычислить оптимальное значение, исходя из количества требующейся для анализа загруженного файла виртуальной памяти.

Поскольку каждый флаг виртуальной памяти требует четыре байта страничной памяти (т.к. помимо 8 бит содержимого ячейки хранит 24 бита атрибутов), грубо оценить потребности страничной памяти можно умножением размера загружаемого файла на четыре. Разумеется, в зависимости от формата файла истинное значение может значительно отличается от расчетного и предсказанная оценка окажется неверной. Это не нарушит работоспособности дизассемблера, поскольку недостающая память будет выделена автоматически по мере необходимости, однако, неоптимальное соотношение размера и количества страниц ухудшат производительность, поэтому, в некоторых случаях его выгоднее вычислять вручную.

По умолчанию размер страницы (`VPAGESIZE`) в зависимости от версии IDA равен либо 4096 либо 8192 байт, а, поскольку, количество страниц выражается 16-разрядным целым числом, доступное адресное пространство виртуальной памяти равно 64 и 128 мегабайт соответственно. Дальнейшее увеличение размера страниц расширяет границы доступного адресного пространства виртуальной памяти, но одновременно с этим ухудшает производительность за счет большей грануляции, поэтому, прибегать к нему рекомендуется в тех и только в тех случаях, когда требуется свыше 128 мегабайт виртуальной памяти.

Ввиду станичной адресации базы, IDA не позволяет «на лету» изменять размер страниц и при загрузке файлов `*.idb` величина размера страниц берется оттуда, а значения поля `VPAGESIZE` игнорируется. Поэтому, необходимо внимательно отнестись к выбору размера страницы – в дальнейшем изменить его не удастся! Существует возможность необратимо уничтожить результаты своей работы, выбрав слишком малый размер страниц – в этом случае адресного пространства виртуальной памяти может не хватить и попытка выделения очередного блока виртуальной памяти провалится, а сеанс работы с IDA аварийно завершится!

Поле VPAGES, задающие размер буфера окна в страницах, по умолчанию равно нулю, и его значение IDA самостоятельно вычисляет по следующему алгоритму (см. таблицу 2).

??? #Верстальщику CreateNewTable

Размер файла	Размер окна в страницах	Размер окна в байтах
0 КБ -- 255 КБ	$(\text{FILESIZE} * 4) / \text{VPAGESIZE}$	1 МБ
256 КБ – 1023 КБ	$1048576 / \text{VPAGESIZE}$	1 МБ
1024 КБ – 2559 КБ	$\text{FILESIZE} / \text{VPAGESIZE}$	1 - 2,5 МБ
2560 КБ – 10 МБ	$4194304 / \text{VPAGESIZE}$	4 МБ
> 10 МБ	$(\text{FILESIZE} * 2) / (\text{VPAGESIZE} * 5)$	> 4 МБ

**Таблица 2 Алгоритм автоматического выделения памяти**

Важно понять – размер буфера окна не ограничивает количество доступной страничной памяти! Окно – всего лишь кэш-буфер, и IDA будет работать даже в том случае, если уменьшить его до размера одной страницы (правда, это чрезвычайно снизит производительность). Операционные системы Windows и OS/2 позволяют выделить под окно больше памяти, чем ее имеется в наличии, сбрасывая избыток на диск в файл подкачки. В результате количество обращений к диску удваивается и производительность дизассемблера резко падает. Рекомендуется выбирать размер окна таким образом, чтобы он полностью уместился в физической памяти компьютера. Значения, вычисляемые IDA по умолчанию, рассчитаны на компьютеры, обладающие 16 или менее мегабайтами оперативной памяти, при наличии же большего его количества (на конец 2000 года компьютер типичной конфигурации содержит 32-64 мегабайт оперативной памяти) можно **значительно** улучшить производительности, увеличив размер буфера окна.

Помимо виртуальной памяти, содержащей образ загруженного фала, дизассемблеру требуется какое-то количество страничной памяти для хранения меток, имен функций, комментариев и т.д. В терминологии IDA такая память именуется **DATEBASE\_MEMORY** или **Memory for b-tree** – память базы данных двоичного дерева. Поле "DATEBASE\_MEMORY" конфигурационного файла <ida.cfg> позволяет изменять выделенное количество памяти под буфер базы данных. Объем памяти измеряется в байтах, но округляется до целого числа страниц, размер которых в текущих версиях IDA равен 8 килобайт (8.192 байт). Для нормальной работы дизассемблеру необходимо по крайней мере 5 страниц (40 килобайт), в противном случае IDA сообщит о нехватке памяти «*bTree error: not enough memory*» и аварийно завершит работу.

Если DATEBASE\_MEMORY = 0, IDA самостоятельно определяет оптимальный размер буфера по следующему алгоритму (см. таблицу 3):

Размер файла	Размер окна в страницах	Размер окна в байтах
0 – 256 КБ	5	256 КБ
256 КБ – 1 МБ	128	1 МБ
1 МБ – 2.5 МБ	128 – 320	1 МБ – 2.5 МБ
2.5 МБ – 5 МБ	512	4 МБ
> 5 МБ	$\text{FILESIZE} / 20 / \text{PAGESIZE}$	$\text{FILESIZE} / 20$

**Таблица 3 Алгоритм автоматического определения размера окна**

С целью увеличения производительности, IDA динамически создает список указателей на имена меток, для работы с которым так же требуется некоторое количество страничной памяти. Размер буфера в страницах задается значением поля **NPAGES** файла



<ida.cfg>, а размер страницы значением поля NPAGESIZE. По умолчанию IDA резервирует 64 страницы, объемом 1024 байт (1 КБ). Каждый указатель занимает 4 байта страничной памяти, следовательно, 64-кб буфер вмещает свыше 16 тысяч имен. Следует отметить, при дизассемблировании программ, написанных на Delphi, IDA генерирует огромное количество имен и увеличение значение поля NPAGES может существенно улучшить производительность.

При загрузке файла в окне сообщений выдается отчет о выделении страничной памяти под нужды IDA: окно виртуальной памяти (*"allocating memory for virtual array"*), буфер двоичного дерева (*"allocating memory for b-tree"*) и буфер указателей на имена (*"allocating memory for name pointers"*).

Например, выделение памяти при загрузке файла "first.exe" в IDA 3.84 происходит следующим образом:

```

bytes  pages size description
-----
262144   32 8192 allocating memory for b-tree...
65536   16 4096 allocating memory for virtual array...
65536   64 1024 allocating memory for name pointers...
-----

```

**Рисунок 15 "Отчет о выделении памяти при загрузке IDA"**

### Взаимодействие с физической памятью

Взаимодействие с физической памятью становится возможным благодаря наличию четырех недокументированных функций `_peek`, `_poke`, `_lpoke` и `_call`, прототипы которых приведены ниже:

- `long _poke(long ea, long value)`
- `long _lpoke(long ea, long value)`
- `long _peek(long ea, long value)`
- `long _call(long ea)`

Функции `_poke` и `_lpoke` записывают байт и длинное целое **value** по линейному адресу **ea** физической памяти соответственно, возвращая прежнее значение ячейки. Функция `_peek` читает байт по линейному адресу **ea** физической памяти, а функция `_call` передает управление на машинный код, расположенный по линейному адресу **ea** физической памяти.

Следует отметить, указанные функции по-разному функционируют в различных версиях IDA и результат их выполнения может быть непредсказуем, поэтому, их использование не рекомендуется.

Пример, приведенный ниже, демонстрирует копирование содержимое ПЗУ компьютера в виртуальную память дизассемблера, с последующим анализом обработчика прерывания INT 0x13. Ввиду различной реализации функций низкоуровневой работы с памятью, его успешная работа гарантируется лишь при запуске из MS-DOS-версии IDA Pro. Операционная система Windows 9x эмулирует наличие ПЗУ и позволяет функции `_peek` обращаться к нему даже из 32-разрядных версий IDA Pro.

```

auto a;
SegCreate(0xF0000, 0xFFFFF, 0x0F000, 0, 0, 0);
Message("Ждите... читаю BIOS...");
for (a=0; a<0xFFFF; a++)
    PatchByte(0xF0000+a, _peek(0xF0000+a));
Message("OK \n Дизассемблирую обработчик Int 0x13");
MakeCode(0xFEC59);

```

```
Message ("OK \n");
Jump (0xFEC59);
```

По окончании работы скрипта экран дизассемблера должен выглядеть так:

```
seg001:EC59 loc_E000_EC59: ; CODE XREF: seg001:0188↑J
seg001:EC59 ; seg001:019B↑J
seg001:EC59 jmp loc_E000_EE3B
seg001:EC5C ; -----
seg001:EC5C mov al, 0C0h ; 'L'
seg001:EC5E call sub_E000_EC8E
seg001:EC61 jmp short loc_E000_EC66
seg001:EC63 ; -----
seg001:EC63 call loc_E000_EC8A
seg001:EC66
seg001:EC66 loc_E000_EC66: ; CODE XREF: seg001:EC61↑J
seg001:EC66 call sub_E000_EC6C
seg001:EC69 cmp al, ah
seg001:EC6B retn
```

### Навигатор по функциям

Начать изучение виртуальной памяти лучше всего с загрузки двоичного (бинарного) файла. Сначала необходимо создать сам файл – это можно сделать командой “echo Hello, IDA Pro! > tutor.bin”.

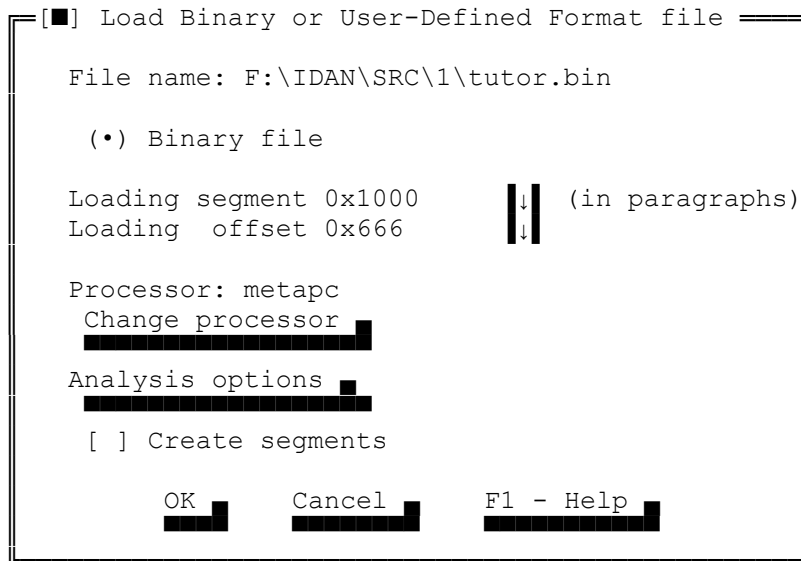


Рисунок 16 Диалог загрузки бинарного файла

Дождавшись появления диалога загрузки консольной версии IDA Pro, запомните базовый адрес сегмента, записанный в поле “Loading segment ... (in paragraphs)”, измените адрес загрузки на любой отличный от нуля (нулевое смещение крайне ненаглядно для иллюстрации), скажем, на 0x666, и сбросьте флажок “Create segment” – для предотвращения автоматического создания сегмента (работа с сегментами будет рассмотрена позже, в главе «Сегменты и селекторы»).

**Замечание:** доступные автору графические версии IDA Pro содержат ошибку реализации, всегда создавая новый сегмент при загрузке файла, независимо от установленных настроек.

После окончания загрузки файла экран дизассемблера должен выглядеть следующим образом:

```
0:00010666 ; File Name   : F:\IDAN\SRC\1\tutor.bin
0:00010666 ; Format      : Binary File
0:00010666 ; Base Address: 1000h Range: 10666h - 10678h Loaded length: 0012h
0:00010666
0:00010666
0:00010666      db 48h ; H
0:00010667      db 65h ; e
0:00010668      db 6Ch ; l
0:00010669      db 6Ch ; l
0:0001066A      db 6Fh ; o
0:0001066B      db 2Ch ; ,
0:0001066C      db 20h ;
0:0001066D      db 49h ; I
0:0001066E      db 44h ; D
0:0001066F      db 41h ; A
0:00010670      db 20h ;
0:00010671      db 50h ; P
0:00010672      db 72h ; r
0:00010673      db 6Fh ; o
0:00010674      db 21h ; !
0:00010675      db 20h ;
0:00010676      db 0Dh ;
0:00010677      db 0Ah ;
0:00010677
0:00010677      end
```

Слева каждой строки указывается ее линейный адрес, причем адрес первого байта равен  $0x1000*0x10+0x666$ , т.е. сумме базового адреса, указанного при загрузке, умноженного на шестнадцать и адреса смещения.

Чтение ячеек виртуальной памяти осуществляется функциями – **Byte(long ea)**, **Word(long ea)** и **Dword(long ea)** – возвращающими байт, слово и двойное слово соответственно. Если запрошенная ячейка не существует или не инициализирована, функция возвращает 0xFF (при этом следует быть особенно осторожным с функциями Word и Dword, некорректно сигнализирующих об ошибке – подробнее об этом можно прочитать в под главах Word и Dword соответственно).

Поэтому, перед чтением ячейки памяти следует убедиться, что она есть и содержит какое-то значение. Это можно осуществить анализом младшего бита поля атрибутов – если он сброшен – ячейка отсутствует или не инициализирована. Получить содержимое поля атрибутов можно вызовом функции **GetFlags** (см. описание функции GetFlags) следующим образом:

```
if (MS_VAL & GetFlags(ea))
    // значение ячейки определено, можно читать;
else
    // значение ячейки не определено либо ячейка не существует;
```

...или же воспользоваться макросом **hasValue(F)**, определенным в `<idc.idc>`, который следует вызывать так:

```
if (hasValue(GetFlags(ea)))
    // значение ячейки определено, можно читать;
else
    // значение ячейки не определено либо ячейка не существует
```

Более короткий путь предоставляет макрос **isLoaded(ea)**, определенный там же с аналогичным назначением:

```

if(isLoaded(ea))
    // значение ячейки определено, можно читать;
else
    // значение ячейки не определено либо ячейка не существует"

```

**Замечание:** макрос **byteValue(F)**, определенный в файле <idc.idc>, при правильном употреблении позволяет сократить количество вызовов **GetFlags**, а, следовательно, увеличить производительность программы. Вызывать его следует так:

```

F = GetFlags(ea);
if (hasValue(F)) val = byteValue(F);

```

Использование функции **Byte** обычно требует двух вызовов **GetFlags** – один раз для проверки значения ячейки, второй – для ее чтения. Если же ячейка заведомо существует и гарантировано содержит инициализированное значение, проверку можно опустить – в этом случае макрос **byteValue** не будет иметь никаких преимуществ перед функцией **Byte**.

Пример использования:

```

auto a;
Message(">");
for (a=0x10666;a<0x10676;a++)
    if (isLoaded(a)) Message("%c",Byte(a));
    else Message("!ОШИБКА!\n");

>Hello, IDA Pro!

```

Модификация ячеек виртуальной памяти осуществляется функциями **PatchByte (long ea, long value)**, **PatchWord (long ea, long value)** и **PatchDword (long ea, long value)** записывающих байт, слово и двойное слово соответственно.

Попытка модификации несуществующей ячейки виртуальной памяти заканчивается провалом.

Следующий пример меняет регистр всех символов на противоположный:

```

0:00010666      db 48h ; H
0:00010667      db 65h ; e
0:00010668      db 6Ch ; l
0:00010669      db 6Ch ; l
0:0001066A      db 6Fh ; o
0:0001066B      db 2Ch ; ,
0:0001066C      db 20h ;
0:0001066D      db 49h ; I
0:0001066E      db 44h ; D
0:0001066F      db 41h ; A
0:00010670      db 20h ;
0:00010671      db 50h ; P
0:00010672      db 72h ; r
0:00010673      db 6Fh ; o
0:00010674      db 21h ; !
0:00010675      db 20h ;
0:00010676      db 0Dh ;
0:00010677      db 0Ah ;

```

а) исходные данные – требуется заменить регистр всех символов на противоположный

```

auto a;
for (a=0x10666;a<0x10674;a++)

```

```
PatchByte(a, Byte(a) ^ 0x20);
```

**b) скрипт, меняющий регистр всех символов на противоположный, посредством вызова функции PatchByte**

```
0:00010666      db 68h ; h
0:00010667      db 45h ; E
0:00010668      db 4Ch ; L
0:00010669      db 4Ch ; L
0:0001066A      db 4Fh ; O
0:0001066B      db 0Ch ;
0:0001066C      db 0 ;
0:0001066D      db 69h ; i
0:0001066E      db 64h ; d
0:0001066F      db 61h ; a
0:00010670      db 0 ;
0:00010671      db 70h ; p
0:00010672      db 52h ; R
0:00010673      db 4Fh ; O
0:00010674      db 21h ; !
0:00010675      db 20h ;
0:00010676      db 0Dh ;
0:00010677      db 0FFh ;
```

**c) результат – регистр всех символов изменен на противоположный**

Пара функций **NextAddr(long ea)** и **PrevAddr(long ea)** позволяют трассировать адресное пространство виртуальной памяти, «проходясь» по цепочке «включенных» адресов.

Функция **NextAddr(long ea)** возвращает первый «включенный» адрес, следующий за «ea», соответственно **PrevAddr(long ea)** возвращает первый «включенный» адрес, предшествующий «ea».

Пример использования:

```
auto a;
a=0;
while(1)
{
    a=NextAddr(a);
    if (a==BADADDR) break;
    Message("0:%08X\t db  %x;", a, Byte(a));
    if (Byte(a)>0x20)
        Message("' %c'", Byte(a));
    Message("\n");
}
```

**a) скрипт, демонстрирующий трассировку адресов**

```
0:00010666 db 68; 'h'
0:00010667 db 45; 'E'
0:00010668 db 4c; 'L'
0:00010669 db 4c; 'L'
0:0001066A db 4f; 'O'
0:0001066B db c;
0:0001066C db 0;
0:0001066D db 69; 'i'
0:0001066E db 64; 'd'
0:0001066F db 61; 'a'
0:00010670 db 0;
0:00010671 db 70; 'p'
0:00010672 db 52; 'R'
0:00010673 db 4f; 'O'
0:00010674 db 21; '!'
```

```

0:00010675oddb 20;
0:00010676oddb d;
0:00010677oddb ff; ' '

```

b) результат – отображены только существующие адреса

## Сводная таблица функций

??? #Верстальщику ChangeTable

<b>функции возвращающие значение ячейки виртуальной памяти</b>	
имя функции	краткое описание
long Byte(long ea)	возвращает содержимое ячейки виртуальной памяти, расположенной по адресу ea
long Word(long ea)	возвращает содержимое ячеек виртуальной памяти, расположенных по адресам ea и ea+1, располагая их в младшем и старшем байте машинного слова соответственно.
long Dword(long ea)	возвращает содержимое ячеек виртуальной памяти, расположенных по адресам ea, ea+1, ea+2 и ea+3, располагая их в младших и старших байтах младшего и старшего слова соответственно
<b>функции модифицирующие значение ячейки виртуальной памяти</b>	
имя функции	краткое описание
void PatchByte(long ea, long value)	записывает в ячейку виртуальной памяти, расположенную по адресу ea, значение value
void PatchWord(long ea, long value)	записывает в ячейки виртуальной памяти, расположенные по адресам ea и ea+1, младший и старший байт значения value соответственно
void PatchDword(long ea, long value)	записывает в ячейки виртуальной памяти, расположенные по адресам ea, ea+1, ea+2 и ea+3 младшие и старшие байты младшего и старшего слова соответственно
<b>функции трассирующие адреса виртуальной памяти</b>	
имя функции	краткое описание
long NextAddr(long ea)	возвращает следующий линейный адрес, если он существует, в противном случае - ошибку
long PrevAddr(long ea)	возвращает предыдущий линейный адрес, если он существует, в противном случае – ошибку
<b>функции поиска</b>	
имя функции	краткое описание
long FindBinary(long ea, long flag, char str)	
<b>функции, манипулирующие с флагами</b>	
имя функции	краткое описание
long GetFlags (long ea)	возвращает значение флагов виртуальной памяти
long SetFlags(long ea, long flags)	задает новые значения флагов виртуальной памяти

long Byte (long ea)

Функция возвращает значение байта виртуальной памяти, расположенного по

адресу **ea**. Если указанный адрес не существует, функция возвращает 0xFF, сигнализируя об ошибке, такое же значение возвратится и в том случае если ячейка имеет не инициализированное значение.

Поэтому, до вызова функции `Byte` следует убедиться, что ячейка действительно существует и имеет определенное значение. Проверить это можно, проанализировав младший бит поля атрибутов, определенный в файле `<idc.idc>` константой `FF_INV`, - если он не равен нулю, то все о'кей:

```
if (FF_INV & GetFlags(ea)) value=Byte(ea);
    else // ячейка не существует или имеет неинициализированное значение
```

В файле `<idc.idc>` определены два макроса `hasValue(F)` и `isLoaded(ea)`, выполняющие такие проверки. Макрос `hasValue(F)` ожидает флаг виртуальной памяти, а `isLoaded(ea)` линейный адрес ячейки, т.е.:

```
if (hasValue(GetFlags(ea)) value=Byte(ea);
    else //ячейка не существует или имеет неинициализированное значение

if (isLoaded(ea)) value=Byte(ea);
    else //ячейка не существует или имеет неинициализированное значение
```

Альтернативой функции `Byte` служит вызов `GetFlags` с последующей маскировкой старших 24-бит, содержащих поле атрибутов. Маской может служить определенная в файле `<idc.idc>` константа `MS_VAL` или непосредственное значение – 0xFF. Это может выглядеть так: `value = (MS_VAL & GetFlags(ea))`.

Для увеличения производительности скрипта можно использовать макрос `byteValue(F)`, определенный в файле `<idc.idc>`, передавая ему значение флагов, ранее полученное для выполнения проверки существования ячейки: `"F=GetFlags(ea); if (hasValue(F)) value=byteValue(F);"` – это позволяет избавиться от вызова функции `Byte`, экономя тем самым некоторое количество процессорного времени.

**Замечание:** *если читаемые ячейки заведомо существуют и гарантированно содержат инициализированные значения, в дополнительных проверках никакой необходимости нет и использование макроса `byteValue` будет ничуть не быстрее вызова функции `Byte`*

Пример использования:

```
0:00010000      db  48h ; H
0:00010001      db  65h ; e
0:00010002      db  6Ch ; l
0:00010003      db  6Ch ; l
0:00010004      db  6Fh ; o
0:00010005      db  2Ch ; ,
0:00010006      db  20h ;
0:00010007      db  49h ; I
0:00010008      db  44h ; D
0:00010009      db  41h ; A
0:0001000A      db  20h ;
0:0001000B      db  50h ; P
0:0001000C      db  72h ; r
0:0001000D      db  6Fh ; o
0:0001000E      db  21h ; !
0:0001000F      db  20h ;
0:00010010      db  0Dh ;
0:00010011      db  0Ah ;
```

а) исходные данные – требуется вывести на консоль содержимое отображаемых

## ячеек памяти

```
auto a;
Message(">");
for (a=0x10000;a<0x10011;a++)
Message("%c",Byte(a));
Message("\n");
```

b) последовательный вызов Byte для каждой ячейки

> Hello, IDA Pro!

с) результат

Другие примеры использования функции Byte можно найти в главе «Первые шаги с IDA Pro» и в файле "memcru.idc", входящим в комплект поставки IDA.

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес ячейки виртуальной памяти	
return	=return	пояснения
	==	содержимое байта ячейки виртуальной памяти
	==0xFF	ошибка

**Родственные функции:** Word, Dword

**Интерактивный аналог:** *mem*

## long Word (long ea)

Функция возвращает значение слова (одно слово равно двум байтам) виртуальной памяти, расположенного по адресу *ea*. При попытке чтения *байта*, расположенного по несуществующему адресу, равно как и имеющего неопределенное значение, функция возвращает значение 0xFF, сигнализируя об ошибке. Наглядно продемонстрировать работу функции позволяет следующий пример:

```
0:00010000      db  48h ; H
0:00010001      db  65h ; e
0:00010002      db  6Ch ; l
0:00010003      db  6Ch ; l
0:00010004      db  6Fh ; o
0:00010005      db  2Ch ; ,
0:00010006      db  20h ;
0:00010007      db  49h ; I
0:00010008      db  44h ; D
0:00010009      db  41h ; A
0:0001000A      db  20h ;
0:0001000B      db  50h ; P
0:0001000C      db  72h ; r
0:0001000D      db  6Fh ; o
0:0001000E      db  21h ; !
0:0001000F      db  20h ;
0:00010010      db  0Dh ;
0:00010011      db  0Ah ;
```

```
Message(">%X\n", Word (0x10000));
```



>6548

```
Message(">%X\n", Word (0x0));  
>FFFF
```

```
Message(">%X\n", Word (0x10011));  
>FF0A
```

```
Message(">%X\n", Word (0xFFFF));  
>48FF
```

В первом случае существуют оба адреса (т.е. 0x10000 и 0x10001) и функция обрабатывает успешно; во втором – ни существует ни одного из них – ни 0x0, ни 0x1, в результате чего возвращается 0xFFFF.

Но попытка прочитать слово, расположенное по адресу 0x10011, приводит к тому, что в младшем байте возвращается значение соответствующей ячейки, а в старшем – 0xFF! Аналогично и в последнем примере – несуществующий младший байт дает 0xFF, в то время как старший читается успешно.

Поэтому, до вызова функции Word следует убедиться, что обе ячейка действительно существуют и имеют определенные значение. Проверить это можно, проанализировав младший бит поля атрибутов каждой из ячеек – если он не равен нулю, то все о'кей. О том как это сделать подробно рассказано в описании функции Byte.

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес ячейки виртуальной памяти	
return	==return	пояснения
	==	содержимое слова виртуальной памяти
	==FF?? =??FF	ошибка

**Родственные функции:** Byte, Dword.

**Интерактивный аналог:** *нет*

### long Dword (long ea)

Функция возвращает значение двойного слова виртуальной памяти по указанному адресу.

В остальном она полностью аналогична функции Word.

??? #верстальщику – change table

аргумент	пояснение	
ea	линейный адрес ячейки виртуальной памяти	
return	==return	Пояснения
		содержимое двойного слова виртуальной памяти
	==(FF)	ошибка

**Родственные функции:** Byte, Word

**Интерактивный аналог:** *нет*

void PatchByte (long ea, long value)

Функция модифицирует содержимое байта виртуальной памяти, расположенного по линейному адресу **ea**, на значение **value**.

По замыслу разработчика предназначалась для *ладченья* программы (например, замене 7х на EB, т.е. инструкций условного перехода на безусловный переход – операция часто сопутствующая снятию защит), чем и объясняется ее название. Однако, она нашла применение в решении широкого круга различных задач, в частности копировании фрагментов виртуальной памяти.

Функция не позволяет модифицировать не существующие ячейки памяти и не сигнализирует об ошибках записи, поэтому, перед ее вызовом рекомендуется проверить передаваемый ей линейный адрес на существование вызовом GetFlags (подробнее об этом рассказывается в описании функции Byte).

Пример ее использования можно найти в файле "memcpu.idc", поставляемом вместе с IDA.

??? #верстальщику – change table

аргумент	пояснение
ea	линейный адрес ячейки виртуальной памяти
value	Записываемое значение (байт)

**Родственные функции:** PatchWord, PatchDword

**Интерактивный аналог:** «~EDIT\Patch program\Change byte»

void PatchWord (long ea, long value)

Функция модифицирует содержимое слова виртуальной памяти, расположенного по адресу **ea** на значение **value**. В остальном аналогична PatchByte (см. описание PatchByte).

??? #верстальщику – change table

аргумент	пояснения
ea	линейный адрес ячейки виртуальной памяти
value	Записываемое значение (слово)

**Родственные функции:** PatchByte, PatchDword

**Интерактивный аналог:** «~EDIT\Patch program\Change word»

void PatchDword (long ea, long value)

Функция модифицирует содержимое двойного слова виртуальной памяти, расположенного по адресу **ea** на значение **value**. В остальном аналогична PatchByte (см. описание PatchByte)

??? #верстальщику – change table

аргумент	пояснения
----------	-----------

ea	линейный адрес ячейки виртуальной памяти
value	Записываемое значение (слово)

**Родственные функции:** PatchByte, PatchWord

**Интерактивный аналог:** *нет*

### long NextAddr (long ea)

Функция возвращает следующий существующий виртуальный адрес, и BADADDR в том случае, если такого адреса не существует. Вызов NextAddr (BADADDR) равносильен NextAddr (0x0).

Пример использования:

```

0:00010000          db  48h ; H
0:00010001          db  65h ; e
0:00010002          db  6Ch ; l
0:00010003          db  6Ch ; l
0:00010004          db  6Fh ; o
0:00010005          db  2Ch ; ,
0:00010006          db  20h ;
0:00010007          db  49h ; I
0:00010008          db  44h ; D
0:00010009          db  41h ; A
0:0001000A          db  20h ;
0:0001000B          db  50h ; P
0:0001000C          db  72h ; r
0:0001000D          db  6Fh ; o
0:0001000E          db  21h ; !
0:0001000F          db  20h ;
0:00010010          db  0Dh ;
0:00010011          db  0Ah ;

```

**a) исходные данные – требуется получить список адресов виртуальной памяти**

```

auto a;
a=0;
while(1)
{
    a=NextAddr(a);
    if (a==BADADDR) break;
    Message(">%x\n", a);
}

```

**b) трассировка адресов последовательными вызовами функции NextAddr**

```

>10000
>10001
>10002
>10003
>10004
>10005
>10006
>10007
>10008
>10009
>1000a

```

```
>1000b
>1000c
>1000d
>1000e
>1000f
>10010
>10011
```

с) результат – получение перечня существующих адресов виртуальной памяти

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес ячейки виртуальной памяти	
return	=return	пояснения
	!=BADADDR	следующий за ea адрес виртуальной памяти
	==BADADDR	ошибка

**Родственные функции:** PrevAddr

**Интерактивный аналог:** *нет*

### long PrevAddr (long ea)

Функция возвращает предшествующий **ea** существующий виртуальный адрес, и BADADDR в том случае, если такого адреса не существует.

Пример использования:

```
0:00010000      db  48h ; H
0:00010001      db  65h ; e
0:00010002      db  6Ch ; l
0:00010003      db  6Ch ; l
0:00010004      db  6Fh ; o
0:00010005      db  2Ch ; ,
0:00010006      db  20h ;
0:00010007      db  49h ; I
0:00010008      db  44h ; D
0:00010009      db  41h ; A
0:0001000A      db  20h ;
0:0001000B      db  50h ; P
0:0001000C      db  72h ; r
0:0001000D      db  6Fh ; o
0:0001000E      db  21h ; !
0:0001000F      db  20h ;
0:00010010      db  0Dh ;
0:00010011      db  0Ah ;
```

а) исходные данные – требуется получить список адресов виртуальной памяти

```
auto a;
a=BADADDR;
while(1)
{
    a=PrevAddr(a);
    if (a==BADADDR) break;
    Message(">%X\n", a);
}
```

}

b) трассировка адресов последовательными вызовами функции PrevAddr

>10011  
>10010  
>1000F  
>1000E  
>1000D  
>1000C  
>1000B  
>1000A  
>10009  
>10008  
>10007  
>10006  
>10005  
>10004  
>10003  
>10002  
>10001  
>10000

c) результат – получение перечня существующих адресов виртуальной памяти

??? #верстальщику – change table

аргумент	пояснение	
ea	линейный адрес ячейки виртуальной памяти	
return	=return	пояснения
	!=BADADDR	предшествующий ea адрес виртуальной памяти
	==BADADDR	ошибка

**Родственные функции:** NextAddr

**Интерактивный аналог:** *нет*

long GetFlags(long ea)

Функция возвращает значение флагов виртуальной памяти, ассоциированных с виртуальным адресом **ea**. Если указанного виртуального адреса не существует, функция возвращает ноль.

О назначении каждого бита флагов рассказано при описании связанных с ним функций.

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес ячейки виртуальной памяти	
return	=return	Пояснения
	!=0	значение флагов
	==0	ошибка

**Родственные функции:** SetFlags

**Интерактивный аналог:** *нет*

## void SetFlags(long ea)

Функция задает новое значение флагов виртуальной памяти, ассоциированных с виртуальным адресом **ea**. Допустимо модифицировать флаги лишь существующей ячейки виртуальной памяти.

**Внимание:** *настоятельно рекомендуется по возможности избегать непосредственной модификации флагов – допущенная ошибка может привести к зависанию дизассемблера!*

??? #верстальщику – change table

аргумент	пояснения
ea	линейный адрес ячейки виртуальной памяти

**Родственные функции:** GetFlags

**Интерактивный аналог:** *нет*

## long FindBinary(long ea,long flag,char str)

Функция ищет заданную подстроку в виртуальной памяти и в случае успешного поиска возвращает ее линейный адрес, иначе возвращает значение BADADDR, сигнализируя об ошибке.

В зависимости от флага направления поиск может идти как вперед (от младших адресов к старшим), так и назад (от старших адресов к младшим), регистр символов может как различаться, так и нет.

Аргумент **ea** задает линейный адрес начала поиска и не обязательно должен существовать.

Аргумент **str** задает подстроку поиска, выраженную в шестнадцатеричных кодах символов (точнее – в системе исчисления, установленной системой исчисления по умолчанию), разделенных между собой пробелами. Суффикс “h”, равно как и префикс “x” при этом указывать не нужно.

Аргумент **flag** задает направление поиска и определяет чувствительность к регистру символов: если его младший бит установлен поиск идет от младших адресов к старшим и, соответственно, наоборот; если первый справа бит (считая от нуля) установлен – прописные и строчечные буквы различаются и, соответственно, наоборот.

Пример использования:

```
seg000:0000      db  48h ; H
seg000:0001      db  65h ; e
seg000:0002      db  6Ch ; l
seg000:0003      db  6Ch ; l
seg000:0004      db  6Fh ; o
seg000:0005      db  2Ch ; ,
seg000:0006      db  20h ;
seg000:0007      db  49h ; I
seg000:0008      db  44h ; D
seg000:0009      db  41h ; A
seg000:000A      db  20h ;
seg000:000B      db  50h ; P
seg000:000C      db  72h ; r
seg000:000D      db  6Fh ; o
seg000:000E      db  21h ; !
seg000:000F      db   0 ;
```

```
Message(">%s\n", atoa (FindBinary ( SegByName ("seg000"), 1, "49 44
41" ) ) );
```

??? #верстальщику – change table

аргумент	пояснения		
ea	линейный адрес начала поиска		
flag	=flag		
	бит	#	
	0	0	поиск от старших адресов к младшим
		1	поиск от младших адресов к старшим
	1	0	не различать регистр символов
1		различать регистр символов	
return	=return		
	Пояснения		
	!=BADADDR	линейный адрес найденной подстроки	
	==BADADDR	ошибка	

Родственные функции: *нет*

Интерактивный аналог: “~Search\Text”, <Alt-T>

## СЕГМЕНТЫ И СЕЛЕКТОРЫ

### #Definition

**Сегментом** называется непрерывная область памяти, адресуемая относительно базового адреса сегмента.

Каждый сегмент характеризуется **базовым адресом** сегмента, **адресом начала** сегмента и **адресом конца** сегмента.

Базовый адрес сегмента обычно выражается в параграфах, адреса начала и конца - в байтах.

Адрес начала сегмента задает наименьший адрес, принадлежащей сегменту; адрес конца сегмента – адрес, на единицу больше превышающий наибольший адрес, принадлежащий сегменту.

Никакой линейный адрес не может принадлежать более чем одному сегменту одновременно – т.е. сегменты не могут пересекаться.

В дальнейшем, если не оговорено обратное, адрес начала сегмента обозначается “startea”, адрес конца сегмента – “endea”, а базовый адрес – “BASE”.

Смещение первого байта в сегменте обозначается “startoffset” и связано с адресом начала и базовым адресом следующим соотношением:

$$\text{startoffset} = \text{startea} - \text{BASE} * 0x10$$

### Формула 1 Смещение первого байта в сегменте

Смещения в сегменте измеряются целыми неотрицательными числами, следовательно, приравняв startoffset к нулю, получаем:  $\text{startea} \geq (\text{BASE} * 0x10)$ .

Сегментный адрес [BASE:offset] связан с линейным адресом следующим соотношением:

$$ea = \text{BASE} * 0x10 + \text{offset}$$

## Формула 2 Перевод сегментного адреса в линейный

подавляющее большинство функций IDA работают не с сегментными, а с линейными адресами. Пользователь же, напротив, видит на экране дизассемблера в основном сегментные адреса, а линейные от его взора скрыты.

Для облегчения преобразования сегментных адресов в линейные предусмотрен специальный макрос **MK\_FP(long BASE, long offset)**, возвращающий значение  $\text{BASE} * 0x10 + \text{offset}$ , а так же оператор «квадратные скобки» - “[BASE, offset]” аналогичного назначения.

Линейные адреса начала и конца сегмента представляют собой 32-битовые значения, ограничивающие максимальный размер сегмента четырьмя гигабайтами.

Базовый адрес представляет собой 16-битовое значение, ограничивающее адресуемую память одним мегабайтом, которого в ряде случаев оказывается недостаточно.

**Замечание:** размер сегмента ограничен разрядностью линейных адресов его начала и конца и составляет 4 гигабайта, но выбор адреса начала сегмента, первый байт которого имеет нулевое смещение, ограничен разрядностью базового адреса, и равен  $\text{BASE}_{\text{max}} * 0x10 = 0xFFFF * 0x10 = 0xFFFF0$ , т.е. немногим менее одного мегабайта.

Выход состоит в использовании **селекторов**, ссылающихся на 32-разрядные адреса, что позволяет адресовать с их помощью до 4 гигабайт.

К селектору можно обратиться по его **индексу** в таблице селекторов. Индексы представляют собой 16-разрядные целые значения, увеличивающиеся с каждым очередным элементом на единицу.

Элементы таблицы – 32-разрядные базовые адреса сегмента, измеряемые в параграфах.

Таблица селекторов представляет собой разряженный массив, допуская создание элементов с несмежными индексами. Например, 0x5,0x07,0x16,0x88...

Если при создании сегмента в качестве базового адреса указать индекс созданного ранее селектора, его значение будет автоматически использовано для базирования данного сегмента. Аналогично, если создать селектор, совпадающий с базовым адресом некоторого сегмента, для его базирования станет использоваться значение селектора, а не базовый адрес.

С каждым сегментом связан ряд атрибутов – имя сегмента, кратность выравнивания, разрядность и объединение. Никакой из атрибутов, включая имя, уникальной характеристикой сегмента не является и вполне допустимо существование двух и более сегментов с одинаковыми именами (однако, ассемблеры не смогут откомпилировать исходный тест, содержащий несколько одноименных сегментов).

**Замечание:** создание двух сегментов с одинаковыми базовыми адресами допускается, но пользоваться этой возможностью категорически не рекомендуется.

Над каждым сегментом можно выполнять следующие операции – создание и удаление сегментов, получение и изменение основных характеристик сегментов (линейного адреса начала, линейного адреса конца, базового адреса), получение и изменение атрибутов сегмента (имя, кратность выравнивания, разрядность и т.д.).



Подробнее об этом рассказано в главе «Функции, работающие с сегментами и селекторами».

## Навигатор по функциям

Для изучения организации сегментов и селекторов рекомендуется загрузить полученный в главе «Виртуальная память» файл "tutor.idb" в дизассемблер, при этом экран IDA должен выглядеть так:

```
0:00010000      db  48h ; H
0:00010001      db  65h ; e
0:00010002      db  6Ch ; l
0:00010003      db  6Ch ; l
0:00010004      db  6Fh ; o
0:00010005      db  2Ch ; ,
0:00010006      db  20h ;
0:00010007      db  49h ; I
0:00010008      db  44h ; D
0:00010009      db  41h ; A
0:0001000A      db  20h ;
0:0001000B      db  50h ; P
0:0001000C      db  72h ; r
0:0001000D      db  6Fh ; o
0:0001000E      db  21h ; !
0:0001000F      db  20h ;
0:00010010      db  0Dh ;
```

Для создания нового сегмента можно воспользоваться вызовом функции **SegCreate(long startea,long endea,long base,long use32,long align,long comb)**, последовательно передав ей адрес начала сегмента, адрес конца сегмента, базовый адрес сегмента, разрядность сегмента, кратность выравнивания и атрибуты (о трех последних аргументах подробно рассказано в описании функции SegCreate, сейчас их можно принять равными нулю), например, так: «SegCreate(0x10000, 0x10012, 0x1000, 0, 0, 0);»

```
seg000:0000 ; Segment type: Regular
seg000:0000 seg000      segment at 1000h private '' use16
seg000:0000      assume cs:seg000
seg000:0000      assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000      db  48h ; H
seg000:0001      db  65h ; e
seg000:0002      db  6Ch ; l
seg000:0003      db  6Ch ; l
seg000:0004      db  6Fh ; o
seg000:0005      db  2Ch ; ,
seg000:0006      db  20h ;
seg000:0007      db  49h ; I
seg000:0008      db  44h ; D
seg000:0009      db  41h ; A
seg000:000A      db  20h ;
seg000:000B      db  50h ; P
seg000:000C      db  72h ; r
seg000:000D      db  6Fh ; o
seg000:000E      db  21h ; !
seg000:000F      db  20h ;
seg000:0010      db  0Dh ;
seg000:0011      db  0Ah ;
seg000:0011 seg000      ends
```

Создав новый сегмент, IDA автоматически присвоила ему имя “seg000”, где “000” порядковый номер (считая от нуля) созданного сегмента. Последующие сегменты будут названы “seg001”, “seg002” и т.д.

Функция “**long SegByName(char segname)**” позволяет узнать линейный адрес базового адреса сегмента<sup>4</sup> по его имени. Ее вызов может выглядеть, например, так:

```
Message(">%X\n", SegByName("seg000"));
> 10000
```

Переименовать сегмент можно с помощью функции “**success SegRename(long ea, char name)**” принимающей в качестве первого аргумента любой линейный адрес, принадлежащий указанному сегменту, а вторым – его новое имя. Например:

```
SegRename(SegByName("seg000"), "MySeg");
```

```
MySeg:0000 ; Segment type: Regular
MySeg:0000 MySeg      segment at 1000h private '' usel6
MySeg:0000          assume cs:MySeg
MySeg:0000          assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
MySeg:0000          db 48h ; H
MySeg:0001          db 65h ; e
MySeg:0002          db 6Ch ; l
MySeg:0003          db 6Ch ; l
MySeg:0004          db 6Fh ; o
MySeg:0005          db 2Ch ; ,
MySeg:0006          db 20h ;
MySeg:0007          db 49h ; I
MySeg:0008          db 44h ; D
MySeg:0009          db 41h ; A
MySeg:000A          db 20h ;
MySeg:000B          db 50h ; P
MySeg:000C          db 72h ; r
MySeg:000D          db 6Fh ; o
MySeg:000E          db 21h ; !
MySeg:000F          db 20h ;
MySeg:0010          db 0Dh ;
MySeg:0011          db 0Ah ;
MySeg:0011 MySeg      ends
```

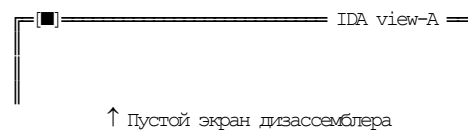
Для удаления сегментов предусмотрена функция “**success SegDelete (long ea, long disable)**” Если флаг “disable” равен нулю, будет удален лишь сам сегмент, а содержимое принадлежащих ему ячеек сохранится, в противном случае вместе с сегментом будут удалены и все принадлежащие ему адреса виртуальной памяти.

Сравните:

```
SegDelete(0x10000, 0);
```

```
0:00010000          db 48h ; H
0:00010001          db 65h ; e
0:00010002          db 6Ch ; l
0:00010003          db 6Ch ; l
0:00010004          db 6Fh ; o
0:00010005          db 2Ch ; ,
0:00010006          db 20h ;
0:00010007          db 49h ; I
0:00010008          db 44h ; D
0:00010009          db 41h ; A
0:0001000A          db 20h ;
0:0001000B          db 50h ; P
0:0001000C          db 72h ; r
0:0001000D          db 6Fh ; o
```

```
SegDelete(0x10000, 0);
```



<sup>4</sup> Т.е. функция возвращает базовый адрес, тут же умножая его на 0x10 для перевода в линейный.

```

0:0001000E          db  21h ; !
0:0001000F          db  20h ;
0:00010010          db  0Dh ;
0:00010011          db  0Ah ;

```

Описанные выше операции можно выполнять и интерактивно – с помощью горячих клавиш и системы меню. Для последующих экспериментов потребуется перезагрузить исследуемый файл “tutor.bin”, восстановив его с прилагаемого к книге диска, поскольку удаление виртуальной памяти необратимо.

Для интерактивного создания сегмента достаточно в меню “View” выбрать пункт “Segments” и дождавшись появления списка существующих сегментов (в данном случае – пустого), нажать клавишу <insert> и надлежащим образом заполнить соответствующие поля появившегося диалога. Если предварительно выделить некоторую область курсорными клавишами, удерживая <Shift>, IDA автоматически подставит линейные адреса ее начала и конца в поля адреса начала и адреса конца сегмента соответственно (см. рисунок 17)

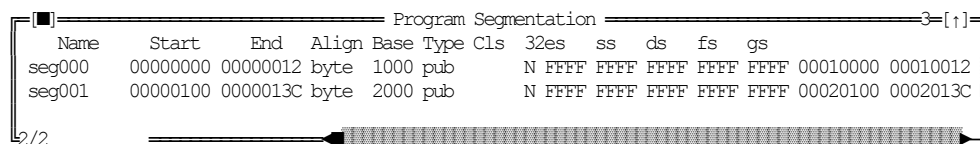
### Рисунок 17 “0x020” Создание сегмента по выделенной области

Если поле «Segment Name» оставить пустым, IDA присвоит имя сегменту самостоятельно.

По умолчанию базовый адрес равен наибольшему возможному значению, т.е.  $BASE_{def} = \frac{Startea}{0x10}$ , при этом  $offset_{def} = Startea \text{ AND } 0xF$ , где  $offset_{def}$  -- смещение первого байта в сегменте. Очевидно, что  $offset_{def} \leq 0xF$ .

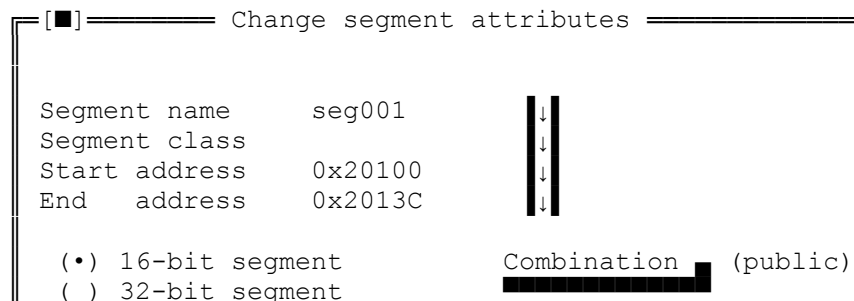
Например, если дозагрузить бинарный файл (~File\Load file\Additional binary file), скажем, *Crypt.com*, по виртуальному адресу 0x20100 и, выделив его, попытаться создать сегмент, IDA по умолчанию выберет базовый адрес, равный 0x2010, в результате чего смещение первого байта в сегменте будет равно 0x0, а не 0x100! Следует очень внимательно относиться к значениям, предлагаемым по умолчанию – они не всегда соответствуют требуемым.

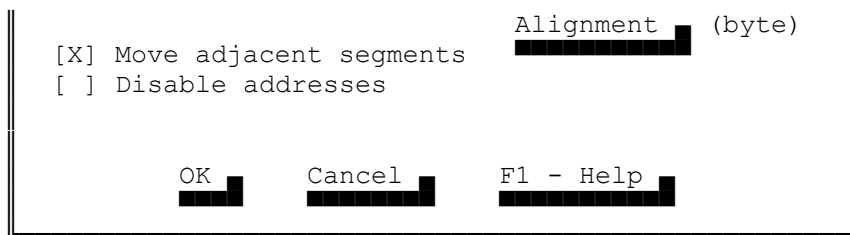
В действительности, базовый адрес должен быть равен 0x2000, тогда после создания сегмента экран дизассемблера будет выглядеть так:



### Рисунок 18

Для изменения свойств сегмента достаточно подвести к нему курсор и нажать клавишу <Ctrl-E>. Появится диалоговое окно следующего вида:

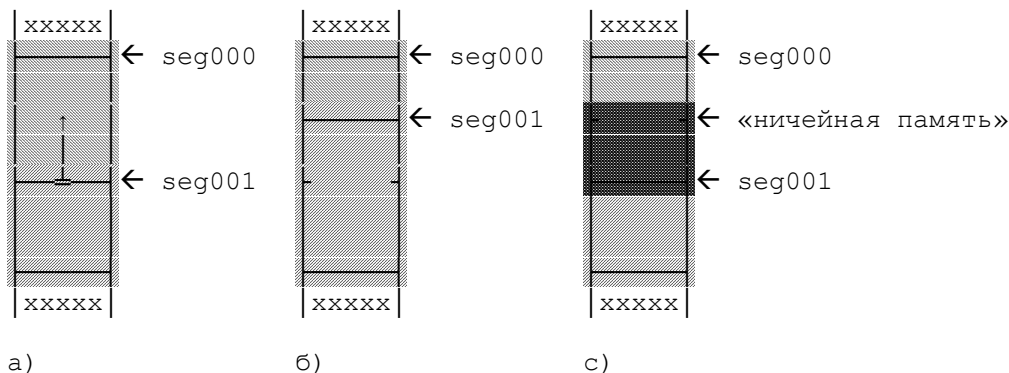




**Рисунок 19**

Интерактивно можно изменять все характеристики и атрибуты сегмента за исключением его базового адреса. На модификацию адресов начала и конца сегмента наложены некоторые ограничения:  $(BASE * 0x10) \leq Startea > Endea$ , т.е. адрес начала должен быть больше либо равен базовому адресу, умноженному на шестнадцать, а адрес конца сегмента должен быть по крайней мере на единицу превышать адрес начала.

При изменении границ одного из смежных сегментов, IDA автоматически расширит или сузит другой сегмент, если флаг “Move adjacent segments” установлен. Графически это можно изобразить так:



**Рисунок 20 Изменение границ сегментов**

??? #Художнику – перерисовать, сохраняя наклон штриховки. Крайняя левая картинка должна изображать как границу сегментов тянут вверх – можно нарисовать привязанную нитку и поднимающую ее руку.

На рисунке (10.a) показано уменьшение нижней границы сегмента seg000, к концу которого вплотную примыкает сегмент seg001. Если флажок “Move adjacent segment” взведен, IDA автоматически изменит адрес начала сегмента “seg001” как показано на рисунке 10.b; напротив сброс этого флажка приведет к тому, что между сегментами образуется дыра «ничейной памяти», изображенная на рисунке 10.c

Установка флажка “Disable addresses” автоматически уничтожит «дыру», удалив принадлежащие ей адреса памяти, вместе с их содержимым. **Внимание!** Эта операция необратима и вернуть утерянные данные обратно уже не удастся!

Для закрепления всего вышесказанного рекомендуется провести несколько простых экспериментов. Для начала попробуйте расширить границы сегмента “seg000” до адреса 0x20120 (или любого другого, принадлежащего сегменту “seg001”). Одним лишь изменением атрибутов сегмента “seg000” это сделать не удастся – IDA сообщит об ошибке “set\_seg\_m\_end(10000) -> 20120: areas overlap” и, независимо от состояния флага “Move adjacent segment”, прервет выполнение операции.

Причина в том, что автоматическое изменение границ работает с смежными и только с смежными сегментами, а “seg000” и “seg001” таковыми, очевидно, не являются,

поскольку адрес конца сегмента “seg000” равен 0x10012, а адрес начала “seg001” – 0x20100.

Выполнить поставленную задачу можно, по меньшей мере, двумя путями: *первое* – предварительно увеличить адрес начала сегмента “seg001” и повторить операцию; *второе* – увеличить адрес конца сегмента “seg000” до максимально возможного значения (т.е. 0x20100), добившись слияния сегментов, что даст возможность расширить “seg000” до требуемого адреса установкой флага “Move adjacent segment”.

*Последовательности нажатий клавиш для первого способа:* переместить курсор в пределы сегмента “seg001” и нажать <Alt-S>, в появившемся диалоговом окне изменить значение поля “Start address” на 0x20120; затем переместится в границы сегмента “seg000”, нажать <Alt-S> и изменить значение поля “End Address” на 0x20120”. Задача выполнена.

*Последовательности нажатий клавиш для второго способа:* переместить курсор в пределы сегмента “seg000” и нажать <Alt-S>, в появившемся диалоговом окне изменить значение поля “End address” на 0x20100, затем повторено вызвав тот же диалог, распахнуть сегмент до требуемого адреса, изменив значение поля “End address” на 0x20120. Задача выполнена.

Программно изменить атрибуты сегмента можно вызовами функций SegCreate, SegBounds, SegRename, SegClass, SegAlign, SegComb, SegAddrng

Функция **success SegBounds (long ea,long startea,long endea,long disable)** принимает в качестве первого аргумента любой адрес, принадлежащий сегменту, startea, endea – новые адреса начала и конца сегмента соответственно, ненулевое значение флага disable удаляет виртуальную память освободившуюся при уменьшении сегмента (при расширении сегмента его значение игнорируется). Следует помнить, что  $startea \geq \text{BEGIN\_ADDRES} * 0x10$ , т.е. верхнее расширение сегмента жестко ограничено его базовым адресом.

Например, для восстановления сегмента “seg000” до его прежних размеров можно воспользоваться следующим кодом: “SegBounds(0x10000,0x10000,0x10012,1);”

Если базовый адрес создаваемого сегмента больше или равен 0x10000, IDA автоматически создает селектор для его адресации.

Например, результат выполнения “SegCreate(0x100000,0x100100,0x10000,0,0,0);” приведет к следующему:

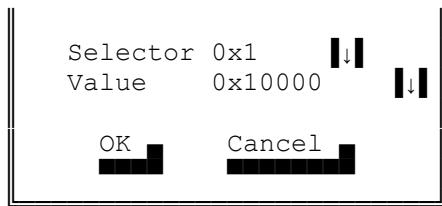
Name	Start	End	Align	Base	Type	Cls	32es	ss	ds	fs	gs
seg000	00000000	00000012	byte	1000	pub	CODE	N	FFFF	FFFF	FFFF	FFFF
seg001	00000100	0000013C	byte	2000	pub		N	FFFF	FFFF	FFFF	FFFF
seg002	00000000	00000100	at	0001	pub		N	FFFF	FFFF	FFFF	FFFF

Базовый адрес сегмента “seg002” равен 0x1, но не смотря на это, смещение первого байта в сегменте равно 0x0, а не  $0x100000 - 0x1 * 0x10 = 0xFFFFF0$ . Причина в том, что 0x1 – это селектор, а не сегмент. Чтобы убедиться в этом достаточно просмотреть список селекторов, вызвать который можно нажатием “<Alt-V>,<L>”:

Sel	Value
0001	00010000

В отличие от базового адреса, значение селектора легко изменить, для чего достаточно выделить его курсором и нажать <Ctrl-E>. Появится диалоговое окно следующего содержания:

Define a selector



Значение селектора может совпадать с базовым адресом любого из существующих сегментов, на него не действует ограничение  $startea \geq BASE\_ADDRES * 0x10$ , но если присвоить селектору значение численно равное его индексу, он будет немедленно удален.

Присвоение селектору базового адреса, превышающего адрес начала сегмента, приводит к появлению байт с отрицательными смещениями, которые автоматически дополняются до нуля, т.е.  $offset = NEG(|startea - SEL\_VALUE|)$ .

Например, после увеличения значения селектора 0x1 на один параграф экран дизассемблера будет выглядеть так:

```

seg002:FFFFFFF0 seg002      segment at 10001h private '' usel6
seg002:FFFFFFF0          assume cs:seg002
seg002:FFFFFFF0          ;org 0FFFFFFF0h
seg002:FFFFFFF0          assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg002:FFFFFFF0          db ? ; unexplored
seg002:FFFFFFF1          db ? ; unexplored
seg002:FFFFFFF2          db ? ; unexplored
seg002:FFFFFFF3          db ? ; unexplored
seg002:FFFFFFF4          db ? ; unexplored
seg002:FFFFFFF5          db ? ; unexplored
seg002:FFFFFFF6          db ? ; unexplored
seg002:FFFFFFF7          db ? ; unexplored
seg002:FFFFFFF8          db ? ; unexplored
seg002:FFFFFFF9          db ? ; unexplored
seg002:FFFFFFFA          db ? ; unexplored
seg002:FFFFFFFB          db ? ; unexplored
seg002:FFFFFFFC          db ? ; unexplored
seg002:FFFFFFFD          db ? ; unexplored
seg002:FFFFFFFE          db ? ; unexplored
seg002:FFFFFFF0          db ? ; unexplored
seg002:0000              db ? ; unexplored
seg002:0001              db ? ; unexplored
seg002:0002              db ? ; unexplored
seg002:0003              db ? ; unexplored
seg002:0004              db ? ; unexplored
seg002:0005              db ? ; unexplored
seg002:0006              db ? ; unexplored
seg002:0007              db ? ; unexplored

```

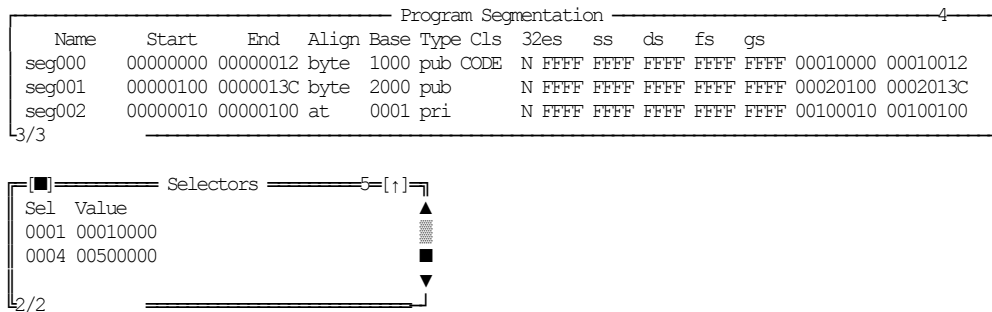
Исправить ситуацию можно соответствующим увеличением адреса начала сегмента. После вызова «SegBounds(0x100000,0x100010,0x100100,1);» экран дизассемблера должен выглядеть так:

Name	Start	End	Align	Base	Type	Cls	32es	ss	ds	fs	gs	
seg000	00000000	00000012	byte	1000	pub	CODE	N	FFFF	FFFF	FFFF	FFFF	00010000 00010012
seg001	00000100	0000013C	byte	2000	pub		N	FFFF	FFFF	FFFF	FFFF	00020100 0002013C
seg002	00000000	000000F0	at	0001	pub		N	FFFF	FFFF	FFFF	FFFF	00100010 00100100

Смещение первого байта в сегменте “seg002” равно нулю, чего и требовалось достичь.

Программно создать новый селектор или изменить значение уже существующего селектора можно с помощью функции “void SetSelector (long sel,long value)”, где sel – индекс селектора, а value – его значение, измеряемое в параграфах.

Например, вызов "SetSelector(0x1,0x10000);" изменит значение селектора 0x1 и смещение первого байта в сегменте "seg002" будет равно нулю, а вызов "SetSelector(0x4,0x500000);" создаст новый селектор с индексом 0x4 (индексы селекторов не обязательно должны следовать друг за другом).



Удалить селектор можно вызовом функции "void DelSelector (long sel)" либо присвоением ему собственного индекса – SetSelector (sel, sel).

Если удалить используемый селектор, то смещение первого байта в сегменте, который на него ссылался станет равным:  $offset = startea - sel * 0x10$ , где sel – индекс (не значение!) селектора.

Например, после вызова "SelDelete(0x1);" экран дизассемблера будет выглядеть так:

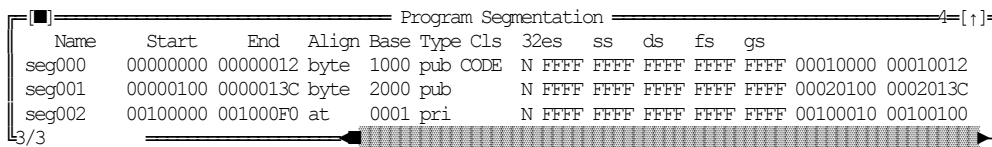


Рисунок 21

Операция удаления обратима и повторное создание уничтоженного селектора все вернет на свои места.

Функция **long FirstSeg()** возвращает линейный адрес начальный сегмента, имеющего наименьший адрес начала, соответственно функция **long NextSeg(long ea)** возвращает линейный адрес сегмента, следующего за ea.

Пример использования:

```

auto a;
a=FirstSeg();
while(a!=BADADDR)
{
Message(">%08x\n", a, SegName(a));
a=NextSeg(a);
}

>00010000
>00020100
>00100010

```

Зная адрес начала сегмента можно получить его имя вызовом функции SegName, а зная имя сегмента можно получить его базовый адрес при помощи функции SegByName; более быстрого способа вычисления базового адреса по-видимому не существует.

Передав функциям SegStart и SegEnd любой принадлежащий сегменту линейный адрес можно получить линейный адрес его начала и конца соответственно.

Скрипт, приведенный ниже, распечатывает список всех существующих сегментов, с указанием их основных характеристик.

```

auto a;
a=FirstSeg();
Message(">Name | Start |End |BASE\n");
Message(">-----\n");
while(a!=BADADDR)
{
    Message(">%s|%08x|%08x|%08x\n",
    SegName(a),a,SegEnd(a),SegByName(SegName(a))/0x10);
    a=NextSeg(a);
}
Message(">-----\n\n");

>Name | Start |End |BASE
>-----
>seg000|00010000|00010012|00001000
>seg001|00020100|0002013c|00002000
>seg002|00100010|00100100|00010000
>-----

```

Узнать обо всех остальных атрибутах сегмента можно при помощи функции GetSegmentAttr.

### Сводная таблица функций

??? # Верстальщику #Unfortunately Change Table

<b>функции преобразования адресов</b>	
имя функции	краткое описание
long MK_FP (long seg, long off)	преобразует сегментный адрес в линейный
char atoa (long ea)	преобразует линейный адрес в строковой сегментный
<b>функции, работающие с сегментами</b>	
<b>функции создания и удаления сегментов</b>	
имя функции	краткое описание
success SegCreate(long startea,long endea,long base,long use32,long align,long comb)	создает новый сегмент или изменяет атрибуты уже существующего сегмента
success SegDelete (long ea,long disable)	удаляет сегмент и при необходимости, принадлежащие ему адреса виртуальной памяти
<b>функции изменения основных характеристик сегмента</b>	
имя функции	краткое описание
success SegBounds (long ea,long startea,long endea,long disable)	задает новый адрес начала и адрес конца сегмента, при необходимости удаляет освободившиеся адреса виртуальной памяти
<b>функции получения основных характеристик сегмента</b>	
имя функции	краткое описание
long SegStart (long ea)	возвращает линейный адрес начала сегмента
long SegEnd (long ea)	возвращает линейный адрес конца сегмента
long SegByName (char segname)	по имени сегмента определяет его базовый адрес
long SegByBase(long base)	по базовому адресу сегмента определяет линейный адрес его начала
<b>Функции изменения атрибутов сегмента</b>	
имя функции	краткое описание



success SegRename (long ea,char name)	изменяет имя сегмента
success SegAddrng (long ea,long use32)	изменяет разрядность сегмента
success SegAlign (long ea,long alignment)	изменяет кратность выравнивания сегмента
success SegComb (long segea,long comb)	изменяет атрибут объединения сегментов
success SegClass (long ea,char class)	изменяет класс сегмента
success SegDefReg (long ea,char reg,long value)	изменяет значение сегментных регистров
success SetSegmentType(long segea,long type)	изменяет тип сегмента
<b>функции получения атрибутов сегмента</b>	
<b>имя функции</b>	<b>краткое описание</b>
long GetSegmentAttr (long segea,long attr)	возвращает атрибуты сегмента
char SegName (long ea)	возвращает имя сегмента
<b>функции трассировки сегментов</b>	
<b>имя функции</b>	<b>краткое описание</b>
long FirstSeg ()	возвращает линейный адрес начала первого сегмента
long NextSeg (long ea)	возвращает линейный адрес начала следующего сегмента
<b>Функции, работающие с селекторами</b>	
<b>функции создания и удаления селекторов</b>	
<b>имя функции</b>	<b>краткое описание</b>
void SetSelector (long sel,long value)	создает новый селектор или изменяет значение уже существующего селектора
void DelSelector (long sel)	удаляет селектор
<b>утилиты</b>	
<b>имя функции</b>	<b>краткое описание</b>
long AskSelector (long sel)	возвращает значение селектора в параграфах
long FindSelector (long val)	возвращает селектор с указанным значением

### long MK\_FP (long seg,long off)

Функция преобразует сегментный адрес в линейный по следующей схеме  $ea = seg * 0x10 + off$ . Перед ее использованием необходимо убедиться что "seg" представляет собой именно базовый адрес сегмента, выраженный в параграфах, а не селектор, иначе полученный результат будет неверен.

Оператор "квадратные скобки" полностью аналогичен функции MK\_FP, но обладает более компактной формой записи (6 символов "MK\_FP()" вместо двух "[ ]").

**Замечание:** в комментариях, содержащихся в файле <idc.idc>, часто используется конструкция ["имя сегмента", смещение]. Попытка использования этой схематической конструкции в коде скриптов приведет к появлению синтаксической ошибки, но если передать имя сегмента в строковой переменной, оператор «квадратные скобки» автоматически подставит его базовый адрес – если, конечно, такой сегмент существует; в противном случае строка будет преобразована в число, согласно правилам преобразования «строка-→ число» в IDA-Cи (см. «Язык скриптов IDA Си» - «Объявление переменных, типы переменных, преобразования переменных»)

Напротив, макрос MK\_FP всегда преобразует переданное ему имя сегмента в строку, даже если сегмент с таким именем существует.

Пример использования:

```
Message(">[seg %X,off%X]=%X=%X\n",0x1000,0x6,MK_FP(0x1000,0x6),[0x1000,0x6]);
```

```
>[seg 1000,off6]=10006=10006
```

??? #верстальщику – change table

аргумент	пояснения
seg	базовый адрес сегмента (не селектор!), выраженный в параграфах
off	смещение ячейки в сегменте
return	пояснения
long	32-битный линейный адрес ячейки

**Родственные функции:** оператор []

**Интерактивный аналог:** “~View\Calculate” <?>

char atoa(long ea)

Функция преобразует линейный адрес ea в строковой сегментный, действуя по следующему алгоритму:

- если линейный адрес ea принадлежит некоторому сегменту, смещение вычисляется относительно его базового адреса, а сам адрес записывается в виде “имя сегмента:смещение”
- если линейный адрес ea не принадлежит ни одному сегменту, преобразование выполняется по формуле  $seg = \frac{ea}{0x10}$ ;  $off = ea - seg$ .

Пример использования:

```
Message(">%s\n", atoa(0x200010));
>0:00200010

SegCreate(0x200000, 0x201000, 0x20000, 0, 0, 0);
0. Creating a new segment (00200000-00201000) ... .. OK
Message(">%s\n", atoa(0x200010));
>seg000:0010
```

??? #верстальщику – change table

аргумент	пояснения	
ea	32-разрядный линейный адрес	
return	=return	пояснения
	!=	сегментный адрес в строковом представлении
	==	ошибка

**Родственные функции:** нет

**Интерактивный аналог:** нет

success SegCreate(long startea, long endea, long base, long use32, long align, long comb)

Функция создает новый сегмент. Сегмент задается линейным адресом начала (**startea**), линейным адресом конца (**endea**) и базовым адресом (**BASE**), используемым для адресации ячеек внутри сегмента.

Адрес начала задает наименьший линейный адрес, принадлежащий данному сегменту, напротив, адрес конца, задает линейный адрес на единицу превышающий наибольший адрес, принадлежащий указанному сегменту. Такая мера необходима для поддержки сегментов нулевой длины – в силу архитектурных ограничений IDA, один линейный адрес не может являться и началом, и концом сегмента одновременно, поэтому, приходится «искусственно» увеличивать адрес конца сегмента.

Смещение первого байта внутри сегмента вычисляется по формуле  $startoffset = startea - BASE * 0x10$ . Поддержка **базирования** позволяет создавать сегменты с произвольным начальным смещением, например, равным 0x100. Поскольку, смещения выражаются неотрицательными величинами, начальный адрес должен быть не меньше базы сегмента, измеряемой в байтах.

Напротив, базовый адрес по известному адресу начала сегмента и смещению первого байта в сегменте вычисляется по формуле  $BASE = \frac{startea - startoffset}{0x10}$

Атрибут **use32** задает разрядность сегмента – нулевое значение соответствует 16-разрядному сегменту, любое другое – 32-разрядному. Разрядность оказывает влияние только на способ дизассемблирования машинных инструкций, принадлежащих этому сегменту, но независимо от разрядности смещения в сегменте всегда задаются 32-битовыми значениями.

Атрибут **align** задает кратность выравнивания сегмента, помещая соответствующую директиву в ассемблерный листинг. Никакого влияния на размещение сегмента в виртуальной памяти атрибут align не оказывает!

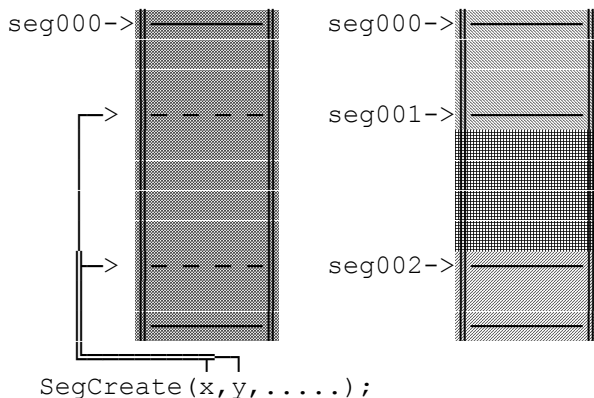
Атрибут **comb** задает флаг комбинирования, разрешающий (запрещающий) линкеру объединять несколько сегментов в один. Никакого влияния на объединение сегментов в виртуальной памяти атрибут comb не оказывает – независимо от его значения смежные сегменты не будут автоматически объединены.

### Детали:

а) Базовый адрес задается неотрицательным 16-разрядным значением и может адресовать не более одного мегабайта памяти ( $\frac{0x10000 * 0x10}{1024 * 1024} = 1$ ). Если этого недостаточно, следует указать вместо базового адреса **селектор**, который необходимо предварительно создать вызовом функции SetSelector.

Если базовый адрес, переданный функции SegCreate, больше 0x10000, IDA автоматически создаст новый селектор и использует его для адресации данного сегмента.

б) Если передать линейный адрес, принадлежащий некоторому сегменту, но отличный от адреса начала (т.е. попытаться создать вложенный сегмент), функция автоматически укорит или разобьет данный сегмент и создаст новый (см. рисунок)



## Рисунок 22 ??? Художнику - перерисовать

Пример использования:

Пусть необходимо создать сегмент с (задача вычисления базовго адреса)

```
SegCreate(0x1000,0x4000,0x100,0,0,0);
0. Creating a new segment (00001000-00004000) ... .. OK
```

Name	Start	End	Align	Base	Type	Cls	32es	ss	ds
seg000	00000000	00003000	at	0100	pri		N FFFF	FFFF	FFFF 00001000 00004000

```
SegCreate(0x2000,0x3000,0x200,0,0,0);
1. Creating a new segment (00002000-00003000) ...
   Additional segment (00003000-00004000) ...
2. Creating a new segment (00003000-00004000) ... .. OK
... OK
```

Name	Start	End	Align	Base	Type	Cls	32es	ss	ds
seg000	00000000	00001000	at	0100	pri		N FFFF	FFFF	FFFF 00001000 00002000
seg001	00000000	00001000	at	0200	pri		N FFFF	FFFF	FFFF 00002000 00003000
seg002	00002000	00003000	at	0100	pri		N FFFF	FFFF	FFFF 00003000 00004000

??? #Верстальщику – change table

аргумент	пояснения	
startea	32-разрядный линейный адрес начала сегмента	
endea	величина на единицу большая последнего принадлежащего сегменту адреса	
Base	16-разрядный базовый адрес в параграфах или указатель на селектор	
use32	=use32	пояснения
	==0	16-разрядный сегмент
	==1	32-разрядный сегмент
aling	кратность выравнивания начала сегмента	
comb	комбинирование сегмента	
return	=return	пояснения
	==1	операция завершилась успешно
	==0	ошибку

**Родственные функции:** SetSelector; SegClass; SegAlign; SegComb; SegAddrng;

**Интерактивный аналог:** “~\View\Segments”, <Insert>;

### success SegDelete(long ea,long disable)

Функция удаляет сегмент вместе с результатами дизассемблирования (метками, функциями, переменными и т.д.) и адресным пространством виртуальной памяти, принадлежащей сегменту.

Аргумент **ea** – представляет собой любой линейный адрес, принадлежащий сегменту, но не адрес его конца (см. описание функции SegCreate – адрес конца сегмента на единицу больше наибольшего адреса, принадлежащего сегменту)

Аргумент **disable** будучи неравным нулю указывает на необходимость удаления адресного пространства виртуальной памяти, принадлежащей сегменту, в противном случае удаляется лишь сам сегмент вместе с результатами дизассемблирования, а содержимое ячеек виртуальной памяти остается неизменным.

**Внимание:** операция удаления сегментов необратима – повторное создание уничтоженного сегмента не восстановит удаленные вместе с сегментом метки, функции, переменные, комментарии и другие результаты дизассемблирования.

Если переданный функции линейный адрес не принадлежит ни одному сегменту, она возвратит нулевое значение, сигнализируя об ошибке. Нормальное завершение возвращает единицу.

Пример использования:

```
seg000:0000 seg000      segment byte public '' use16
seg000:0000            assume cs:seg000
seg000:0000            assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000 aHelloIdaPro db 'Hello, IDA Pro! ',0Dh,0Ah          ; Test
seg000:0000 seg000      ends
```

а) исходный сегмент, содержащий строку “aHelloIdaPro” и комментарий “Test”

```
SegDelete (SegByBase (SegByName ("seg000") >>4) , 0) ;
```

б) удаление сегмента без уничтожения виртуальной памяти

```
0:00010000            db 48h ; H
0:00010001            db 65h ; e
0:00010002            db 6Ch ; l
0:00010003            db 6Ch ; l
0:00010004            db 6Fh ; o
0:00010005            db 2Ch ; ,
0:00010006            db 20h ;
0:00010007            db 49h ; I
0:00010008            db 44h ; D
0:00010009            db 41h ; A
0:0001000A            db 20h ;
0:0001000B            db 50h ; P
0:0001000C            db 72h ; r
0:0001000D            db 6Fh ; o
0:0001000E            db 21h ; !
0:0001000F            db 20h ;
0:00010010            db 0Dh ;
```

с) результат – сегмент удален, содержимое виртуальной памяти - нет

```
SegCreate (0x10000, 0x10012, 0x1000, 0, 0, 0) ;
```

д) Попытка создания нового сегмента для восстановления удаленного

```
seg000:0000
seg000:0000 ; Segment type: Regular
seg000:0000 seg000      segment at 1000h private '' use16
seg000:0000            assume cs:seg000
seg000:0000            assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000            db 48h ; H
seg000:0001            db 65h ; e
seg000:0002            db 6Ch ; l
seg000:0003            db 6Ch ; l
seg000:0004            db 6Fh ; o
seg000:0005            db 2Ch ; ,
seg000:0006            db 20h ;
seg000:0007            db 49h ; I
seg000:0008            db 44h ; D
seg000:0009            db 41h ; A
seg000:000A            db 20h ;
seg000:000B            db 50h ; P
```

```

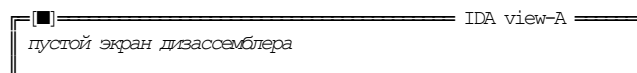
seg000:000C      db  72h ; r
seg000:000D      db  6Fh ; o
seg000:000E      db  21h ; !
seg000:000F      db  20h ;
seg000:0010      db  0Dh ;
seg000:0011      db  0Ah;
seg000:0011 seg000      ends

```

е) результат – сегмент восстановлен, а строка и комментарий – нет.

```
SegDelete(0x10000, 1);
```

ф) удаление сегмента вместе с принадлежащей ему виртуальной памятью



г) результат – виртуальная память удалена

```
SegCreate(0x10000, 0x10012, 0x1000, 0, 0, 0);
```

к) попытка создания нового сегмента для восстановления виртуальной памяти

```

seg000:0000 ; Segment type: Regular
seg000:0000 seg000      segment at 1000h private '' use16
seg000:0000      assume cs:seg000
seg000:0000      assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000      db  ? ; unexplored
seg000:0001      db  ? ; unexplored
seg000:0002      db  ? ; unexplored
seg000:0003      db  ? ; unexplored
seg000:0004      db  ? ; unexplored
seg000:0005      db  ? ; unexplored
seg000:0006      db  ? ; unexplored
seg000:0007      db  ? ; unexplored
seg000:0008      db  ? ; unexplored
seg000:0009      db  ? ; unexplored
seg000:000A      db  ? ; unexplored
seg000:000B      db  ? ; unexplored
seg000:000C      db  ? ; unexplored
seg000:000D      db  ? ; unexplored
seg000:000E      db  ? ; unexplored
seg000:000F      db  ? ; unexplored
seg000:0010      db  ? ; unexplored
seg000:0011      db  ? ; unexplored
seg000:0011 seg000      ends

```

л) результат – адресное пространство восстановлено, но содержимое виртуальной памяти – нет.

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес принадлежащий сегменту	
disable	=disable	пояснения
	==0	не уничтожать адреса, принадлежащие сегменту
	==1	уничтожать адреса, принадлежащие сегменту
return	=return	
	==1	успешное завершение операции
	==0	ошибка

Родственные функции: SegCreate

Интерактивный аналог: “~Edit\Segments\Delete segment”; “~View\Segments”, <Del>

### success SegBounds(long ea,long startea,long endea,long disable)

Функция позволяет изменять адреса начала и конца сегмента, при необходимости уничтожая освободившуюся при уменьшении сегмента виртуальную память. Базовый адрес она изменить не в состоянии и при возникновении необходимости его модификации следует воспользоваться функцией SegCreate.

Перед увеличением границ сегмента необходимо убедиться что для расширения сегмента имеется достаточное количество свободного места, в противном случае требуется предварительно укоротить соседние сегменты на соответствующую величину.

Адрес начала сегмента должен быть не меньше базового адреса сегмента, умноженного на шестнадцать, в противном случае первые байты сегмента будут иметь отрицательное смещение. Изменить базовый адрес можно вызовом SegCreate или SetSelector (если базовый адрес задан селектором).

Адрес конца сегмента должен на единицу превышать наибольший адрес, принадлежащий сегменту. Подробнее об этом рассказывается в описании функции SegCreate.

При уменьшении сегмента ранее принадлежащая ему виртуальная память освобождается. Позднее она может быть выделена в отдельный сегмент или же объединена с одним из смежных сегментов. Если же ни то, ни другое не планируется, то освободившуюся память можно удалить, передав функции ненулевое значение флага **disable**.

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес, принадлежащий сегменту	
startea	32-разрядный линейный адрес нового начала сегмента	
endea	величина на единицу большая последнего принадлежащего сегменту адреса	
disable	уничтожать освободившиеся адреса	
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** SegCreate

**Интерактивный аналог:** <Alt-S>; “~\View\Segments”, <Ctrl-E>

### long SegStart(long ea)

Функция принимает любой линейный адрес, принадлежащий сегменту, и возвращает линейный адрес начала данного сегмента.

Если переданный адрес не принадлежит ни одному сегменту, функция возвратит BADADDR, сигнализируя об ошибке.

Пример использования:

```
SegCreate(0x10000,0x20000,0x1000,0,0,0);
```

а) создаем сегмент с адресом начала 0x10000 и адресом конца 0x20000

0. Creating a new segment (00010000-00020000) ... .. OK

б) сегмент успешно создан

```
Message(">%X\n",SegStart(0x10100));
```

с) вызываем функцию SegStart, передавая ей один из адресов, принадлежащих сегменту

>10000

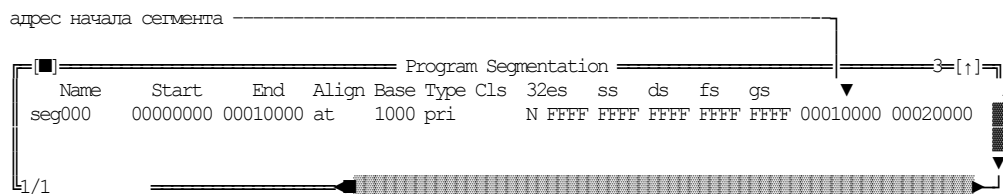
d) результат – функция вернула адрес начала сегмента

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес принадлежащий сегменту	
return	=return	пояснение
	!=BADADDR	линейный адрес начала сегмента
	==BADADDR	ошибка

**Родственные функции:** SegEnd

**Интерактивный аналог:** “~\View\Segments”



### long SegEnd(long ea)

Функция принимает любой линейный адрес, принадлежащий сегменту, и возвращает линейный адрес конца данного сегмента.

Если указанный адрес не принадлежит ни одному сегменту, функция возвратит значение BADADDR, сигнализируя об ошибке.

Линейный адрес конца сегмента на единицу больше наибольшего адреса, принадлежащего сегменту. Подробнее об этом рассказывается в описании функции SegCreate.

Пример использования:

```
SegCreate(0x1000, 0x2000, 0x100, 0, 0, 0);
```

```
SegCreate(0x2000, 0x3000, 0x200, 0, 0, 0);
```

а) создаем два смежных сегмента с адресами начала и конца 0x1000; 0x2000 и 0x2000; 0x3000 соответственно

```
0. Creating a new segment (00001000-00002000) ... .. OK
```

```
1. Creating a new segment (00002000-00003000) ... .. OK
```

б) сегменты успешно созданы

```
Message(">%X\n", SegEnd(0x1100));
```

с) вызываем функцию SegEnd, передавая ей один из адресов, принадлежащих первому сегменту

>2000

d) результат – адрес конца первого сегмента

```
Message(">%X\n", SegStart(0x2000));
```

е) вызываем функцию SegStart, передавая ей адрес конца первого сегмента

>2000

f) результат – адрес начала **второго** сегмента. Таким образом, наглядно



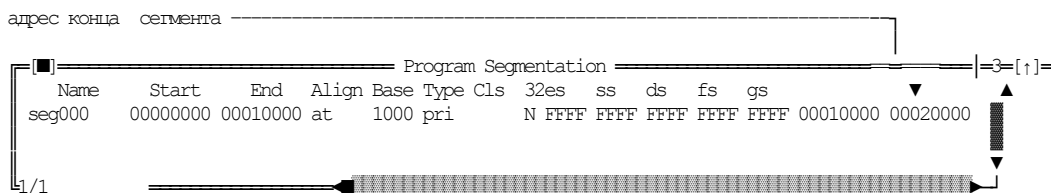
продемонстрировано – адрес конца сегмента не принадлежит самому сегменту.

??? #верстальщику – change table

аргумент	пояснения	
ea	любой 32-разрядный линейный адрес, принадлежащий сегменту	
return	=return	пояснения
	!=BADADDR	линейный адрес конца сегмента
	==BADADDR	ошибка

**Родственные функции:** SegStart

**Интерактивный аналог:** “~View\Segments”



### long SegByName(char segname)

Функция по имени сегмента (с учетом регистра) определяет его базовый адрес.

Если базовый адрес представляет собой селектор, то функция автоматически возвратит его значение.

Если сегмента с указанным именем не существует, функция возвратит значение BADADDR, сигнализируя об ошибке.

#### **Детали:**

а) вызов SegByName – единственный способ определения базового адреса сегмента. Чтобы определить базовый адрес сегмента, по некому линейному адресу, принадлежащему сегменту, необходимо воспользоваться конструкцией `BASE = SegByName(SegName(ea))`.

б) Поскольку, IDA допускает совместное существование двух и более одноименных сегментов, определение базового адреса сегмента по его имени позволяет определить базовый адрес лишь одного из них.

Но никакого другого (во всяком случае документированного) способа определения базового адреса не существует и выходом из такой ситуации становится отказ от использования одноименных сегментов.

**Пример использования:**

```
SegCreate(0x1000, 0x2000, 0x100, 0, 0, 0);
SegRename(0x1000, "MySeg");
```

а) создаем новый сегмент с базовым адресом 0x1000 и тут же переименовываем его в “MySeg”

```
Message(">%X\n", SegByName("MySeg"));
```

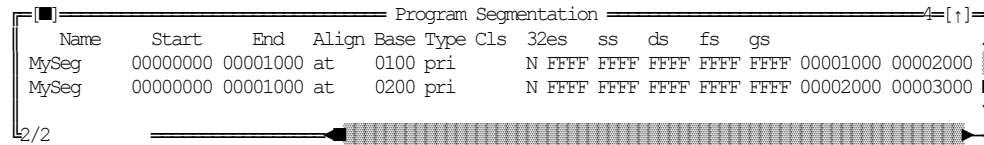
б) вызываем функцию SegByName передавая ей имя только что созданного сегмента

>1000

c) результат – базовый адрес сегмента MySeg

```
SegCreate (0x2000, 0x3000, 0x200, 0, 0, 0) ;
SegRename (0x2000, "MySeg") ;
```

d) создаем второй сегмент с именем "MySeg", но на этом раз базовым адресом, равным 0x2000



e) это получилось! Теперь существуют два одноименных сегмента

```
Message (">%X\n", SegByName ("MySeg")) ;
```

f) вызываем функцию SegByName передавая ей имя "MySeg"

>1000

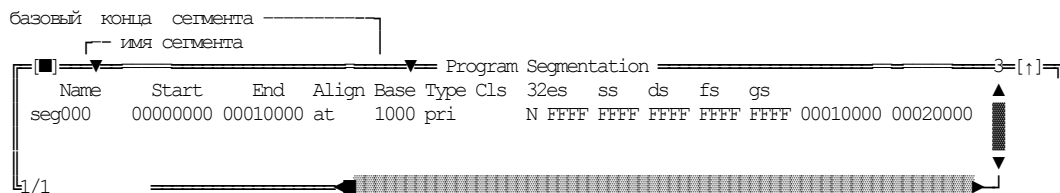
g) результат – базовый адрес первого сегмента

??? #верстальщику – change table

аргумент	пояснения	
segname	имя сегмента (регистр учитывается)	
return	=return	пояснения
	!=BADADDR	базовый адрес сегмента или значение селектора
	==BADADDR	ошибка

**Родственные функции:** SegRename, SegName

**Интерактивный аналог:** "~View\Segments"



### long SegByBase(long base)

Функция по базовому адресу (селектору) сегмента определяет линейный адрес его начала. Если передать базовый адрес, не принадлежащий ни одному сегменту, или существующий, но не использующийся ни одним сегментом селектор, функция возвратит значение BADADDR, сигнализируя об ошибке.

Поскольку, IDA допускает совместное существование двух и более сегментов с одинаковыми базовыми адресами, определение адреса начала сегмента по его базовому адресу позволяет определить адрес начала лишь одного из них.

Пример использования:

```
SegCreate (0x1100, 0x2000, 0x100, 0, 0, 0) ;
```

a) создаем сегмент с базовым адресом 0x100 и линейным адресом начала 0x1100

```
Message (">%X\n", SegByBase (0x100)) ;
```

b) вызываем функцию SegByBase для получения линейного адреса начала сегмента по его базе

>1100

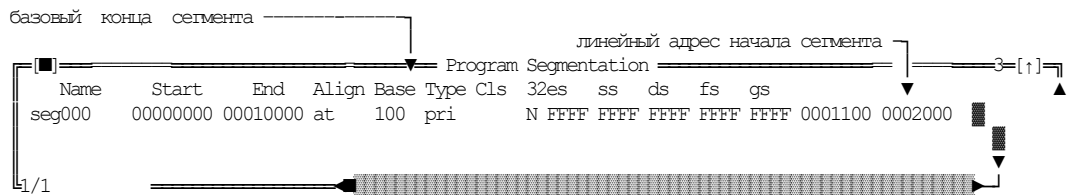
c) результат – линейный адрес начала сегмента

??? #верстальщику – change table

аргумент	пояснения	
base	базовый адрес сегмента или значение селектора	
return	=return	пояснения
	!=BADADDR	линейный адрес начала сегмента
	==BADADDR	ошибка

**Родственные функции:** SegByName

**Интерактивный аналог:** “~View\Segments”



success SegRename(long ea,char name)

Функция переименовывает сегмент, возвращая при удачном завершении операции ненулевое значение и ноль в противном случае.

Аргумент **ea** представляет собой любой линейный адрес, принадлежащий сегменту. Если передать адрес не принадлежащий никакому сегменту, функция возвратит ошибку.

Аргумент **name** задает новое имя сегмента с учетом регистра. Если имя начинается с цифры, функция автоматически дополнит его знаком прочерка. Если в имени содержится недопустимые символы, они будут автоматически заменены знаком прочерка.

Допустимые символы перечисляются в полях NameChars конфигурационного файла <ida.cfg>, значение которых по умолчанию приведено в таблице ???

Попытка присвоить сегменту пустое имя (аргумент name равен "") приводит к ошибке.

IDA допускает существование двух и более сегментов с одинаковыми именами, однако, использование этой возможности влечет невозможность определения базового адреса сегмента. Подробнее – см. описание функции SegByName.

??? #Верстальщику Create New Table

платформа	перечень символов, допустимых в именах
PC	"\$?@" <sup>5</sup>
	" 0123456789"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz";
Java	"\$ _@?!" <sup>6</sup>

<sup>5</sup> Служебные символы ассемблера

<sup>6</sup> Символы, определенные **только** для специальных режимов Java-ассемблера

	"0123456789<>"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
	"abcdefghijklmnopqrstuvwxyz"
	"АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ" <sup>7</sup>
	"абвгдежзийклмнопрстуфхцчшщъыьэюя";
TMS320C6	"\$_0123456789"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
	"abcdefghijklmnopqrstuvwxyz"
PowerPC	"_0123456789."
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
	"abcdefghijklmnopqrstuvwxyz"

**Таблица 4**

Пример использования:

```
SegCreate(0x1000, 0x2000, 0x100, 0, 0, 0);
```

```
Message(">%s\n", SegName(0x1000));
```

a) создаем сегмент и тут же определяем его имя

```
>seg000
```

b) имя сегмента – “seg000”

```
SegRename(0x1000, "666");
```

```
Message(">%s\n", SegName(0x1000));
```

c) вызываем функцию SegRename для переименования сегмента в “666” и тут же получаем имя сегмента при помощи SegName

```
>_666
```

d) результат – новое имя сегмента “\_666”, - функция автоматически добавила спереди знак прочерка, поскольку имя начиналась с цифры

```
SegRename(0x1000, "Русский квас");
```

```
Message(">%s\n", SegName(0x1000));
```

e) вызываем функции SegRename для переименования сегмента в «Русский квас» и тут же получаем имя сегмента при помощи SegName

```
>
```

f) результат – все запрещенные символы были заменены знаками прочерка

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес, принадлежащий сегменту	
name	новое имя сегмента	
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** SegName, SegByName

**Интерактивный аналог:** “~View\Segments”, <Ctrl-E>; <Alt-S>

<sup>7</sup> Национальные (русские символы)

### success SegAddrng(long ea,long use32)

Функция изменяет разрядность сегмента, определяя каким образом будет дизассемблироваться машинные инструкции.

На платформе Intel 386+ префикс 0x66 перед инструкцией в 16-разрядном сегменте указывает на использование 32-битных операндов и, соответственно, в 32-разрядном сегменте наоборот. Изменение адресации так же затрагивает интерпретацию префикса предопределения адреса - 0x67, видов адресации и т.д.

Неверный выбор разрядности сегмента приводит к некорректному дизассемблированию – появлению бессмысленного «мусора», бессвязных инструкций.

IDA определяет разрядность сегмента на основе информации, содержащейся в заголовках файлов. При загрузке бинарных файлов, равно как и MS-DOS exe файлов разрядность созданных сегментов по умолчанию равна нулю. Если IDA неправильно определила разрядность одного или нескольких сегментов, ее следует исправить вручную.

**Внимание:** изменение разрядности сегмента уничтожает все результаты дизассемблирования – метки, функции, переменные и т.д.

Аргумент **ea** задает любой линейный адрес, принадлежащий сегменту. Если передать адрес не принадлежащий никакому сегменту, функция возвратит ошибку.

Нулевое значение аргумента **use32** указывает на 16-битовую разрядность сегмента и приводит к помещению в ассемблерный листинг атрибута размера сегмента **use16**; в противном случае сегмент считается 32-разрядным, а атрибут размера сегмента – **use32**.

Независимо от разрядности сегмента, IDA всегда выражает смещения 32-битовыми значениями и допустимо создания 16-разрядного сегмента размером более 64 килобайт, однако, следует помнить, что в дальнейшем такой «большой» сегмент не сможет быть ассемблирован.

Функция не учитывает выбранный тип процессора и допускает использование 32-разрядного режима даже с 8086 (!) процессором.

Пример использования:

```
SegCreate(0x1000,0x2000,0x100,0,0,0);
```

а) Создание 16-разрядного сегмента

```
seg000:0000 seg000 segment at 100h private '' use16
```

б) Сегмент успешно создан. Атрибут размера сегмента выделен жирным шрифтом.

```
SegAddrng(0x1000,1);
```

с) Вызов функции SegAddrng для изменения разрядности сегмента

```
seg000:00000000 seg000 segment at 100h private '' use32
```

е) Разрядность сегмента успешно изменена

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес, принадлежащий сегменту	
use32	=use32	пояснения
	==0	16-разрядный сегмент
	==1	32-разрядный сегмент
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

## Родственные функции: SegCreate

**Интерактивный аналог:** “~View\Segments”, <Ctrl-E>; <Alt-S>

success SegAlign(long ea,long alignment)

Функция управляет выравниванием сегмента, помещая в ассемблерный текст соответствующий атрибут вырывания (byte, word, dword, para, page, mpage). Подробнее о каждом из этих атрибутов можно прочитать в документации, прилагаемой к используемому линкеру.

Никакого влияния на дизассемблируемый образ файла функция не оказывает и **не выравнивает** сегмент в виртуальной памяти. Это подтверждает следующий эксперимент:

```
SegCreate(0x1003,0x2000,0x100,0,0,0);
```

а) создание нового сегмента с адресом начала равным 0x1003

```
seg000:0003 seg000          segment at 100h private '' use16
```

б) сегмент создан; смещение первого байта в сегменте равно трем

```
Message(">1%x\n",SegAlign(0x1003,saRelWord));
```

с) вызов функции SegAlign для выравнивания сегмента по границе слова

```
seg000:0003 seg000          segment word private '' use16
```

```
>1
```

д) результат – функция поместила в ассемблерный текст атрибут выравнивания ‘word’ (он выделена жирным шрифтом) и сигнализировала об успешном завершении. Но линейный адрес начала сегмента не был изменен (он выделен жирным шрифтом)!

Атрибут выравнивания возымеет действие только на стадии компоновки объективного файла, расположив сегмент по адресу, кратным двум. Это приведет к несоответствию смещений в дизассемблируемом и целевом файле, что, скорее всего, вызовет невозможность корректного выполнения откомпилированной программы. Для выравнивания сегмента в виртуальной памяти следует воспользоваться функцией SegBounds для перемещения его начала по заданному адресу.

Аргумент **ea** задает любой линейный адрес, принадлежащий сегменту. Если передать адрес не принадлежащий никакому сегменту, функция возвратит ошибку.

Аргумент **alignment** представляет собой атрибут выравнивания сегмента и может принимать одно из значений, перечисленных в таблице ???

определение	#	пояснения
saAbs	0	Безусловное выравнивание
saRelByte	1	Выравнивание по границе байта (8 бит)
saRelWord	2	Выравнивание по границе слова (16 бит)
saRelPara	3	Выравнивание по границе параграфа (16 байт)
saRelPage	4	Выравнивание по границе страницы (256-байт для чипов Intel)
saRelDble	5	Выравнивание по границе двойного слова (4 байта)
saRel4K	6	Выравнивание по границе страницы (4 килобайта для PharLap OMF) <sup>8</sup>
saGroup	7	Segment group
saRel32Bytes	8	Выравнивание по границе 32 байта
saRel64Bytes	9	Выравнивание по границе 64 байта
saRelQword	10	Выравнивание по границе 8 байт

<sup>8</sup> Не поддерживается стандартным линкером LINK.

**Таблица 5**

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес, принадлежащий сегменту	
alignment	кратность выравнивания (смотри определения в таблице выше)	
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** SegCreate

**Интерактивный аналог:** “~View\Segments”, <Ctrl-E>; <Alt-S>

success SegComb(long segea,long comb)

Функция управляет объединением сегментов, помещая в ассемблерный листинг атрибут комбинирования (private, public, common, at, stack).

Атрибут комбинирования указывает компоновщику как следует комбинировать сегменты различных модулей, имеющие одно и то же имя. Подробную информацию о каждом из атрибутов можно найти в документации, прилагаемой к используемому линкеру.

Аргумент **segea** задает любой линейный адрес, принадлежащий сегменту. Если передать адрес не принадлежащий никакому сегменту, функция возвратит ошибку.

Аргумент **comb** представляет собой атрибут выравнивания, и может принимать одно из значений, приведенных в Таблице ???

Пример использования:

```
SegCreate(0x1000,0x2000,0x100,0,0,scPub);
```

a) создаем новый сегмент с атрибутом комбинирования public

```
seg000:0000 seg000 segment at 100h public ''
```

b) сегмент успешно создан, атрибут комбинирования выделен жирным шрифтом

```
SegComb(0x10000,scStack);
```

c) вызываем функцию SegComb для изменения атрибута комбинирования сегмента

```
seg000:0000 seg000 segment at 100h stack ''
```

d) атрибут комбинирования изменен

определение	#	пояснения
scPriv	0	Атрибут private. Не может быть соединен ни с одним другим сегментом
scPub	2	Атрибут public. Может быть объединен с другими сегментами с учетом требуемого выравнивания
scPub2	4	Атрибут, определенный Microsoft, то же самое, что и scPub .
scStack	5	Атрибут stack. Может быть объединен с public сегментами, но с выравниваем в один байт.
scCommon	6	Атрибут common.
scPub3	7	Атрибут, определенный Microsoft, то же самое, что и scPub .

**Таблица 6**

??? #верстальщику – change table

аргумент	пояснения	
ea	любой линейный адрес, принадлежащий сегменту	
comb	Атрибут (смотри определения в таблице 6)	
return	=return	Успешность завершения операции
	1	Операция завершилась успешно
	0	При выполнении операции произошла ошибка

**Родственные функции:** SegCreate

**Интерактивный аналог:** “~\View\Segments”, <Ctrl-E>; <Alt-S>

success SegClass(long ea,char class)

Функция изменяет атрибут класса сегмента, помещая в ассемблерный листинг атрибут класса.

Атрибут класса представляет собой текстовую строку, указывающую линкеру порядок следования сегментов. Если это не запрещено комбинаторным атрибутом (см. описание SegComb), линкер объединяет вместе сегменты с одинаковым именем. Рекомендуется назначать имена так, чтобы они отображали функциональное назначение сегментов, например, “CODE”, “DATA”, “STACK” и т.д. Общепринятые имена перечислены в таблице ???.

**Замечание:** большинство линкеров требуют, чтобы в объективном файле присутствовал хотя бы один сегмент с именем “CODE”, в противном случае, они могут отказаться обрабатывать такой файл или обработают его неправильно.

Пример использования:

```
SegCreate(0x1000,0x2000,0,0,scPub);
```

а) создаем сегмент с атрибутом public

```
seg000:0000 seg000          segment at 100h public ''
```

б) сегмент создан, по умолчанию атрибут класса сегмента отсутствует

```
SegClass(0x1000,"MySegment");
```

в) вызываем функцию SegClass для изменения атрибута класса сегмента

```
seg000:0000 seg000          segment at 100h public 'MySegment'
```

д) атрибут класса сегмента изменен (в тексте он выделен жирным шрифтом)

Класс	Пояснения	
CODE	Pure code	Сегмент кода
DATA	Pure data	Сегмент данных
CONST	Pure data	
BSS	Uninitialized data	Неинициализированные данные
STACK	Uninitialized data	Сегмент стека
XTRN	Extern definitions segment	

**Таблица 7** Общепринятые наименования классов сегментов

??? #верстальщику – change table



аргумент	пояснения	
ea	линейный адрес, принадлежащий сегменту	
class	класс сегмента	
return	=return	Успешность завершения операции
	1	Операция завершилась успешно
	0	При выполнении операции произошла ошибка

**Родственные функции:** SegCreate

**Интерактивный аналог:** “~View\Сегмент”, <Ctrl-E>; <Alt-S>

success SegDefReg(long ea,char reg,long value)

Функция определяет значение сегментных регистров, помещая в ассемблерный текст директиву ASSUME. Это указывает дизассемблеру (ассемблеру) к какому именно сегменту обращается тот или иной адресный операнд. Более подробную информацию по этому вопросу можно получить, обратившись к описанию директивы ASSUME в документации, прилагаемой к используемому ассемблеру или справочной (учебной) литературе по языку ассемблера.

Аргумент **ea** задает любой линейный адрес, принадлежащий сегменту, в которой необходимо поместить директиву ASSUME.

Аргумент **reg** задает сегментный регистр в символьном представлении, например, “DS”, “ES”, “SS” и т.д. Строчечные и прописные символы не различаются. Тип процессора при назначении сегментного регистра учитывается.

Аргумент **value** содержит базовый адрес сегмента, загружаемый в регистр и выраженный в параграфах. Если сегмента с указанным базовым адресом не существует, регистр приобретает неопределенное (“*nothing*”) значение.

В реализации этой функции допущена одна ошибка – независимо от успешности выполнения, возвращаемое значение никогда не сигнализирует об ошибке.

Пример использования (см. файл “*assume.idb*”, содержащийся на диске, прилагаемом к книге): пусть имеется один сегмент кода (“seg00”) и два сегмента данных (“seg001” и “seg002”), содержащих переменные “My666” и “My777” соответственно. По умолчанию значения сегментных регистров неопределенны и IDA не может определить к каким именно сегментам происходит обращения в командах “mov ax,ds:[0]” и “mov dx,es:[0]”, поэтому, вынуждена оставить операнды в виде непосредственных смещений. (в тексте они выделены жирным шрифтом)

```

seg000:0000 seg000          segment byte public 'CODE'
seg000:0000                assume cs:seg000
seg000:0000                assume es:nothing, ss:nothing, ds:nothing
seg000:0000                mov     ax, ds:0
seg000:0003                mov     dx, es:0
seg000:0003 seg000          ends
seg000:0003
seg001:0000 ; =====
seg001:0000
seg001:0000 ; Segment type: Pure data
seg001:0000 seg001          segment byte public 'DATA'
seg001:0000                assume cs:seg001
seg001:0000 My666           dw     6666h
seg001:0000 seg001          ends
seg001:0000
seg002:0000 ; =====

```

```

seg002:0000
seg002:0000 ; Segment type: Pure data
seg002:0000 seg002      segment byte public 'DATA'
seg002:0000      assume cs:seg002
seg002:0000 My777      dw 7777h
seg002:0000 seg002      ends

```

Задать значения сегментных регистров можно с помощью функции DefSegReg, вызов которой может выглядеть так:

```

DefSegReg (SegByName ("seg000"), "DS", SegByName ("seg001") >>4);
DefSegReg (SegByName ("seg000"), "ES", SegByName ("seg002") >>4);

```

В результате ее выполнения, IDA смогла отследить обращения к переменным, автоматически подставив их имена вместо смещений (в тексте они выделены жирным шрифтом). Поместив курсор в границы того или другого имени, нажатием клавиши <Enter> можно перейти к соответствующей ячейке памяти.

```

seg000:0000 seg000      segment byte public 'CODE'
seg000:0000      assume cs:seg000
seg000:0000      assume es:seg002, ss:nothing, ds:seg001
seg000:0000      mov     ax, My666
seg000:0003      mov     dx, es:My777
seg000:0003 seg000      ends
seg000:0003
seg001:0000 ; =====
seg001:0000
seg001:0000 ; Segment type: Pure data
seg001:0000 seg001      segment byte public 'DATA'
seg001:0000      assume cs:seg001
seg001:0000 My666      dw 6666h           ; DATA XREF: seg000:0000r
seg001:0000 seg001      ends
seg001:0000
seg002:0000 ; =====
seg002:0000
seg002:0000 ; Segment type: Pure data
seg002:0000 seg002      segment byte public 'DATA'
seg002:0000      assume cs:seg002
seg002:0000 My777      dw 7777h           ; DATA XREF: seg000:0003r
seg002:0000 seg002      ends
seg002:0000

```

??? #верстальщику – change table

аргумент	пояснение
ea	линейный адрес, принадлежащий сегменту
reg	сегментный регистр в строковом представлении (например, "DS")
val	базовый адрес сегмента, загружаемого в регистр, выраженный в параграфах
return	всегда единица

Родственные функции: *нет*

Интерактивный аналог: "~Edit\Segments\Change segment register value", <Alt-G>

### success SetSegmentType (long segea,long type)

Функция изменяет тип сегмента, оказывая влияние на его дизассемблирование.

Аргумент **segea** задает любой линейный адрес, принадлежащий сегменту. Если передать адрес не принадлежащий никакому сегменту, функция возвратит ошибку.

Аргумент **type** указывает на тип сегмента и может принимать одно из значений, перечисленных в таблице ????. При создании сегмента функцией SegCreate ему присваивается тип «неизвестный» - SEG\_NORM.

определение	#	Пояснения
SEG_NORM	0	Неизвестный тип
SEG_XTRN	1	Внешний ('extern') сегмент. Инструкции исключены
SEG_CODE	2	Сегмент кода
SEG_DATA	3	Сегмент данных
SEG_IMP	4	Сегмент Java implementation
SEG_GRP	6	Group of segments
SEG_NULL	7	Сегмент нулевой длины
SEG_UNDF	8	Сегмент неопределенного типа (не используется)
SEG_BSS	9	Неинициализированный сегмент
SEG_ABSSYM	10	Сегмент с определением абсолютных символов
SEG_COMM	11	Сегмент с общими определениями
SEG_IMEM	12	Внутренняя память процессора 8051

Таблица 8

Пример использования:

```
SegCreate (0x1000, 0x2000, 0x100, 0, 0, 0);
```

a) создаем новый сегмент (по умолчанию неизвестного типа)

```
seg000:0000 ; Segment type: Regular  
seg000:0000 seg000          segment at 100h private ''  
seg000:0000                assume cs:seg000  
seg000:0000                assume es:nothing, ss:nothing, ds:nothing
```

b) сегмент создан (тип выделен жирным шрифтом), автоматически внедрена директива ASSUME для определения сегментных регистров.

```
SetSegmentType (0x1000, SEG_DATA);
```

c) вызов функции SetSegnetType для установки типа «сегмент данных»

```
seg000:0000 ; Segment type: Pure data  
seg000:0000 seg000          segment at 100h private ''  
seg000:0000                assume cs:seg000
```

d) тип сегмента изменен, директива ASSUME, задающая значение регистров DS, ES и SS удалена.

??? #верстальщику – change table

аргумент	пояснения	
ea	любой линейный адрес, принадлежащий сегменту	
type	тип сегмента (возможные значения приведены в таблице ????)	
return	=return	пояснения
	!=0	успешное завершение операции

	0	ошибка
--	---	--------

**Родственные функции:** GetSegmentAttr

**Интерактивный аналог:** *нет*

long GetSegmentAttr(long segea, long attr)

Функция позволяет узнать следующие атрибуты сегмента: кратность выравнивания, комбинацию, привилегии доступа, разрядность, флаги сегмента, селектор, использующийся для базирования сегмента, тип сегмента и значения сегментных регистров, определенных функцией DefSegReg.

Об атрибуте выравнивания можно подробнее прочитать в описании функции SegAlign, об атрибутах комбинации – в описании функции SegComb, о типе сегмента рассказывается в описании функции “SetSegmentType”. Использование селекторов для базирования сегментов подробно описано в главах «Организация сегментов», “SegCreate” и “SetSelector”.

Аргумент **segea** задает любой линейный адрес, принадлежащий сегменту. Если передать адрес не принадлежащий никакому сегменту, функция возвратит ошибку.

Аргумент **attr** указывает функции – содержимое какого атрибута необходимо вернуть. Возможные значения аргумента attr приведены в таблице ???

??? #Верстальщику – Change Table

константа	#	пояснения	функция
SEGATTR_ALIGN	20	получить выравнивание сегмента	SegAlign
SEGATTR_COMB	21	получить атрибуты комбинации	SegComb
SEGATTR_PERM	22	<b>привилегии доступа</b>	
		SEGPPerm_EXEC	1   исполнение
		SEGPPerm_WRITE	2   запись
		SEGPPerm_READ	4   чтение
SEGATTR_USE32	23	32 разрядный сегмент	SegAddrmd
SEGATTR_FLAGS	24	<b>флаги сегмента</b>	
		ADDSEG_NOSREG	все сегментные регистры, заданные по умолчанию неопределены
		ADDSEG_OR_DIE	невозможно добавить сегмент
SEGATTR_SEL	26	селектор сегмента	SetSelector
SEGATTR_DEF_ES	28	значение регистра ES по умолчанию	DefSegReg
SEGATTR_DEF_CS	30	значение регистра CS по умолчанию	
SEGATTR_DEF_SS	32	значение регистра SS по умолчанию	
SEGATTR_DEF_DS	34	значение регистра DS по умолчанию	
SEGATTR_DEF_FS	36	значение регистра FS по умолчанию	
SEGATTR_DEF_GS	38	значение регистра GS по умолчанию	
SEGATTR_TYPE	40	тип сегмента	SetSegmentType

**Таблица 9 Типы сегментов**

??? #верстальщику – change table

аргумент	пояснения
ea	линейный адрес, принадлежащий сегменту
Type	тип сегмента (возможные значения приведены в таблице 9)

return	=return	Успешность завершения операции
	!=0	Операция завершилась успешно
	0	При выполнении операции произошла ошибка

**Родственные функции:** SegAddrng, SegAling, SegComb, SegClass, SegDefReg, SetSegmentType

**Интерактивный аналог:** *нет*

### char SegName(long ea)

Функция возвращает имя сегмента, заданного любым принадлежащим ему линейным адресом **ea**. Если передать адрес не принадлежащий никакому сегменту, функция возвратит пустую строку.

Пример использования:

```
SegCreate (0x1000, 0x2000, 0x100, 0, 0, 0);
SegRename (0x1000, "MySeg");
```

a) создаем сегмент и тут же переименовываем его в "MySeg"

```
MySeg:0000 MySeg segment at 100h private ''
```

b) сегмент успешно создан (имя выделено жирным шрифтом)

```
Message (">%s\n", SegName (0x1000));
```

c) вызываем функцию SegName для получения имени сегмента

```
>MySeg
```

d) результат – имя сегмента

??? #Верстальщику Table Change

аргумент	пояснения	
ea	линейный адрес, принадлежащий сегменту	
return	=return	Пояснение
	!=""	имя сегмента
	""	ошибка

**Родственные функции:** SegRename, SegByName

**Интерактивный аналог:** *по умолчанию имя сегмента отображается в адресе каждой ячейки*

### long FirstSeg()

Функция возвращает линейный адрес начала сегмента с наименьшим линейным адресом начала. Если не существует ни одного сегмента, функция возвратит значение BADADDR, сигнализируя об ошибке.

**Замечание:** *FirstSeg обычно используется в паре с NextSeg для получения списка адресов начала всех существующих сегментов.*

Пример использования:

```
SegCreate (0x1000, 0x2000, 0x9, 0, 0, 0);
```

```
SegCreate(0x100, 0x200, 0x10, 0, 0, 0);
```

a) создаем два сегмента с адресами начала 0x100 и 0x1000.

```
Message(">%X\n", FirstSeg());
```

b) вызываем функцию FirstSeg для получения наименьшего линейного адреса сегмента

```
>100
```

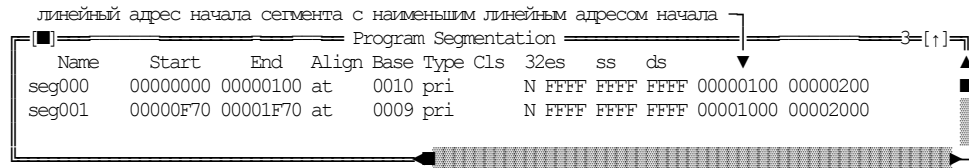
c) результат – линейный адрес сегмента с наименьшим линейным адресом начала

??? #Верстальщику Change Table

аргумент	пояснения	
нет	Нет	
return	=return	пояснение
	!=BADADDR	линейный адрес начала сегмента с наименьшим линейным адресом
	==BADADDR	ошибка

**Родственные функции:** NextSeg

**Интерактивный аналог:** “~View\Segments”



### long NextSeg(long ea)

Функция возвращает линейный адрес начала сегмента, линейный адрес начала которого больше чем переданный линейный адрес **ea**. Если сегмента удовлетворяющего такому условию не существует, функция возвращает значение BADADDR, сигнализируя об ошибке.

Передаваемый функции линейный адрес не обязательно должен быть адресом начала какого-то сегмента, более того, он вообще может не принадлежать ни одному сегменту – это ничуть не мешает функции вернуть линейный адрес следующего сегмента, при условии, что такой сегмент есть.

**Замечание:** NextSeg обычно используется в паре с FirstSeg для получения списка адресов начала всех существующих сегментов.

Конструкция NextSeg(0) аналогичная FirstSeg() при условии, что начальные адреса всех сегментов отличны от нуля (как правило, так практически всегда и бывает).

**Замечание:** функции, возвращающей линейный адрес начала сегмента, линейный адрес которого меньше переданного линейного адреса **ea**, не существует. Тем не менее, эта задача может быть решена на основе имеющихся функций FirstSeg и NextSeg

Ниже приводится пример реализации функции PrevSeg, работающей по следующему алгоритму: вызовами NextSeg сегменты перебираются один за другим до тех пор, пока линейный адрес начала очередного сегмента не превысит переданное значение **ea**. Затем – возвращается линейный адрес

начала предыдущего сегмента.

```
static PrevSegEx(ea)
{
    auto a;
    a=0;
    while (SegEnd(NextSeg(a))<ea) a=NextSeg(a);
    return a;
}
```

Пример использования NextSeg для получения списка адресов начала всех существующих сегментов:

```
SegCreate(0x100, .0x200, 0x10, 0, 0, 0);
SegCreate(0x1000, 0x2000, 0x100, 0, 0, 0);
SegCreate(0x10000, .0x20000, 0x1000, 0, 0, 0);
```

а) создаем три сегмента с адресами начала 0x100, 0x1000 и 0x10000

```
0. Creating a new segment (00000100-00000200) ... .. OK
1. Creating a new segment (00001000-00002000) ... .. OK
2. Creating a new segment (00010000-00020000) ... .. OK
```

б) сегменты успешно созданы

```
auto a;
a=FirstSeg();
while(a!=BADADDR)
{
    Message(">%X\n", a);
    a=NextSeg(a);
}
```

с) вызываем функцию NextSeg в цикле для получения линейных адресов начала всех существующих сегментов.

```
>100
>1000
>10000
```

д) результат – линейные адреса начала всех существующих сегментов

??? #Верстальщику Change Table

аргумент	пояснение	
ea	линейный адрес не обязательно принадлежащий какому-нибудь сегменту	
return	=return	Пояснение
	!=BADADDR	линейный адрес начала следующего сегмента
	==BADADDR	ошибка

Родственные функции: FirstSeg

Интерактивный аналог: “~View\Segments”

линейный адрес начала следующего сегмента-----

Name	Start	End	Align	Base	Type	Cls	32es	ss	ds		
seg000	00000000	00000100	at	0010	pri	N	FFFF	FFFF	FFFF	00000100	00000200
seg001	00000000	00001000	at	0100	pri	N	FFFF	FFFF	FFFF	00001000	00002000
seg002	00000000	00010000	at	1000	pri	N	FFFF	FFFF	FFFF	00010000	00020000



void SetSelector(long sel,long value)

Функция создает новый селектор или изменяет значение уже существующего селектора. Создаваемый селектор не должен совпадать с базовым адресом ни одного сегмента, иначе для базирования этого сегмента будет автоматически использоваться созданный селектор, а не его базовый адрес, что приведет к искажению всех смещений внутри сегмента.

Максимальное допустимое количество селекторов равно 4096 (0x1000 в шестнадцатеричной системе исчисления), а индексы селекторов могут принимать любые значение от 0x0 до 0xFFFF включительно.

Аргумент **sel** задает 16-разрядный индекс создаваемого или модифицируемого селектора. Старшее слово, передаваемого 32-битного значения автоматически обрезается, в результате чего существует угроза непреднамеренного искажения другого селектора, что подтверждает следующий эксперимент:

```
SetSelector(0x1,0x666);
Message(">%X\n",AskSelector(0x1));
```

a) создаем селектор с индексом 0x1 и значением 0x666 и тут же проверяем его значение

```
>666
```

b) селектор имеет значение 0x666

```
SetSelector(0x10001,0x777);
```

c) пытаемся создать селектор с индексом 0x10001 и значением 0x777

```
Message(">%X\n",AskSelector(0x1));
```

d) проверяем значение селектора 0x1

```
>777
```

e) результат – значение селектора 0x1 искажено! Старшие 16 бит индекса 0x10001 были обрезаны, в результате чего был модифицирован селектор 0x10001 AND 0xFFFF == 0x1

Аргумент **value** содержит 32-разрядное значение **базы** сегмента в параграфах. Оно будет автоматически использовано для базирования сегмента базовый адрес которого равен индексу данного селектора.

**Замечание:** функция *DeleteAll* (см. описание функции *DeleteAll*), удаляющая все сегменты, не уничтожает селекторов и их приходится удалять «вручную» вызовом *DelSelector*.

??? #Верстальщику Change Table

аргумент	пояснения	
sel	16-разрядный индекс селектора	
val	32-разрядное значение селектора в параграфах	
return	=return	пояснение
	void	функция не возвращает никакого значения

**Родственные функции:** AskSelector, DelSelector

**Интерактивный аналог:** “~View\Selector”, <Insert> - создает новый, <Ctrl-E>



изменяет значение уже существующего селектора.

#### void DelSelector(long sel)

Функция удаляет селектор, если он существует, в противном случае ничего не происходит. Если данный селектор использовался для базирования сегмента, новый базовый адрес сегмента будет равен  $base = sel * 0x10$ , а смещение первого байта в сегмента соответственно:  $startoffset = startea - sel * 0x10$ . Создав заново уничтоженный селектор вызовом SetSelector, можно все вернуть на свои места.

Аргумент **sel** задает 16-разрядный индекс удаляемого селектора. Старшее слово, передаваемого 32-битного значения автоматически обрезается, в результате чего существует угроза непреднамеренного уничтожения другого селектора, что подтверждает следующий эксперимент:

```
SetSelector(0x1, 0x666);  
Message(">%X\n", AskSelector(0x1));
```

a) создаем селектор с индексом 0x1 и значением 0x666 и тут же проверяем его значение

```
>666
```

b) селектор имеет значение 0x666

```
DelSelector(0x10001, 0x777);
```

c) пытаемся удалить селектор с индексом 0x10001

```
Message(">%X\n", AskSelector(0x1));
```

d) проверяем значение селектора 0x1

```
>FFFFFFFF
```

e) результат – селектора 0x1 был уничтожен! Старшие 16 бит индекса 0x10001 были обрезаны, в результате чего был удален селектор  $0x10001 \text{ AND } 0xFFFF == 0x1$

??? #Верстальщику Change Table

аргумент	пояснения	
sel	16-разрядный индекс уничтожаемого селектора	
return	=return	пояснение
	void	функция не возвращает никакого значения

**Родственные функции:** SetSelector

**Интерактивный аналог:** “~View\Selectors”, <Del>

#### long AskSelector(long sel)

Функция возвращает значение селектора в параграфах. Если запрошенный селектор отсутствует, функция возвратит переданное ей значение, сигнализируя об ошибке (внимание! не BADADDR). Поскольку, значение селектора не может быть равно его индексу, никакой неоднозначности не возникает, однако, автору книги совершенно непонятно чем вызвано такое решение – не лучше ли при неуспешном завершении функции возвращать BADADDR – значение, которое не может иметь ни один селектор?

Аргумент **sel** задает 16-разрядный индекс запрашиваемого селектора. Старшее слово передаваемого 32-битного значения автоматически обрезается, в результате существует возможность обращения совсем к другому селектору, чем предполагалось.

Ввиду всего вышесказанного проверка успешности завершения функции должна выглядеть так:

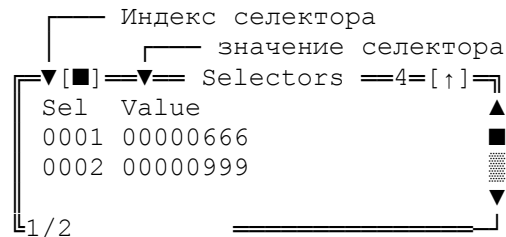
```
if ((selvalue=AskSelector(sel)) == (sel & 0xFFFF))
// ошибка
else
// успешное завершение функции
```

??? #Верстальщику Change Table

аргумент	пояснения	
sel	16-разрядный индекс запрашиваемого селектора	
return	=return	пояснения
	sel & 0xFFFF	ошибка
	!=(sel & 0xFFF)	32-разрядное значение селектора в параграфах

Родственные функции: SetSelector, FindSelector

Интерактивный аналог: “~View\Selectors”



long FindSelector(long val)

Функция возвращает индекс селектора с указанным значением **val**, выраженном в параграфах.

Если существуют два и более селектора с идентичными значениями, функция возвращает индекс первого из них в порядке создания. Если же ни одного селектора с таким значением не существует, функция возвращает младшие 16 бит переданного ей аргумента обратно.

Поскольку, значение селектора не может быть равно его индексу, никакой неоднозначности не возникает, однако, автору книги совершенно непонятно чем вызвано такое решение – не лучше ли при неуспешном завершении функции возвращать BADADDR – значение, которое не может иметь ни один селектор?

Ввиду всего вышесказанного проверка успешности завершения функции должна выглядеть так:

```
if ((sel=FindSelector(selvalue)) == (selvalue & 0xFFFF))
// ошибка
else
// успешное завершение функции
```

**Замечание:** поскольку с данным базовым адресом может существовать только один сегмент, в создании двух и более селекторов с одинаковыми значениями никакой необходимости нет. Тем не менее, функция SetSelector не препятствует этому и существованию двух и более селекторов с одинаковыми индексами в принципе возможно, – один селектор может использоваться для базирования некоторого сегмента, а остальные простаивать. В такой

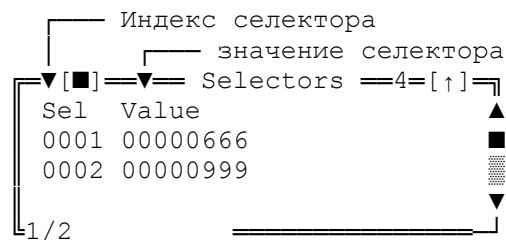
ситуации функция *FindSelector* может вернуть неверный результат, поэтому, перед ее вызовом следует убедиться, что существует не более одного селектора с каждым значением. Единственный документированный способ решения этой задачи заключается в последовательном переборе всех возможных селекторов в интервале от 0x0 до 0xFFFF.

??? #Верстальщику Change Table

аргумент	пояснения	
val	32-разрядное значение селектора в параграфах для поиска	
return	=return	пояснение
	==(val & 0xFFFF)	ошибка
	!=(val & 0xFFFF)	16-разрядный индекс селектора с указанным значением

**Родственные функции:** SetSelector, AskSelector

**Интерактивный аналог:** “~View\Selectors”



## ЭЛЕМЕНТЫ

### #Defenition

С каждым адресом виртуальной памяти связано 32-разрядное поле **флагов** (см. главу «Виртуальная память»). Флаги хранят содержимое ячейки, описывают ее представление и указывают на наличие связанных с ней объектов.

**Содержимое ячейки:** флаги хранят содержимое ячейки виртуальной памяти (в восьми младших битах) или указывают на то, что ячейка содержит неинициализированное значение (см. главу «Виртуальная память»).

**Представление:** флаги определяют будет ли отображаться данный байт в виде данных или машинной инструкции. Они так же позволяют уточнить в виде каких именно данных (массива, строки, переменной, непосредственного значения или смещения) он должен отображаться (см. главу «Типы элементов»)

**Связанные объект:** флаги указывают на наличие связанных с ячейкой объектов – меток, имен, комментариев и т.д. Сами объекты хранятся в отдельном виртуальном массиве, проиндексированного линейными адресами. В принципе без флагов, ссылающихся на объекты можно было бы и обойтись, но тогда бы пришлось при отображении каждой ячейки просматривать все виртуальные массивы на предмет поиска объектов, ассоциированных с данным линейными адресом, что отрицательно сказалось бы на производительности дизассемблера. Напротив, перенос этой информации в флаги позволяет ускорить работу – обращение к виртуальному массиву происходит только в тех случаях, когда с ячейкой заведомо связан какой-то объект. (см. главу «Объекты»)

Размер машинных инструкций и типов данных различен, и это не дает возможности быстро определить к какой инструкции (переменной) принадлежит ячейка с таким-то линейным адресом. Можно провести следующую аналогию – текст со словами не

отделенными друг от друга пробелами из произвольной точки читать невозможно, поскольку неизвестно чем является та или иная буква – началом слова или его серединой.

Образно говоря, при дизассемблировании IDA разбивает исследуемый файл на «слова», в терминологии разработчиков - **элементы**.

Элементом называется последовательность смежных ячеек виртуальной памяти, содержащих одну самостоятельную конструкцию языка ассемблера, с которой можно оперировать как с одним целым, – инструкцию, строку, переменную и т.д.

Первая ячейка элемента называется **головой**, а все, последующие за ней – **хвостом**. Элементы единичной длины состоят только из головы и не имеют хвоста.

Признаком головы является ненулевое значение бита **FF\_DATA** (см. таблицу 10) поля флагов. Соответственно признаком хвоста является нулевое значение бита **FF\_DATA** и ненулевое значение бита **FF\_TAIL**.

Свойства элемента определяются свойствами его головы. Свойства головы определяются флагами, связанными с данной ячейкой виртуальной памяти.

Существуют элементы двух видов – элементы **кода (CODE)** и элементы **данных (DATA)**.

Вид элемента задается сочетанием двух битов FF\_DATA и FF\_TAIL следующим образом (см. таблицу 11) – если бит головы (FF\_DATA) установлен, то значение бита хвоста (FF\_TAIL) интерпретируется типом элемента – его единичное значение задает тип CODE, в противном случае – DATA; напротив, если бит головы сброшен, единичное значение бита FF\_TAIL трактуется признаком хвоста элемента; если же оба бита FF\_DATA и FF\_TAIL сброшены – элемент считается неопределенным (unexplored), т.е. несуществующим.

бит	...	A	9				8		7	...	1	0
	поле флагов										поле содержимого	
обозначение	...	FF_DATA	FF_TAIL			FF_IVL		MS_VAL				
назначение	...	Голова	голова		хвост	==1		==0		содержимое ячейки		
			тип элемента		признак хвоста	инициализированный байт	не инициализированный байт					
			==0	==1				==1				
			DATA	CODE								
маска	...	0x400	0x200				0x100		0xFF			

**Таблица 10 устройство элемента**

		FF_DATA	
		0	1
FF_TAIL	0	неопределенный элемент FF_UNK	элемент данных, голова FF_DATA
	1	хвост элемента FF_FAIL	элемент кода, голова FF_CODE

**Таблица 11 Определение типа элемента**

Хвостовые флаги в двенадцати старших битах содержат смещения относительно начала и конца элемента. Флаги, расположенные по четному линейному адресу – относительно конца, а флаги расположенные по нечетному линейному адресу – относительно его начала.

Это позволяет легко определить к какому элементу принадлежит такая-то ячейка, а так же узнать линейный адрес начала и конца элемента, на основании которых легко вычислить его длину.

**Замечание:** в файле `<INCLUDE\Bytes.hpp>`, входящим в IDA SDK определена функция, возвращающая значение хвостовых бит `"inline ushort gettof(flags_t F) { return ushort((F & TL_TOFF) >> TL_TSFT);}"`

**Внимание:** создавать новые объекты, использовать или изменять хвостовые биты может только ядро IDA, но не пользовательские модули! В противном случае это может привести к некорректной работе и зависанию дизассемблера.

Элементы могут существовать только внутри сегментов – попытка создания элемента по адресу, не принадлежащему ни одному сегменту, обречена на провал.

## Навигатор по функциям

Созданием элементов всецело занимается ядро IDA; пользовательские скрипты хотя и имеют достаточные для «ручного» создания элементов привилегии, пользоваться этими привилегиями категорически не рекомендуется - даже незначительная ошибка способна вызвать непредсказуемое поведения дизассемблера вплоть до его полного «зависания».

Напротив, работая с уже созданными ядром элементами, пользователь может решить ряд задач, облегчающих дизассемблирование исследуемого файла. Пусть, например, требуется отыскать в тексте все условные переходы, стоящие после инструкции `"test"`, следующей за вызовом функции, иначе говоря, произвести поиск по шаблону `"call *test *, *j? *"` и вывести протокол отчета.

Это можно осуществить с помощью следующего листинга, сердцем которого является цикл, вызывающий функцию `NextHead`, последовательно переходящей от одной машинной инструкции к другой:

```
#include <ida.idc>

static main()
{
    auto a;
    a=0;
    while(a!=BADADDR)
    {
        if (isCode(GetFlags(a)))
            if( (GetMnem(a)=="call")
                && (GetMnem(NextHead(a, BADADDR))=="test")
                && (Byte(NextHead(NextHead(a, BADADDR), BADADDR)) > 0x6F)
                && (Byte(NextHead(NextHead(a, BADADDR), BADADDR)) < 0x80))
            Message(">%s %4s %s\n>%s %4s %s,%s\n>%s %s %s\n>-----\n",
                atoa(a), GetMnem(a), GetOpnd(a, 0),
                atoa(NextHead(a, BADADDR)),
                GetMnem(NextHead(a, BADADDR)),
                GetOpnd(NextHead(a, BADADDR), 0),
                GetOpnd(NextHead(a, BADADDR), 1),

                atoa(NextHead(NextHead(a, BADADDR), BADADDR)),
                GetMnem(NextHead(NextHead(a, BADADDR), BADADDR)),
                GetOpnd(NextHead(NextHead(a, BADADDR), BADADDR), 0));
    }
}
```

```

        a=NextHead(a, BADADDR);
    }
}

```

Результат его работы может быть следующим (в данном примере использовался файл first.exe – см. главу «Первые шаги с IDA Pro»)

```

>004010C0 call ostream::opfx(void)
>004010C5 test eax, eax
>004010C7 jz loc_4010E0
>-----
>0040111F call ios::~ios(void)
>00401124 test [esp+4+arg_0], 1
>00401129 jz loc_401132
>-----
>004011BE call ios::~ios(void)
>004011C3 test [esp+4+arg_0], 1
>004011C8 jz loc_4011D1
>-----
...

```

Две функции **NextHead** и **PrevHead** обеспечивают прямую (от младших адресов к старшим) и обратную (от старших к младшим) трассировку элементов, возвращая линейный адрес головы следующего элемента или значение BADADDR если достигнут последний элемент в цепочке.

Это позволяет рассматривать анализируемый код не как поток бессвязных байт, а упорядоченную последовательность инструкций и данных. Разница между ними заключается в том, что поиск байт из интервала 0x70-0x7F в первом случае обнаружит не только все условные переходы, но и множество других инструкций и данных, частью которых является ячейка с таким значением, например, “DW 6675h”; “MOV AX, 74h”; напротив, во втором случае можно быть уверенным, что анализируется именно начало инструкции, а не нечто иное.

Разбивка потока байт на элементы позволяет решить и другую задачу – определить какой именно инструкции (или данным) принадлежит такой-то линейный адрес, т.е. по линейному адресу ячейки определить адрес головы элемента, которому эта ячейка принадлежит.

Манипулируя с флагами узнать это достаточно просто – достаточно проанализировать старшие 12 бит хвостовых атрибутов – флаги, расположенные по четному линейному адресу – содержат количество оставшихся байт до конца элемента, а флаги расположенные по нечетному линейному адресу – до его начала.

Реализация функции, возвращающей адрес головы элемента по любому принадлежащему элементу адресу может выглядеть так:

```

#include <idc.idc>

static MyGetHead(ea)
{
    auto off, F;
    F=GetFlags(ea);
    if (!F) return -1; //Адрес не принадлежит ни одной ячейке
    if (!(F & FF_TAIL)) //Это голова, возвращает ее адрес
        return ea;

    if (ea & 1) // ...нечетный линейный адрес
        return (ea - (F >> 20));
}

```

```

// ...четный линейный адрес
return MyGetHead(ea-1);
}

```

Недостатком такого подхода является его потенциальная несовместимость с последующими версиями IDA Pro, но обойтись для решения проблемы одними лишь высокоуровневыми функциями, встроенными в IDA, не получается.

Задача могла бы быть решена при помощи вызовов функций **ItemSize** и **ItemEnd**, предназначенных для вычисления длины и адреса конца элемента соответственно, очевидно, адрес начала элемента равен  $ItemEnd(ea) - ItemSize(ea)$ , но **ItemSize** возвращает не размер элемента, а смещение до его конца!

Ниже приведен другой вариант реализации функции **MyGetItemHeadEA**, вместо низкоуровневых манипуляций с флагами, использующий вызов **PrevHead**, однако, полностью отказаться от обращений к флагам не удастся – приходится вызывать макрос **isTail**, определенный в файле `<idc.idc>` для проверки не является ли переданный адрес адресом головы элемента.

```

static MyGetItemHeadEA(ea)
{
    if (!GetFlags(ea)) return -1; // ошибка

    if (!isTail(GetFlags(ea)) // голова
        return ea;
    return PrevHead(ea, 0);
}

```

Пара функций **NextNotTail** и **PrevNotTail** возвращают адрес следующего (предыдущего) элемента или бестипового байта. В полностью дизассемблированной программе не должно быть ни одного бестипового байта, но они могут присутствовать на промежуточной стадии анализа – все ячейки, на которые IDA не смогла распознать ни одной ссылки, она оставляет бестиповыми байтами, которые надлежит перевести в элементы кода или данных пользователю.

### Сводная таблица функций

функции, возвращающие основные характеристики элементов	
имя функции	краткое описание
long ItemSize (long ea)	возвращает расстояние до конца элемента (не его размер!)
long ItemEnd (long ea)	возвращает значение на единицу превышающее линейный адрес конца элемента
функции трассировки элементов	
имя функции	краткое описание
long NextHead (long ea)	возвращает линейный адрес следующей головы элемента
long NextHead (long ea, long maxea)	
long PrevHead (long ea)	возвращает линейный адрес предыдущей головы элемента
long PrevHead (long ea, long minea)	
long NextNotTail (long ea)	возвращает линейный адрес следующей головы элемента или неопределенного байта
long PrevNotTail (long ea)	возвращает линейный адрес предыдущей головы элемента или неопределенного байта

### long ItemSize(long ea)

В контекстной помощи IDA сообщается, что эта функция определяет размер элемента, как и следует из ее названия. На самом же деле она возвращает расстояние в байтах до конца элемента, что доказывает следующий эксперимент:

```
seg000:0000 aHelloIdaPro    db 'Hello,IDA Pro!'  
seg000:000E a1234          db '1234'
```

а) исходные данные

```
auto a,b;  
a=SegByName ("seg000");  
for (b=0;b<0x12;b++)  
Message(">ea:%X -%c-> %d\n",a+b,Byte(a+b),ItemSize(a+b));
```

б) скрипт, последовательно передающий функции различные адреса от начала объекта

```
>ea:1000 -H-> 14  
>ea:1001 -e-> 13  
>ea:1002 -l-> 12  
>ea:1003 -l-> 11  
>ea:1004 -o-> 10  
>ea:1005 -, -> 9  
>ea:1006 -I-> 8  
>ea:1007 -D-> 7  
>ea:1008 -A-> 6  
>ea:1009 - -> 5  
>ea:100A -P-> 4  
>ea:100B -r-> 3  
>ea:100C -o-> 2  
>ea:100D -!-> 1  
>ea:100E -1-> 4  
>ea:100F -2-> 3  
>ea:1010 -3-> 2  
>ea:1011 -4-> 1
```

в) результат: функция ItemSize возвращает расстояние до конца объекта в байтах

Минимальное значение, возвращаемое функцией, равно единице. Это же значение возвращается при возникновении ошибки – если функции передать адрес, не принадлежащий ни одному элементу.

Альтернативная реализация функции ItemSize содержится в файле *“kpsc.idc”*, находящимся на диске, прилагаемом к этой книге. Ее исходный код приведен ниже (см. MyGetItemSize)

```
static MyGetItemHeadEA (ea)  
{  
    if (GetFlags(ea) & FF_DATA) // голова  
        return ea;  
    if (GetFlags(ea) & FF_TAIL) // хвост  
        return PrevHead(ea,0);  
    // если не голова и не хвост - ошибка  
    return -1;  
}
```



```

static MyGetItemSize (ea)
{
    if (GetFlags(ea) & MS_CLS) // элемент?
        return ItemEnd(ea) - MyGetItemHeadEA(ea);

    return -1;
}

```

Пример использования:

```

seg000:0000 aHelloIdaPro    db 'Hello,IDA Pro!'
seg000:000E a1234         db '1234'

```

**a) исходные данные**

```

auto a,b;
a=SegByName("seg000");
for(b=0;b<0x12;b++)
Message(">ea:%X -%c-> %d\n",a+b,Byte(a+b),MyGetItemSize(a+b));

```

**b) скрипт, последовательно передающий функции MyGetItemSize различные адреса от начала объекта**

**Замечание:** перед исполнением скрипта файл "krpc.idc" должен быть загружен в память. Это можно осуществить нажатием клавиши <F2>

```

>ea:1000 -H-> 15
>ea:1001 -e-> 15
>ea:1002 -l-> 15
>ea:1003 -l-> 15
>ea:1004 -o-> 15
>ea:1005 -, -> 15
>ea:1006 -I-> 15
>ea:1007 -D-> 15
>ea:1008 -A-> 15
>ea:1009 - -> 15
>ea:100A -P-> 15
>ea:100B -r-> 15
>ea:100C -o-> 15
>ea:100D -!-> 15
>ea:100E -1-> 5
>ea:100F -2-> 5
>ea:1010 -3-> 5
>ea:1011 -4-> 5

```

**c) результат – корректное выполнение функции**

??? #Верстальщику – Change Table

аргумент	пояснение	
ea	линейный адрес, принадлежащий элементу	
return	=return	Пояснения
	!=0	расстояние до конца элемента в байтах (не его размер!)
	==1	ошибка

**Родственные функции:** нет

**Интерактивный аналог:** *нет*

long ItemEnd(long ea)

Функция возвращает значение на единицу превышающее линейный адрес конца элемента, заданного линейным адресом **ea**.

Если переданный адрес не принадлежит ни одному сегменту, функция возвратит единицу, сигнализируя об ошибке.

**Замечание:** *если по указанному адресу расположен бестиповой байт (т.е. переданный адрес не принадлежит ни одному элементу), функция возвратит не ошибку, а линейный адрес следующей ячейки.*

Пример использования:

```
seg000:0000 aHelloIdaPro    db 'Hello,IDA Pro!'
seg000:000E a1234        db '1234'
```

а) исходные данные

```
Message(">%s\n", atoa( ItemEnd( SegByName("seg000") ) ) );
```

б) вызываем функцию `ItemEnd`, передавая ей адрес, принадлежащий элементу "Hello, IDA Pro!".

```
>seg000:000E
```

с) результат – функция вернула значение, на единицу превышающее адрес конца данного элемента

??? #Верстальщику Change Table

аргумент	пояснение	
<code>ea</code>	линейный адрес, принадлежащий элементу или бестиповому байту	
<code>return</code>	<code>=return</code>	пояснения
	<code>!=1</code>	значение на единицу превышающее линейный адрес конца элемента
	<code>==1</code>	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** *нет*

long NextHead(long ea)

(версия IDA 3.85 и младше)

Возвращает линейный адрес головы следующего элемента, в противном случае – `BADADDR`, сигнализируя об ошибке. Переданный функции линейный адрес **ea** не обязательно должен принадлежать какому-то элементу – он может даже вообще не существовать.

Пример использования:

```
seg000:0000 aHelloIdaPro    db 'Hello,IDA Pro!'
seg000:000E a1234        db '1234'
```

а) исходные данные

```
Message(">%s\n", atoa(NextHead(SegByName("seg000"))));
```

b) вызываем функцию NextHead, передавая ей адрес, принадлежащий элементу "Hello, IDA Pro!".

```
>seg000:000E
```

с) результат – функция вернула адрес головы следующего элемента.

Другой пример использования функции NextHead приведен в файле <href.idc>, распространяемого вместе с IDA.

**Замечание:** линейный адрес начала следующего элемента будет возвращен даже в том случае, если элемент находится в другом сегменте.

??? #Верстальщику – Change Table

аргумент	пояснение	
ea	линейный адрес, не обязательно принадлежащий какому-то элементу	
return	=return	пояснения
	!=BADADDR	линейный адрес головы следующего элемента
	==BADADDR	ошибка

**Родственные функции:** PrevHead

**Интерактивный аналог:** *нет*

long NextHead(long ea, long maxea)

(версия IDA 4.0 и старше)

В версии 4.0 прототип функции NextHead(long ea) (см. ее описание) был изменен, добавлением еще одного аргумента – **maxea**, ограничивающего диапазон адресов, доступных функции.

Функция возвращает адрес головы следующего элемента, при условии, что он меньше, чем maxea, т.е. отвечает следующему условию  $ea < return\ value < maxea$ .

Изменение прототипа повлекло за собой неработоспособность всех ранее созданных скриптов, использующих эту функцию и необходимости внесения в них исправлений – замены NextHead(ea) на NextHead(ea, BADADDR).

**Замечание:** ограничение максимального адреса облегчает написание скриптов, работающих с выделенными регионами (см. описание функций SelStart и SelEnd)

??? #Верстальщику – Change Table

аргумент	пояснение	
ea	линейный адрес, не обязательно принадлежащий какому-то элементу	
maxea	значение на единицу превышающее наибольший адрес, доступный функции	
return	=return	пояснения
	!=BADADDR	линейный адрес головы следующего элемента
	==BADADDR	ошибка

**Родственные функции:** PrevHead

**Интерактивный аналог:** *нет*

long PrevHead(long ea)

(версия IDA 3.85 и младше)

Функция возвращает линейный адрес предыдущей головы элемента (не головы предыдущего элемента!). Если указать на хвост элемента, функция возвратит адрес его головы. Переданный функции линейный адрес **ea** не обязательно должен принадлежать какому-то элементу – он может даже вообще не существовать.

Пример использования:

```
seg000:0000 aHelloIdaPro      db 'Hello,IDA Pro!'  
seg000:000E a1234          db '1234'
```

а) исходные данные

```
Message(">%s\n", atoa(PrevHead(SegByName("seg000")+0x2)));
```

б) вызываем функцию PrevHead, передавая ей адрес принадлежащий элементу "Hello, IDA Pro!".

```
>seg000:000E
```

с) результат – функция вернула адрес предыдущей головы элемента (этого элемента)

??? #Верстальщику – Change table

аргумент	пояснение	
ea	линейный адрес, не обязательно принадлежащий какому-то элементу	
return	=return	пояснения
	!=BADADDR	линейный адрес предыдущей головы элемента (не головы предыдущего элемента!)
	==BADADDR	ошибка

**Родственные функции:** NextHead

**Интерактивный аналог:** *нет*

long PrevHead(long ea, long minea)

(версия IDA 4.0 и старше)

В версии 4.0 прототип функции PrevHead(long ea) (см. ее описание) был изменен, добавлением еще одного аргумента – **minea**, ограничивающего диапазон адресов, доступных функции.

Функция возвращает адрес предыдущей головы элемента, при условии, что он не меньше, чем minea, т.е. отвечает следующему условию  $minea \geq return\ value > ea$ .

Изменение прототипа повлекло за собой неработоспособность всех ранее созданных скриптов, использующих эту функцию и необходимости внесения в них исправлений – замены PrevHead(ea) на PrevHead(ea, 0).

**Замечание:** ограничение максимального адреса облегчает написание скриптов, работающих с выделенными регионами (см. описание функций SelStart и SelEnd)

??? #Верстальщику – Change Table

аргумент	пояснение	
ea	линейный адрес, не обязательно принадлежащий какому-то элементу	
minea	минимальный адрес, доступный функции	
return	=return	пояснение
	!=BADADDR	линейный адрес предыдущей головы элемента (не головы предыдущего элемента!)
	==BADADDR	ошибка

**Родственные функции:** NextHead

**Интерактивный аналог:** *нет*

long NextNotTail(long ea)

Функция возвращает линейный адрес головы следующего элемента или бестипового байта. Переданный функции линейный адрес **ea** не обязательно должен принадлежать какому-то элементу – он может даже вообще не существовать.

Пример использования:

```
seg000:0000 aHelloIdaPro    db 'Hello,IDA Pro!'
seg000:000E a1234                db '1234'
```

a) исходные данные

```
Message(">%s\n", atoa(NextNotTail(0)));
```

b) вызываем функцию NextNotTail, передавая ей нулевой адрес.

```
>seg000:0000
```

c) результат – функция вернула адрес головы первого элемента

??? #Верстальщику – Change table

аргумент	пояснение	
ea	линейный адрес, не обязательно принадлежащий какому-то элементу	
return	=return	пояснения
	!=BADADDR	линейный адрес головы следующего элемента или бестипового байта
	==BADADDR	ошибка

**Родственные функции:** PrevNotTail

**Интерактивный аналог:** *нет*

long PrevNotTail(long ea)

Функция возвращает линейный адрес головы предыдущего элемента (не предыдущей головы элемента!). Переданный функции линейный адрес **ea** не обязательно должен принадлежать какому-то элементу – он может даже вообще не существовать.

Пример использования:

```
seg000:0000 aHelloIdaPro db 'Hello,IDA Pro!'
seg000:000E a1234 db '1234'
```

а) исходные данные

```
Message(">%s\n", atoa(NextNotTail(BADADDR)));
```

б) вызываем функцию PrevNotTail, передавая ей значение BADADDR

```
>seg000:000E
```

с) результат – функция вернула адрес головы самого последнего из существующих элемента

**Замечание:** в отличие от функции NextNotTail, функция PrevNotTail игнорирует бестиповые байты.

??? #Верстальщику – Change table

аргумент	пояснение	
ea	линейный адрес, не обязательно принадлежащий какому-то элементу	
return	==return	пояснения
	!=BADADDR	линейный адрес головы предыдущего элемента
	==BADADDR	ошибка

**Родственные функции:** PrevNotTail

**Интерактивный аналог:** *нет*

## ТИПЫ ЭЛЕМЕНТОВ

### #Definition

Поле флагов головы элемента определяет является ли данный элемент элементом кода или элементом данных (см. главу «Элементы»), а так же уточняет его тип. Например, один и та же цепочка из четырех байт может быть двойным словом, типом float, ASCII-строкой, массивом байт или двойных слов и т.д.

**Замечание:** типотизация данных – одно из немаловажных достоинств IDA Pro, ощутимо повышающее качество дизассемблирования. В полностью дизассемблированной программе не должно быть ни одного байта данных неопределенного типа.

IDA Pro поддерживает следующие типы данных – **байт, слово, двойное слово, четвертное слово, восьмерное слово, float, double, packed real, ASCII-строка, массив**, состоящих из любых вышеперечисленных типов, а так же тип **align** – байты, использующиеся для выравнивания кода (данных) по кратным адресам (см. таблицу 12)

??? #Верстальщику – change table

константа	#	тип
FF_BYTE	0x00000000	байт
FF_WORD	0x10000000	слово
FF_DWRD	0x20000000	двойное слово
FF_QWRD	0x30000000	четвертное слово
FF_TBYT	0x40000000	восьмерное слово

FF_ASCII	0x50000000	ASCII-строка
FF_STRU	0x60000000	структура
FF_XTRN	0x70000000	внешние данные неизвестного размера
FF_FLOAT	0x80000000	float
FF_DOUBLE	0x90000000	double
FF_PACKREAL	0xA0000000	упакованное десятичное целое
FF_ALIGN	0xB0000000	директива выравнивания

**Таблица 12 поддерживаемые типы данных**

Цепочка бестиповых байт может быть преобразована в любой поддерживаемый IDA Pro тип данных, при условии что имеет достаточный для такой операции размер. Уже существующий элемент данных, также может быть преобразован в данные другого типа, если имеет достаточный для такой операции размер, либо за его хвостом следует цепочка бестиповых байт необходимой длины.

Если в результате преобразования, размер элемента уменьшается, его остаток преобразуется в один или несколько бестиповых байт.

Примеры:

```

seg000:0000 Var          db  48h ; H
seg000:0001              db  65h ; e
seg000:0002              db  6Ch ; l
seg000:0003              db  6Ch ; l
seg000:0004              db  6Fh ; o
seg000:0005 Var2        db  2Ch
seg000:0006              db  20h ;
seg000:0007              db  49h ; I
seg000:0008              db  44h ; D
seg000:0009              db  41h ; A
seg000:000A              db  20h ;

```

Бестиповая переменная “Var” может быть преобразована в байт, слово, двойное слово, float, ASCII-строку, но попытка преобразовать ее в четверное, восьмерное слово, double, packed real приведет к ошибке – поскольку тому препятствует элемент данных, расположенный по адресу “seg000:0005”. Если же его уничтожить, преобразование пройдет успешно. Аналогично:

```

seg000:0000 Var          dw  6548h
seg000:0002              db  6Ch
seg000:0003              db  6Ch
seg000:0000 Var          dw  6548h
seg000:0002              dw  6C6Ch

```

а) преобразование Var в двойное слово возможно

б) преобразование Var в двойное слово невозможно – требуется предварительно уничтожить следующий за ним элемент

Два и более последовательных элементов одного типа могут быть объединены в массив – как бы *макроэлемент*, собирающий их всех под одну крышу. С массивом IDA Pro работает как с единым целым, в частности, в начале каждой строки указывает не адрес текущего элемента, а адрес начала массива:

```

seg000:0000              db  6Ch, 6Fh, 2Ch, 20h, 49h, 44h, 41h, 20h, 50h, 72h, 6Fh
seg000:0000              db  21h, 0

```

Функции трассировки элементов (см. главу «Элементы») будут возвращать только адреса начала массива и следующего за массивом элемента, но не адреса элементов самого массива! Аналогично, функции изменения представления операндов (см. главу

«Операнды») будут изменять представление всех элементов массива одновременно, но никак не выборочно.

Поэтому, разумно объединять в массив данные лишь тогда, когда они имеют одинаковый тип, одинаковое представление и нет никакой необходимости в индивидуальной работе ни с одним элементом. В противном случае создание массива доставит больше неудобств, чем пользы.

Важно понять – массив в IDA Pro это **один** элемент, а не совокупность множества элементов другого типа.

Максимальная длина элемента ограничена четырьмя килобайтами – это связано с тем, что под хвостовые биты, содержащие смещения относительно начала (конца) элемента отведено 12 разрядов, отсюда и ограничение на длину.

Если требуется создать строку или массив большего размера, единственный выход – создать два (или более) массивов (строк), расположив их последовательно друг за другом.

Элемент данных может иметь тип `align` – указывающий, что эти байты используются для выравнивания кода (данных) по кратным адресам. Формально тип `align` – такой же точно тип как, например, слово или двойное слово, и с ним можно выполнять точно те же операции, что и над любым другим элементом данных. Единственное его отличие от состоит в том, что принадлежащие ему байты, в ассемблерный листинг не попадают:

```
seg000:0000 db 48h ; H      seg000:0000      db 48h ; H
seg000:0001 db 65h ; e      seg000:0001      align 2
seg000:0002 db 6Ch ; l      seg000:0002      db 6Ch ; l
```

Элемент кода не имеет никаких типов, разрядность инструкций определяется разрядностью сегмента, а логика работы дизассемблера – типом выбранного процессора.

Элемент данных не может быть непосредственно преобразован в элемент кода и, соответственно, наоборот. Элемент кода может быть создан только из цепочки бестиповых байт достаточной длины.

Если после создания элемента кода, IDA может определить адрес следующей выполняемой инструкции, она автоматически пытается создать в соответствующем месте очередной элемент кода – так продолжается до тех пор, пока ей не встретится инструкция передающее управление по адресу, который IDA вычислить не в состоянии. Это может быть, например, регистровый переход, команда выхода из подпрограммы (прерывания) и т.д. Возможности создания одного элемента кода в IDA Pro нет.

Строго говоря, элементы кода и данных не являются неделимым целым – IDA Pro предоставляет возможность выборочной работы с их операндами (если они есть) – см. главу «Операнды».

## Навигатор по функциям

Группа функций **MakeByte**, **MakeWord**, **MakeDword**, **MakeQword**, **MakeFloat**, **MakeDouble**, **MakePackedReal**, **MakeTbyte** предназначена для преобразования цепочки бестиповых байт (или уже существующего элемента данных) в элемент данных типа байт, слово, двойное слово, четвертное слово, `float`, `double`, `PackedReal` и восьмерное слово соответственно.

Интерактивный аналог первых трех функций пункт “Data” меню “~Edi” (или «горячая клавиша “D”>», циклично преобразующий тип элемента, находящегося под курсором в байт, слово и двойное слово. При необходимости в эту последовательность можно включить и другие типы данных, вызвав диалог “Setup data types” из меню “Options” («горячая клавиша – “Alt-D”»).

Функция **MakeStr** преобразует цепочку бестиповых байт в ASCII-строку заданной длины или попытается определить ее автоматически. Автоматически распознаются длины следующих типов строк– **ASCIIZ**-строк, заканчивающихся символом нуля; **PASCAL**-строк,



начинающихся с байта, содержащего длину строки и **DELPHI**-строк, начинающиеся со слова (двойного слова), содержащего длину строки. Если строка не принадлежит ни к одному из этих трех типов, концом строки считается:

- a) первый нечитабельный ASCII-символ.
- b) неинициализированный байт
- c) голова элемента кода или данных
- d) конец сегмента

Функция **MakeArray** создает массив состоящий из данных одного типа – байтов, слов, двойных слов, четверных слов, двойных слов в формате float, четверных слов в формате double, packed real, tbyte. Бестиповые байты могут стать частью массива любого типа. Строки не могут быть элементами никакого массива.

Тип массива определяется типом его первого элемента. Все остальные элементы массива на момент его создания должны быть представлены бестиповыми байтами, - последовательность типизированных данных не может быть преобразована в массив.

Элементы массива записываются в строку, отделяясь друг от друга знаком запятой. Если два или более подряд идущих элемента имеют одно и то же значение (в том числе и неинициализированное) для сокращения ассемблерного листинга используется конструкция "DUP".

Функция **Align** помещает в ассемблерный файл директиву выравнивания **align** и исключает из дизассемблируемого листинга байты, используемые для выравнивания.

Функция **MakeCode** создает по указанному адресу элемент кода, выполняя дизассемблирование первой машинной инструкции. Если это возможно, автоматически дизассемблируется и другие инструкции. Это происходит в следующих случаях:

- a) текущая инструкция не изменяет нормального выполнения программы и за ее концом расположены бестиповые байты;
- b) текущая инструкция изменяет нормальное выполнение программы, осуществляя переход по непосредственному адресу, тогда IDA продолжит дизассемблирование с этого адреса

Если встречается инструкция, изменяющая адрес перехода непредсказуемым образом (например, RET) IDA прекращает дизассемблирование.

Во время дизассемблирования IDA при необходимости создает перекрестные ссылки и автогенерируемые метки.

Функция **MakeUnkn** разрушает элемент, заданный любым принадлежащим ему адресом, превращая его содержимое в бестиповые байты. Объекты, связанные с элементом (например, метки, комментарии) при этом не уничтожаются.

## Сводная таблица функций

функции создания новых элементов, преобразования и уничтожения элементов	
имя функции	каткое описание
success MakeByte(long ea)	создает (преобразует) ячейку в байт
success MakeWord(long ea)	создает (преобразует) ячейку в слово (2 байта)
success MakeDword(long ea)	создает (преобразует) ячейку в двойное слово (4 байта)
success MakeQword(long ea)	создает (преобразует) ячейку в четвертное слово (8 байт)
success MakeFloat(long ea)	создает (преобразует) ячейку в тип float (представление с плавающей запятой 4 байта)
success MakeDouble(long ea)	создает (преобразует) ячейку в тип Double (представление с плавающей запятой 8 байт)
success MakePackReal(long ea)	создает (преобразует) ячейку в тип PackReal (от 10 до 12 байт)

success MakeTbyte(long ea)	создает (преобразует) ячейку в тип Tbyte (10 байт)
success MakeStr (long ea,long endea)	создает ASCII строку
success MakeArray (long ea,long nitems)	создает массив
success MakeAlign(long ea,long count,long align)	создает директиву выравнивания
long MakeCode(long ea)	дизассемблирует одну (или больше) инструкций
void MakeUnkn (long ea,long expand);	уничтожает элемент
<b>функции возвращающие свойства элементов</b>	
имя функции	краткое описание
char GetMnem (long ea)	возвращает мнемонику инструкции в символьном виде
<b>функции, поиска элементов</b>	
имя функции	краткое описание
long FindCode(long ea, long flag)	возвращает линейный адрес ближайшего элемента кода
long FindData(long ea,long flag)	возвращает линейный адрес ближайшего элемента данных
long FindUnexplored(long ea,long flag)	возвращает линейный адрес ближайшего бестипового байта
long FindExplored(long ea, long flag);	возвращает линейный адрес ближайшего элемента

### success MakeByte(long ea)

Функция создает по переданному ей линейному адресу **ea** элемент данных типа **байт**.

Если по данному адресу находится голова ранее созданного элемента данных, функция преобразует его в байт, а хвост элемента (если он есть) – в бестиповые байты.

Если по данному адресу находится хвост элемента данных, голова или хвост элемента кода, функция возвратит ошибку.

Наличие неинициализированных байт в создаваемом или преобразуемом элементе не является препятствием для выполнения этой функции.

Пример использования:

#### **1. эксперимент**

```
seg000:0000 db ? ; unexplored
```

а) исходные данные

```
Message (">%x\n", MakeByte (SegByName ("seg000")) );
```

б) вызываем функцию MakeByte для создания нового элемента данных типа байт, передавая ей адрес бестипового байта

```
seg000:0000 db ?
```

```
>1
```

с) результат – элемент данных типа байт успешно создан

**Замечание:** в ассемблерном листинге бестиповые байты помечаются комментарием “unexplored” (или ASCII кодом содержимого), сигнализирующим о неисследованности данной ячейки. В полностью дизассемблированной программе не должно остаться ни одного неисследованного байта – тип каждой ячейки должен быть задан явно.

## 2. эксперимент

```
seg000:0000 aHelloSailor db 'Hello, Sailor'
```

а) исходные данные

```
Message(">%x\n", MakeByte (SegByName ("seg000")));
```

б) вызываем функцию MakeByte для преобразования элемента типа «строка» в элемент типа «байт», передавая ей линейный адрес головы данного элемента

```
seg000:0000 aHelloSailor db 48h
seg000:0001 db 65h ; e
seg000:0002 db 6Ch ; l
seg000:0003 db 6Ch ; l
seg000:0004 db 6Fh ; o
seg000:0005 db 2Ch ; ,
seg000:0006 db 20h ;
seg000:0007 db 53h ; S
seg000:0008 db 61h ; a
seg000:0009 db 69h ; i
seg000:000A db 6Ch ; l
seg000:000B db 6Fh ; o
seg000:000C db 72h ; r
seg000:000D db 66h ; f
>1
```

с) результат- успешное преобразование; хвост элемента типа строка преобразован в бестиповые байты.

## 3. эксперимент

```
seg000:0000 aHelloSailor db 'Hello, Sailor'
```

а) исходные данные

```
Message(">%x\n", MakeByte (1+SegByName ("seg000")));
```

б) вызываем функцию MakeByte для преобразования ячейки, принадлежащей хвосту элемента данных типа «строка», в элемент типа «байт».

```
seg000:0000 aHelloSailor db 'Hello, Sailor'
>0
```

с) результат – функция возвратила ошибку, не выполнив преобразования

## 4. эксперимент

```
seg000:0000 PUSH AX
```

а) исходные данные

```
Message(">%x\n", MakeByte (1+SegByName ("seg000")));
```

б) вызываем функцию MakeByte для преобразования элемента кода в элемент данных типа байт

```
seg000:0000 PUSH AX
>0
```

с) результат – функция возвратила ошибку, не выполнив преобразования

??? #Верстальщику – change table

аргумент	пояснения
ea	линейный адрес бестипового байта или головы элемента данных

return	=return	пояснения
	==1	успешное завершение
	==0	ошибка

**Родственные функции:** MakeWord, MakeDword, MakeQword, MakeFloat, MakeDouble, MakePackReal, MakeTbyte.

**Интерактивный аналог:** “~Edit\Data”; <D>

### success MakeWord(long ea)

Функция создает по переданному ей линейному адресу **ea** элемент данных типа **слово**, длиной два байта. Порядок следования младших и старших байт зависит от выбранного процессора. На микропроцессорах серии Intel 80x86 младший байт располагается по меньшему адресу и, соответственно, наоборот.

Если по данному адресу находится голова ранее созданного элемента данных, функция преобразует его в слово, а хвост элемента (если он есть) – в бестиповые байты. Если размер элемента недостаточен для преобразования, но следом за его хвостом находятся бестиповые байты, они будут автоматически присоединены к новому элементу. В противном случае (если размер элемента недостаточен для преобразования, а следом за его хвостом находится другой элемент не находится ничего) функция возвратит ошибку не выполнив преобразования. Для выполнения такого преобразования необходимо предварительно уничтожить мешающие элементы вызовом MakeUnkn (см. описание функции MakeUnkn)

Ошибка возвратится и в том случае если по переданному функции линейному адресу находится хвост элемента данных, голова или хвост элемента кода.

Если хотя бы один из двух байт имеет неинициализированное значение, все слово приобретает неинициализированное значение.

Пример использования:

#### 1. эксперимент

```
seg000:0000          db ? ; unexplored
seg000:0001          db ? ; unexplored
```

а) исходные данные

```
Message (“>%x\n”, MakeWord (SegByName (“seg000”)));
```

б) вызываем функцию MakeWord для создания нового элемента данных типа слово, передавая ей адрес цепочки из двух бестиповых байта

```
seg000:0000          dw ?
>1
```

с) результат – элемент данных типа слово успешно создан

#### 2. эксперимент

```
seg000:0000          db ? ; unexplored
seg000:0001          db ?
```

а) исходные данные

```
Message (“>%x\n”, MakeWord (SegByName (“seg000”)));
```

б) вызываем функцию MakeWord для создания нового элемента данных типа слово, передавая ей адрес бестипового байта

```
seg000:0000          db ? ; unexplored
seg000:0001          db ?
```

>0

с) результат – функция завершилась с ошибкой, т.к. для создания нового элемента не достаточно места – по адресу seg000:0001 находится элемент данных типа байт, который не может быть автоматически присоединен к слову. Для выполнения преобразования его необходимо уничтожить вызовом MakeUnkn

```
MakeUnkn (SegByName ("seg000")+1, 0);
```

d) вызываем функцию MakeUnkn для удаления элемента данных, расположенного по адресу “seg000:0001”

```
seg000:0000          db ? ; unexplored
seg000:0001          db ? ; unexplored
```

e) результат – элемент данных успешно разрушен.

```
Message ("%x\n", MakeWord (SegByName ("seg000")));
```

f) повторно вызываем функцию MakeWord, на этот раз передавая ей адрес цепочки из двух бестиповых байт

```
seg000:0000          dw ?
>1
```

g) результат – элемент данных типа слово успешно создан

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес бестипового байта или головы элемента данных	
return	=return	пояснения
	==1	успешное завершение
	==0	ошибка

**Родственные функции:** MakeByte, MakeDword, MakeQword, MakeFloat, MakeDouble, MakePackReal, MakeTbyte.

**Интерактивный аналог:** “~Edit\Data”; <D>

success MakeDword(long ea)

Функция создает по переданному ей линейному адресу **ea** элемент данных типа **двойное слово**, длиной четыре байта. Порядок следования младших и старших байт зависит от выбранного процессора. На микропроцессорах серии Intel 80x86 младший байт располагается по меньшему адресу и, соответственно, наоборот.

Если по данному адресу находится голова ранее созданного элемента данных, функция преобразует его в двойное слово, а хвост элемента (если он есть) – в бестиповые байты. Если размер элемента недостаточен для преобразования, но следом за его хвостом находятся бестиповые байты, они будут автоматически присоединены к новому элементу. В противном случае (если следом за его хвостом находится другой элемент не находится ничего) функция возвратит ошибку не выполнив преобразования. Для выполнения такого преобразования необходимо предварительно уничтожить мешающие элементы вызовом MakeUnkn (см. описание функции MakeUnkn)

Ошибка возвратится и в том случае если по переданному функции линейному адресу находится хвост элемента данных, голова или хвост элемента кода.

Если хотя бы один из двух байт имеет неинициализированное значение, все двойное слово приобретает неинициализированное значение.

Пример использования:

## 1. эксперимент

```
seg000:0000      db ? ; unexplored
seg000:0001      db ? ; unexplored
seg000:0002      db ? ; unexplored
seg000:0003      db ? ; unexplored
```

a) исходные данные

```
Message(">%x\n", MakeDword(SegByName("seg000")));
```

b) вызываем функцию MakeDword для создания нового элемента данных типа двойное слово, передавая ей адрес цепочки из четырех бестиповых байта

```
seg000:0000      dd ?
>1
```

c) результат – элемент данных типа двойное слово успешно создан

## 2. эксперимент

```
seg000:0000      db ? ; unexplored
seg000:0001      db ? ; unexplored
seg000:0002      dw ?
```

a) исходные данные

```
Message(">%x\n", MakeDword(SegByName("seg000")));
```

b) вызываем функцию MakeDword для создания нового элемента данных типа двойное слово, передавая ей адрес бестипового байта

```
seg000:0000      db ? ; unexplored
seg000:0001      db ? ; unexplored
seg000:0002      dw ?
>0
```

c) результат – функция завершилась с ошибкой, т.к. для создания нового элемента не достаточно места – по адресу seg000:0002 находится элемент данных типа слово, который не может быть автоматически присоединен к двойному слову. Для выполнения преобразования его необходимо уничтожить вызовом MakeUnkn

```
MakeUnkn(SegByName("seg000")+2, 0);
```

d) вызываем функцию MakeUnkn для удаления элемента данных, расположенного по адресу "seg000:0002"

```
seg000:0000      db ? ; unexplored
seg000:0001      db ? ; unexplored
seg000:0002      db ? ; unexplored
seg000:0003      db ? ; unexplored
```

e) результат – элемент данных успешно разрушен.

```
Message(">%x\n", MakeDword(SegByName("seg000")));
```

f) повторно вызываем функцию MakeDword, на этот раз передавая ей адрес цепочки из четырех бестиповых байт

```
seg000:0000      dd ?
>1
```

g) результат – элемент данных типа слово успешно создан

??? #Верстальщику – change table

аргумент	пояснения
----------	-----------

ea	линейный адрес бестипового байта или головы элемента данных	
return	=return	пояснения
	==1	успешное завершение
	==0	ошибка

**Родственные функции:** MakeByte, MakeWord, MakeQword, MakeFloat, MakeDouble, MakePackReal, MakeTbyte.

**Интерактивный аналог:** “~Edit\Data”; <D>

success MakeQword(long ea)

Функция создает по переданному ей линейному адресу **ea** элемент данных типа **четвертное слово**, длиной восемь байт. Порядок следования младших и старших байт зависит от выбранного процессора. На микропроцессорах серии Intel 80x86 младший байт располагается по меньшему адресу и, соответственно, наоборот.

Если по данному адресу находится голова ранее созданного элемента данных, функция преобразует его в четвертное слово, а хвост элемента (если он есть) – в бестиповые байты. Если размер элемента недостаточен для преобразования, но следом за его хвостом находятся бестиповые байты, они будут автоматически присоединены к новому элементу. В противном случае, если следом за его хвостом находится другой элемент или не находится ничего, функция возвратит ошибку, не выполнив преобразования. Для выполнения преобразования необходимо предварительно уничтожить мешающие элементы вызовом MakeUnkn (см. описание функции MakeUnkn).

Ошибка возвратится и в том случае если по переданному функции линейному адресу находится хвост элемента данных, голова или хвост элемента кода.

Если хотя бы один из двух байт имеет неинициализированное значение, все двойное слово приобретает неинициализированное значение.

Пример использования:

```
seg000:0000          db ? ; unexplored
seg000:0001          db ? ; unexplored
seg000:0002          db ? ; unexplored
seg000:0003          db ? ; unexplored
seg000:0004          db ? ; unexplored
seg000:0005          db ? ; unexplored
seg000:0006          db ? ; unexplored
seg000:0007          db ? ; unexplored
```

a) исходные данные

```
Message (“>%x\n”, MakeQword (SegByName (“seg000”)) );
```

b) вызываем функцию MakeQword для создания нового элемента данных типа четвертное слово, передавая ей адрес цепочки из восьми бестиповых байта

```
seg000:0000          dq ?
>1
```

c) результат – элемент данных типа четвертное слово успешно создан

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес бестипового байта или головы элемента данных	
return	=return	пояснения
	==1	успешное завершение

	==0	ошибка
--	-----	--------

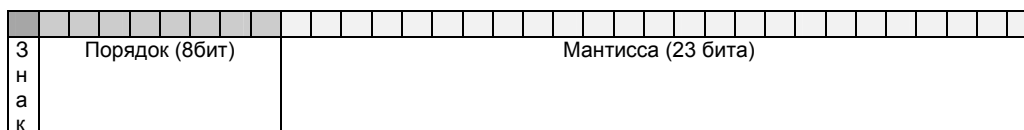
**Родственные функции:** MakeByte, MakeWord, MakeDword, MakeFloat, MakeDouble, MakePackReal, MakeTbyte.

**Интерактивный аналог:** (“~Options\Setup data types”; <Alt-D>), <Q>

**Замечание:** для включения типа «четвертного слова» в список типов данных, пролистываемых нажатием клавиши <D>, необходимо, вызвав диалог “Setup data types” установить галочку напротив пункта “Quadro word”.

### success MakeFloat(long ea)

Функция создает по переданному ей линейному адресу **ea** элемент данных типа **float**, длиной четыре байта. Представление типа float завис от выбранного процессора. На микропроцессорах серии Intel 80x86 он имеет следующее строение (см. рисунок ???)



**Рисунок 23 Представление типа float на микропроцессорах серии Intel 80x86**

Если по данному адресу находится голова ранее созданного элемента данных, функция преобразует его в двойное слово типа float, а хвост элемента (если он есть) – в бестиповые байты. Если размер элемента недостаточен для преобразования, но следом за его хвостом находятся бестиповые байты, они будут автоматически присоединены к новому элементу. В противном случае, если следом за его хвостом находится другой элемент или не находится ничего, функция возвратит ошибку, не выполнив преобразования. Для выполнения преобразования необходимо предварительно уничтожить мешающие элементы вызовом MakeUnkn (см. описание функции MakeUnkn).

Ошибка возвратится и в том случае если по переданному функции линейному адресу находится хвост элемента данных, голова или хвост элемента кода.

Если хотя бы один из четырех байт имеет неинициализированное значение, все двойное слово приобретает неинициализированное значение.

Пример использования:

```
seg000:0000      db  48h ; H
seg000:0001      db  65h ; e
seg000:0002      db  6Ch ; l
seg000:0003      db  6Ch ; l
seg000:0004      db  6Fh ; o
```

а) исходные данные

```
Message(">%x\n", MakeFloat (SegByName ("seg000"))) ;
```

б) вызываем функцию MakeFloat для создания нового элемента данных типа float, передавая ей адрес цепочки из четырех бестиповых байта

```
seg000:0000      dd  1.1431391e27
>1
```

с) результат – элемент данных типа float успешно создан

??? #Верстальщику – change table



аргумент	пояснения	
ea	линейный адрес бестипового байта или головы элемента данных	
return	=return	пояснения
	==1	успешное завершение
	==0	ошибка

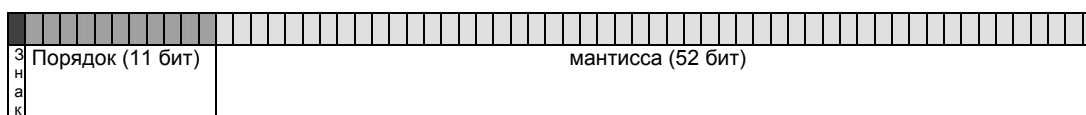
**Родственные функции:** MakeByte, MakeWord, MakeDword, MakeQword, MakeDouble, MakePackReal, MakeTbyte.

**Интерактивный аналог:** (“~Options\Setup data types”; <Alt-D>), <F>

**Замечание:** для включения типа «четвертного слова» в список типов данных, пролистываемых нажатием клавиши <D>, необходимо, вызвав диалог “Setup data types” установить галочку напротив пункта “Float”.

success MakeDouble(long ea)

Функция создает по переданному ей линейному адресу **ea** элемент данных типа **double**, длиной восемь байт. Представление типа double зависит от выбранного процессора. На микропроцессорах серии Intel 80x86 он имеет следующее строение (см. рисунок ???)



**Рисунок 24** Представление типа double на микропроцессорах серии Intel 80x86

Если по данному адресу находится голова ранее созданного элемента данных, функция преобразует его в четвертное слово типа double, а хвост элемента (если он есть) – в бестиповые байты. Если размер элемента недостаточен для преобразования, но следом за его хвостом находятся бестиповые байты, они будут автоматически присоединены к новому элементу. В противном случае, если следом за его хвостом находится другой элемент или не находится ничего, функция возвратит ошибку, не выполнив преобразования. Для выполнения преобразования необходимо предварительно уничтожить мешающие элементы вызовом MakeUnkn (см. описание функции MakeUnkn).

Ошибка возвратится и в том случае если по переданному функции линейному адресу находится хвост элемента данных, голова или хвост элемента кода.

Если хотя бы один из восьми байт имеет неинициализированное значение, все четверное слово приобретает неинициализированное значение.

Пример использования:

```

seg000:0000          db  48h ; H
seg000:0001          db  65h ; e
seg000:0002          db  6Ch ; l
seg000:0003          db  6Ch ; l
seg000:0004          db  6Fh ; o
seg000:0005          db  2Ch ; ,
seg000:0006          db  20h ;
seg000:0007          db  53h ; S

```

а) исходные данные

```
Message (">%x\n", MakeDouble (SegByName ("seg000")) );
```

b) вызываем функцию MakeDouble для создания нового элемента данных типа double, передавая ей адрес цепочки из восьми бестиповых байта

```
seg000:0000                dq 2.635692361932979e92
>1
```

c) результат – элемент данных типа double успешно создан

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес бестипового байта или головы элемента данных	
return	=return	пояснения
	==1	успешное завершение
	==0	ошибка

**Родственные функции:** MakeByte, MakeWord, MakeDword, MakeQword, MakeQword, MakePackReal, MakeTbyte.

**Интерактивный аналог:** (“~Options\Setup data types”; <Alt-D>), <u>

**Замечание:** для включения типа «четвертного слова» в список типов данных, пролистываемых нажатием клавиши <D>, необходимо, вызвав диалог “Setup data types” установить галочку напротив пункта “Double”.

success MakePackReal(long ea)

Функция создает по переданному ей линейному адресу **ea** элемент данных типа **packed real**, занимающий в результате от обстоятельств от десяти до двенадцати байт.

Если по данному адресу находится голова ранее созданного элемента данных, функция преобразует его в packed real, а хвост элемента (если он есть) – в бестиповые байты. Если размер элемента недостаточен для преобразования, но следом за его хвостом находятся бестиповые байты, они будут автоматически присоединены к новому элементу. В противном случае, если следом за его хвостом находится другой элемент или не находится ничего, функция возвратит ошибку, не выполнив преобразования. Для выполнения преобразования необходимо предварительно уничтожить мешающие элементы вызовом MakeUnkn (см. описание функции MakeUnkn).

Ошибка возвратится и в том случае если по переданному функции линейному адресу находится хвост элемента данных, голова или хвост элемента кода.

Если один или более байт имеют неинициализированное значение, они никак не влияют на содержимое остальных и вся цепочка байт packed real *не принимает* неинициализированного значения.

Пример использования:

```
seg000:0000                db ? ; unexplored
seg000:0001                db ? ; unexplored
seg000:0002                db ? ; unexplored
seg000:0003                db ? ; unexplored
seg000:0004                db ? ; unexplored
seg000:0005                db ? ; unexplored
seg000:0006                db ? ; unexplored
seg000:0007                db ? ; unexplored
seg000:0008                db ? ; unexplored
seg000:0009                db ? ; unexplored
```

a) исходные данные

```
Message(">%x\n", MakePackReal (SegByName ("seg000"))) );
```

b) вызываем функцию MakePackReal для создания нового элемента данных типа packed real, передавая ей адрес цепочки из десяти бестиповых байта

```
seg000:0000          db ?, ?, ?, ?, ?, ?, ?, ?, ?, ?
>1
```

c) результат – элемент данных типа packed real успешно создан

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес бестипового байта или головы элемента данных	
return	=return	пояснения
	==1	успешное завершение
	==0	ошибка

**Родственные функции:** MakeByte, MakeWord, MakeDword, MakeFloat, MakeQword, MakeDouble, MakeTbyte.

**Интерактивный аналог:** (“~Options\Setup data types”; <Alt-D>), <P>

**Замечание:** для включения типа «четвертного слова» в список типов данных, пролистываемых нажатием клавиши <D>, необходимо, вызвав диалог “Setup data types” установить галочку напротив пункта “Packed real”.

#### success MakeTbyte(long ea)

Функция создает по переданному ей линейному адресу **ea** элемент данных типа **tbyte**, длиной десять байт. Порядок следования младших и старших байт зависит от выбранного процессора. На микропроцессорах серии Intel 80x86 младший байт располагается по меньшему адресу и, соответственно, наоборот.

Если по данному адресу находится голова ранее созданного элемента данных, функция преобразует его в tbyte, а хвост элемента (если он есть) – в бестиповые байты. Если размер элемента недостаточен для преобразования, но следом за его хвостом находятся бестиповые байты, они будут автоматически присоединены к новому элементу. В противном случае, если следом за его хвостом находится другой элемент или не находится ничего, функция возвратит ошибку, не выполнив преобразования. Для выполнения преобразования необходимо предварительно уничтожить мешающие элементы вызовом MakeUnkn (см. описание функции MakeUnkn).

Ошибка возвратится и в том случае если по переданному функции линейному адресу находится хвост элемента данных, голова или хвост элемента кода.

Если один или более байт имеют неинициализированное значение, они никак не влияют на содержимое остальных и вся цепочка байт tbyte *не принимает* неинициализированного значения.

Пример использования:

```
seg000:0000          db ? ; unexplored
seg000:0001          db ? ; unexplored
seg000:0002          db ? ; unexplored
seg000:0003          db ? ; unexplored
seg000:0004          db ? ; unexplored
seg000:0005          db ? ; unexplored
seg000:0006          db ? ; unexplored
seg000:0007          db ? ; unexplored
```

```
seg000:0008          db ? ; unexplored
seg000:0009          db ? ; unexplored
```

а) исходные данные

```
Message(">%x\n", MakeQword(SegByName("seg000")));
```

б) вызываем функцию MakeTbyte для создания нового элемента данных типа tbyte, передавая ей адрес цепочки из десяти бестиповых байта

```
seg000:0000          db ?, ?, ?, ?, ?, ?, ?, ?, ?, ?
>1
```

с) результат – элемент данных типа tbyte успешно создан

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес бестипового байта или головы элемента данных	
return	=return	пояснения
	==1	успешное завершение
	==0	ошибка

**Родственные функции:** MakeByte, MakeWord, MakeDword, MakeQword, MakeFloat, MakeDouble, MakePackReal.

**Интерактивный аналог:** (“~Options\Setup data types”; <Alt-D>), <T>

**Замечание:** для включения типа «четвертного слова» в список типов данных, пролистываемых нажатием клавиши <D>, необходимо, вызвав диалог “Setup data types” установить галочку напротив пункта “Tbyte”.

success MakeStr(long ea,long endea)

Функция преобразует последовательность бестиповых байт в ASCII-строку, автоматически назначая ей метку, стиль которой задается вызовом “SetLongPrm(INF\_STRTYPE)” (см. описание функции SetLongPrm).

Аргумент **ea** задает линейный адрес начала цепочки бестиповых байт или головы элемента данных, преобразуемого в строку. Если по данному адресу находится хвост элемента данных, голова или хвост элемента кода, функция возвратит ошибку, не выполнив преобразования.

Аргумент **endea** задает линейный адрес конца строки. Если передать функции значение BADADDR, то IDA предпримет попытку вычислить адрес конца автоматически. Поддерживаются следующие типы строк – **ASCIIZ**-строки, заканчивающиеся символом нуля; **PASCAL**-строки, начинающиеся с байта, содержащего длину строки и **DELPHI**-строки, начинающиеся со слова (двойного слова), содержащего длину строки. Если строка не принадлежит ни к одному из этих трех типов, концом строки считается:

а) первый нечитаемый ASCII-символ. Перечень читаемых символов содержится в поле “AsciiStringChars” конфигурационного файла <ida.cfg>. Любой символ, не входящий в этот список, трактуется ограничителем длины строки. По умолчанию содержимое поля “AsciiStringChars” для кодировки cp866 следующее:

```
"\r\n\v\b\t\x1B"
"!\"#$%&'()*+,-./0123456789:;<=>?"
"@ABCDEFGHIJKLMNPQRSTUVWXYZ[\\]^_"
"`abcdefghijklmnopqrstuvwxyz{|}~"
```

```
"АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"
"абвгдежзийклмноп"
"рстуфхцчшщъыьэюя";
```

- b) неинициализированный байт
- c) голова элемента кода или данных
- d) конец сегмента

Если на пути от адреса начала строки до адреса ее конца встретится хотя бы один неинициализированный байт, элемент кода или данных, функция возвратит ошибку без преобразования строки.

**Замечание:** вплоть до версии 3.85 эта функция была реализована с ошибкой и передача значения BADADDR не приводила к автоматическому определению конца строки.

Пример использования:

```
seg000:0000 db 48h ; H
seg000:0001 db 65h ; e
seg000:0002 db 6Ch ; l
seg000:0003 db 6Ch ; l
seg000:0004 db 6Fh ; o
seg000:0005 db 2Ch ; ,
seg000:0006 db 20h ;
seg000:0007 db 53h ; S
seg000:0008 db 61h ; a
seg000:0009 db 69h ; i
seg000:000A db 6Ch ; l
seg000:000B db 6Fh ; o
seg000:000C db 72h ; r
seg000:000D db 0 ;
```

a) исходные данные – ASCIIZ-строка.

```
MakeStr (SegByName ("seg000"), BADADDR);
```

b) вызываем функцию MakeStr для преобразования цепочки бестиповых байтов в ASCII-строку с автоматическим определением ее конца

```
seg000:0000 aHelloSailor db 'Hello, Sailor',0
```

c) результат – успешное создание строки с автоматическим назначением метки, состоящей из допустимых в имени символов

```
??? #Верстальщику change table
```

аргумент	пояснения	
ea	линейный адрес начала цепочки бестиповых байт или головы элемента данных	
endea	!=BADADDR	линейный адрес конца строки
	==BADADDR	указание вычислять адрес конца строки автоматически
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** “~Edit\ASCII”; <A>

**Замечание:** *при нажатии клавиши <A>, IDA пытается создать по текущему адресу, ASCII-строку автоматически определяя ее длину. Для явного задания требуемой длины, необходимо предварительно выделить соответствующую область курсорными клавишами, удерживая нажатым <shift> или мышью, удерживая нажатой правую кнопку.*

#### success MakeArray(long ea,long nitems)

Функция создает массив состоящий из данных одного типа – байтов, слов, двойных слов, четверных слов, двойных слов в формате float, четверных слов в формате double, packed real, tbyte. Бестиповые байты могут стать частью массива любого типа. Строки не могут быть элементами никакого массива.

Тип массива определяется типом его первого элемента. Все остальные элементы массива на момент его создания должны быть представлены бестиповыми байтами, - последовательность типизированных данных не может быть преобразована в массив.

Элементы массива записываются в строку, отделяясь друг от друга знаком запятой. Если два или более подряд идущих элемента имеют одно и то же значение (в том числе и неинициализированное) для сокращения ассемблерного листинга используется конструкция “DUP”.

Аргумент **ea** задает линейный адрес первого элемента массива или линейный адрес начала цепочки бестиповых байт.

Аргумент **nitems** задает размер массива, выраженный в количестве элементов. Массив создается даже в том случае, когда **nitems** равен единице.

Пример использования:

```
seg000:0000      db  48h ; H
seg000:0001      db  65h ; e
seg000:0002      db  6Ch ; l
seg000:0003      db  6Ch ; l
seg000:0004      db  6Fh ; o
seg000:0005      db  2Ch ; ,
seg000:0006      db  20h ;
seg000:0007      db  53h ; S
seg000:0008      db  61h ; a
seg000:0009      db  69h ; i
seg000:000A      db  6Ch ; l
seg000:000B      db  6Fh ; o
seg000:000C      db  72h ; r
```

a) исходные данные

```
MakeArray(SegByName("seg000"),14);
```

b) вызываем функцию MakeArray для преобразования последовательности бестиповых байт в массив байт

```
seg000:0000 db 48h, 65h, 2 dup(6Ch), 6Fh, 2Ch, 20h, 53h, 61h, 69h
seg000:0000 db 6Ch, 6Fh, 72h, 0
```

c) результат – успешно созданный массив.

**Внимание:** *если все элементы массива не уместятся на одной строке, они*

автоматически переносятся на следующую, но слева от них указывается не адрес данного элемента массива, а адрес головы массива!

**Замечание:** для изменения размеров массива (усечения или расширения) достаточно передать функции адрес его начала и новую длину.

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес первого элемента массива или линейный адрес головы уже существующего массива	
nitems	размер массива, выраженный в количестве элементов	
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** “~Edit\Array”; <\*>

success MakeAlign(long ea,long count,long align)

Функция помещает в ассемблерный файл директиву выравнивания **align** и исключает из дизассемблируемого листинга байты, используемые для выравнивания.

**Замечание:** микропроцессоры серии Intel 80x86 используют выравнивание используется для ускорения доступа к данным и инструкциям (подробнее об этом можно технической документации фирмы Intel), но существуют процессоры, которые требуют обязательного выравнивания и при обращении к не выровненным данным (инструкциям) генерируют исключение.

Аргумент **ea** задает линейный адрес первого байта, используемого для выравнивания. Если по этому адресу расположен хвост элемента данных, голова или хвост элемента кода, функция возвратит ошибку.

Аргумент **count** задает количество байт, используемых для выравнивания. Значение **count** должно быть больше нуля и меньше кратности выравнивания, т.е.  $2^{\text{align}} > \text{count} > 0$ , в противном случае функция возвратит ошибку.

Аргумент **align** задает кратность выравнивания и представляет собой показатель степени с основанием два. Т.е. если align равен четырем, то кратность выравнивания – шестнадцати, т.к.  $2^4=16$ . Если align равен нулю, функция определяет необходимую степень выравнивания автоматичен, используя наибольшее возможное значение.

Для изменения величины выравнивания достаточно передать функции MakeAlign линейный адрес уже существующей директивы Align и новые значения count и align.

Пример использования:

```
seg000:0000          db  48h ; H
seg000:0001          db  65h ; e
seg000:0002          db  6Ch ; l
seg000:0003          db  6Ch ; l
seg000:0004          db  6Fh ; o
```

а) исходные данные

```
MakeAlign (SegByName ("seg000")+1, 3, 2) ;
```

b) вызываем функцию MakeAlign для помещения по адресу seg000:0001 директивы align 4. Для выравнивания используются три байта – seg0001, seg0002 и seg0003.

```
seg000:0000          db  48h ; H
seg000:0001          align 4
seg000:0004          db  6Fh ; o
c) результат – успешное создание директивы выравнивания
```

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес первого байта, используемого для выравнивания или уже существующей директивы align	
count	число байт, используемых для выравнивания	
align	=align	пояснения
	==[1..5]	показатель степени выравнивания с основанием два
	==0	автоматическое определение кратности выравнивания
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** “~Edit\Structs\Other\ Create alignment directive”;<L>

#### long MakeCode (long ea)

Функция создает по указанному адресу элемент кода, выполняя дизассемблирование первой машинной инструкции. Если это возможно, автоматически дизассемблируются и другие инструкции. Это происходит в следующих случаях:

- a) текущая инструкция не изменяет нормального выполнения программы и за ее концом расположены бестиповые байты;
- b) текущая инструкция изменяет нормальное выполнение программы, осуществляя переход по непосредственному адресу, тогда IDA продолжит дизассемблирование с этого адреса

Если встречается инструкция, изменяющая адрес перехода непредсказуемым образом (например, RET) IDA прекращает дизассемблирование.

Во время дизассемблирования IDA при необходимости создает перекрестные ссылки и автогенерируемые метки. Подробнее об этом можно прочитать в главах «Перекрестные ссылки» и «Глобальные настройки» соответственно.

**Замечание:** IDA эмулирует выполнение кода на виртуальном процессоре, с целью отслеживания изменения регистра указателя команд, и дизассемблирует все инструкции, на которые он указывает или может указывать при определенных обстоятельствах.

Благодаря этому дизассемблируется все вызываемые функции, условные переходы и все косвенные ссылки, которые IDA в состоянии распознать (например, если это не запрещено настройками, она может автоматически преобразовывать 32-разрядные непосредственные операнды по модулю больше 0x10000 в смещения на код – см. главу «Глобальные настройки»).

При успешном завершении функция возвращает длину первой дизассемблированной инструкции, выраженную в байтах, в противном случае возвращается ноль, сигнализируя об ошибке.



Переча линейного адреса головы уже существующего элемента кода, привет к повторному анализу инструкции; будут заново созданы перекрестные ссылки, авто генерируемые метки и т.д., а функция возвратит длину инструкции, расположенной по линейному адресу ea.

Пример использования:

```

seg000:0100 start      db  83h ; Г
seg000:0101           db  0C6h ; Ѓ
seg000:0102           db   6 ;
seg000:0103           db  0FFh ;
seg000:0104           db  0E6h ; Ц
seg000:0105           db  0B9h ; Ъ
seg000:0106           db  0BEh ; Ы
seg000:0107           db  14h ;
seg000:0108           db   1 ;
seg000:0109           db  0ADh ; Н
seg000:010A           db  91h ; С
seg000:010B           db  56h ; V
seg000:010C           db  80h ; А
seg000:010D           db  34h ; 4
seg000:010E           db  66h ; f
seg000:010F           db  46h ; F
seg000:0110           db  0E2h ; Т
seg000:0111           db  0FAh ; ·
seg000:0112           db  0FFh ;
seg000:0113           db  0E6h ; Ц
seg000:0114           db  18h ;
seg000:0115           db   0 ;

```

**а) исходные данные**

```
Message (">%X\n", MakeCode (SegByName ("seg000")+0x100) );
```

**б) вызываем функцию MakeCode для дизассемблирования кода**

```

seg000:0100           add    si, 6
seg000:0103           jmp    si
seg000:0103 ; -----
seg000:0105           db  0B9h ; Ъ
seg000:0106           db  0BEh ; Ы
seg000:0107           db  14h ;
seg000:0108           db   1 ;
seg000:0109           db  0ADh ; Н
seg000:010A           db  91h ; С
seg000:010B           db  56h ; V
seg000:010C           db  80h ; А
seg000:010D           db  34h ; 4
seg000:010E           db  66h ; f
seg000:010F           db  46h ; F
seg000:0110           db  0E2h ; Т
seg000:0111           db  0FAh ; ·
seg000:0112           db  0FFh ;
seg000:0113           db  0E6h ; Ц
seg000:0114           db  18h ;
seg000:0115           db   0 ;

```

>3

**с) результат** – функция дизассемблировала две инструкции и остановилась,

встретив регистровый переход, целевой адрес которого она предсказать не в силах; по завершению дизассемблирования функция возвратила длину первой инструкции, равную трем байтам

```
Message (">%X\n", MakeCode (SegByName ("seg000")+0x106) );
```

d) повторно вызываем функцию MakeCode, передавая ей адрес следующей инструкции (значение регистра SI при загрузке com файла равно 0x100, а после выполнения инструкции ADD SI, 6 – 0x106, следовательно целевой адрес перехода JMP SI равен 0x106)

```
seg000:0100          add     si, 6
seg000:0103          jmp     si
seg000:0103 ; -----
seg000:0105          db 0B9h ; ¶
seg000:0106 ; -----
seg000:0106          mov     si, 114h
seg000:0109          lodsw
seg000:010A          xchg   ax, cx
seg000:010B          push   si
seg000:010C          loc_0_10C:                ; CODE XREF: seg000:0110j
seg000:010C          xor     byte ptr [si], 66h
seg000:010F          inc     si
seg000:0110          loop   loc_0_10C
seg000:0112          jmp     si
seg000:0112 ; -----
seg000:0114          db 18h ;
seg000:0115          db 0 ;
>3
```

e) результат – функция продолжила дизассемблирование, автоматически создавая перекрестные ссылки и автогенерируемые метки, до тех пор пока не встретила инструкцию регистрового перехода, целевой адрес которого предсказать не в силах.

??? #Верстальщику – chabge table

аргумент	пояснения	
ea	линейный адрес бестипового байта или головы уже существующего элемента кода	
return	=return	пояснения
	!=0	длина первой дизассемблируемой инструкции
	==0	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** “~Edit\Code” <C>

### char GetMnem(long ea)

Функция возвращает символьную мнемонику инструкции элемента кода, расположенного по линейному адресу **ea**. Для получения операндов (если они есть) следует воспользоваться функцией GetOpnd (см. главу «Операнды»)

Пример использования:

```
seg000:0000          mov     ah, 9
```

a) исходные данные – требуется получить символьную мнемонику инструкции

```
Message(">%s\n", GetMnem (SegByName ("seg000"))) );  
b) вызов функции GetMnem
```

```
>mov
```

c) результат – мнемоника инструкции в символьном представлении

??? #Верстальщику – chabge table

аргумент	пояснения	
ea	линейный адрес элемента кода	
return	=return	пояснения
	!=""	мнемоника в символьном представлении
	=="	ошибка

**Родственные функции:** GetOpnd

**Интерактивный аналог:** *нет*

void MakeUnkn(long ea,long expand)

Функция разрушает элемент, заданный любым принадлежащим ему адресом, превращая его содержимое в бестиповые байты. Объекты, связанные с элементом (например, метки, комментарии) при этом не уничтожаются.

**Замечание:** *автогенерируемые метки, назначаемые ASCII-строкам при их разрушении удаляются*

Аргумент **ea** задает любой линейный адрес, принадлежащий разрушаемому элементу.

Аргумент **expand** будучи неравным нулю указывает на необходимость разрушения всей цепочки элементов, связанных друг с другом перекрестными ссылками типа «ссылка на следующую инструкцию» (см. главу «Перекрестные ссылки»)

Пример использования:

### 1. Эксперимент

```
seg000:0000 aHelloSailor db 'Hello, Sailor',0
```

a) исходные данные

```
MakeUnkn (SegByName ("seg000")+0x1, 0) ;
```

b) вызов функции MakeUnkn для разрушения элемента данных типа «ASCII-строка»

```
seg000:0000 db 48h ; H  
seg000:0001 db 65h ; e  
seg000:0002 db 6Ch ; l  
seg000:0003 db 6Ch ; l  
seg000:0004 db 6Fh ; o  
seg000:0005 db 2Ch ; ,  
seg000:0006 db 20h ;  
seg000:0007 db 53h ; S  
seg000:0008 db 61h ; a
```

```

seg000:0009          db  69h ; i
seg000:000A          db  6Ch ; l
seg000:000B          db  6Fh ; o
seg000:000C          db  72h ; r

```

c) результат – элемент данных разрушен

## 2. Эксперимент

```

seg000:0100          add   si, 6
seg000:0103          jmp   si

```

a) исходные данные

```
MakeUnkn (SegByName ("seg000"), 0);
```

b) вызов функции MakeUnkn для разрушения только одного элемента кода

```

seg000:0100 start    db  83h ; Г
seg000:0101          db  0C6h ; Ё
seg000:0102          db   6 ;
seg000:0103 ; _____

```

c) разрушен один элемент кода

## 3. Эксперимент

```

seg000:0100          add   si, 6
seg000:0103          jmp   si

```

a) исходные данные

```
MakeUnkn (SegByName ("seg000"), 1);
```

b) вызов функции MakeUnkn для разрушения всей цепочки элементов кода

```

seg000:0100 start    db  83h ; Г
seg000:0101          db  0C6h ; Ё
seg000:0102          db   6 ;
seg000:0103          db  0FFh ;
seg000:0104          db  0E6h ; ц
seg000:0105          db  0B9h ; ђ

```

c) результат – вся цепочка элементов кода разрушена

??? #верстальщику – change table

агумент	пояснения	
ea	любой линейный адрес, принадлежащий разрушаемому элементу	
expand	==0	разрушение только одного элемента кода или данных
	!=0	разрушение всей цепочки элементов кода или только одного элемента данных.
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** “~Edit\Undefine”; <U>

| **Замечание:** нажатие <U> равносильно вызову `MakeUnk(ScreenEA(),1)` и

разрушает всю цепочку элементов кода. При необходимости разрушения одного элемента, его следует предварительно выделить курсорными клавишами, удерживая нажатым <Shift> или мышью, удерживая нажатой левую кнопку.

### long FindCode(long ea,long flag)

Функция ищет ближайший к переданному ей линейному адресу **ea** элемент кода, возвращая в случае успешного завершения поиска адрес его головы. В зависимости от флага направления поиск может идти как вперед (от младших адресов к старшим), так и назад (от старших адресов к младшим). Переданный функции линейный адрес в этот диапазон поиска **не входит** и не обязательно должен принадлежать какому-нибудь сегменту.

Аргумент **flag** задает направление поиска – если его младший бит установлен поиск идет от младших адресов к старшим и, соответственно, наоборот.

Пример использования:

```
seg000:0100          mov     ax, 9
seg000:0103          mov     dx, 133h
```

а) исходные данные – требуется получить линейный первого элемента кода

```
Message(">%s\n", atoa( FindCode(0,1) ) );
```

б) вызов функции FindCode – адрес начала поиска равен нулю, единичное значение флага направление указывает вести поиск с увеличением адресов

```
>seg000:0100
```

результат – линейный первого элемента кода

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес начала поиска, не обязательно принадлежащий какому-нибудь сегменту	
flag	=flag	пояснения
	==1	прямое направление поиска
	==0	обратное направление поиска
return	=return	пояснения
	!=BADADDR	линейный адрес элемента кода
	==BADADDR	ошибка

**Родственные функции:** FindData, FindExplored, FindUnexplored

**Интерактивный аналог:** "~Nabigate\Search for\Next Code"; <Ctrl-C>

### long FindData(long ea,long flag)

Функция ищет ближайший к переданному ей линейному адресу **ea** элемент кода, возвращая в случае успешного завершения поиска адрес его головы. В зависимости от флага направления поиск может идти как вперед (от младших адресов к старшим), так и назад (от старших адресов к младшим). Переданный функции линейный адрес в этот диапазон поиска **не входит** и не обязательно должен принадлежать какому-нибудь сегменту.

Аргумент **flag** задает направление поиска – если его младший бит установлен поиск идет от младших адресов к старшим и, соответственно, наоборот.

Пример использования:

```

seg000:0000          mov     ah, 9
seg000:0002          mov     dx, 108h
seg000:0005          int     21h
seg000:0005
seg000:0007          retn
seg000:0007 ; _____
seg000:0008 aHelloIda      db 'Hello, IDA'

```

а) исходные данные – требуется получить линейный последнего элемента данных

```
Message(">%s\n", atoa (FindData (BADADDR, 0) ) );
```

б) вызов функции FindData

```
>seg000:0108
```

результат – линейный адрес последнего элемента данных

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес начала поиска, не обязательно принадлежащий какому-нибудь сегменту	
flag	=flag	пояснения
	==1	прямое направление поиска
	==0	обратное направление поиска
return	=return	пояснения
	!=BADADDR	линейный адрес элемента данных
	==BADADDR	ошибка

**Родственные функции:** FindCode, FindExplored, FindUnexplored

**Интерактивный аналог:** "~Nabigate\Search for\Next Data"; <Ctrl-D>

long FindExplored(long ea,long flag)

Функция ищет ближайший к переданному ей линейному адресу **ea** элемент кода или данных, возвращая в случае успешного завершения поиска адрес его головы. В зависимости от флага направления поиск может идти как вперед (от младших адресов к старшим), так и назад (от старших адресов к младшим). Переданный функции линейный адрес в этот диапазон поиска **не входит** и не обязательно должен принадлежать какому-нибудь сегменту.

Аргумент **flag** задает направление поиска – если его младший бит установлен поиск идет от младших адресов к старшим и, соответственно, наоборот.

Пример использования:

```

seg000:0100          DB 99h ; Щ
seg000:0101          DW 666h

```

а) исходные данные – требуется получить линейный первого элемента кода или данных

```
Message(">%s\n", atoa (FindExplored (0, 1) ) );
```

б) вызов функции FindExplored – адрес начала поиска равен нулю, единичное значение флага направление указывает вести поиск с увеличением адресов

```
>seg000:0101
```

результат – линейный первого элемента

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес начала поиска, не обязательно принадлежащий какому-нибудь сегменту	
flag	=flag	пояснения
	==1	прямое направление поиска
	==0	обратное направление поиска
return	=return	пояснения
	!=BADADDR	линейный адрес элемента любого вида
	==BADADDR	ошибка

**Родственные функции:** FindCode, FindData, FindUnexplored

**Интерактивный аналог:** "~Nabigate\Search for\Next explored"; <Ctrl-A>

long FindUnexplored(long ea, long flag)

Функция ищет ближайший к переданному ей линейному адресу **ea** бестиповой байт, возвращая в случае успешного завершения поиска его адрес. В зависимости от флага направления поиск может идти как вперед (от младших адресов к старшим), так и назад (от старших адресов к младшим). Переданный функции линейный адрес в этот диапазон поиска **не входит** и не обязательно должен принадлежать какому-нибудь сегменту.

Аргумент **flag** задает направление поиска – если его младший бит установлен поиск идет от младших адресов к старшим и, соответственно, наоборот.

Пример использования:

```
seg000:0100                DW 666h
seg000:0102                DB 99h ; Щ
```

а) исходные данные – требуется получить линейный первого бестипового байта

```
Message(">%s\n", atoa(FindUnexplored(0,1)));
```

б) вызов функции FindUnexplored – адрес начала поиска равен нулю, единичное значение флага направление указывает вести поиск с увеличением адресов

```
>seg000:0102
```

результат – линейный первого бестипового байта

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес начала поиска, не обязательно принадлежащий какому-нибудь сегменту	
flag	=flag	пояснения
	==1	прямое направление поиска
	==0	обратное направление поиска
return	=return	пояснения
	!=BADADDR	линейный адрес бестипового байта
	==BADADDR	ошибка

**Родственные функции:** FindCode, FindData, FindExplored

**Интерактивный аналог:** "~Nabigate\Search for\Next Unexplored"; <Ctrl-U>

## ОПЕРАНДЫ

### #definition

Элементы могут содержать один и более операндов, включающих в себя непосредственные значения. Одно и то же непосредственное значение может являться и константой, и смещением в сегменте, и базовым адресом сегмента. Константа в свою очередь может отображаться в двоичной, восьмеричной, десятичной и шестнадцатеричной системе исчисления или символьной ASCII-строки.

Тип и представление каждого операнда определяется значением соответствующих битов флагов (см. таблицу 14). Две группы битов определяют представление первого и второго операнда; если же элемент содержит более двух операндов, третий и все последующие операнды имеют тот же самый тип и представление, что и второй операнд.

Если ни один бит атрибутов операнда не установлен, операнд не имеет никакого типа, (т.е. имеет тип "void") и отображается в системе исчисления принятой по умолчанию. В зависимости от настроек IDA Pro бестиповые операнды могут отображаться другим цветом (по умолчанию красным).

Определить имеет ли некий элемент кода непосредственный операнд или нет можно анализом бита FF\_IMMD флагов. Если он установлен – элемент кода имеет непосредственный операнд, и, соответственно, наоборот. Значение бита FF\_IMMD не зависит от типа непосредственного операнда – чем бы он ни был: смещением, константой, базовым адресом сегмента или имел тип void, – флаг FF\_IMMD указывает на сам факт наличия (отсутствия) непосредственного операнда, но никак не связан с его типом.

В полностью дизассемблированной программе не должно быть ни одного операнда с типом void, – типы всех операндов должны заданы явно в соответствии с их назначением, которое можно узнать путем анализа программы.

В некоторых случаях IDA Pro позволяет автоматически отличить смещения от констант:

а) используя информацию о перемещаемых элементах (*fixup info*) IDA Pro может автоматически преобразовать бестиповые операнды в базовые адреса сегментов и смещения

б) в 32-разрядном сегменте, инструкция, имеющая непосредственный операнд, содержащий значение 0x10000 и выше, автоматически преобразуется в смещение, если это разрешено настройками

с) то же, что и в пункте б но для данных

д) непосредственный операнд инструкции push, следующей за инструкцией, заносащий в стек базовый адрес сегмента, автоматически преобразуется в смещение, если это разрешено настройками

е) непосредственный операнд, копируемый инструкцией MOV в один из регистров общего назначения, автоматически преобразуется в смещение, если он предшествует инструкции MOV, заносащий в один из сегментных регистров базовый адрес сегмента

ф) непосредственный операнд, копируемый инструкцией MOV в ячейку памяти независимо от способа ее адресации, автоматически преобразуется в смещение, если он предшествует инструкции MOV, заносащий в ячейку памяти базовый адрес сегмента.

Авто-анализатор IDA Pro непрерывно совершенствуется, и новые версии становятся способными автоматически распознать смещения все в большем и большем числе случаев, - однако, вместе с этим увеличивается и процент ложных срабатываний, поэтому, окончательная доводка дизассемблируемого листинга ложится на плечи пользователя.



представление первого слева операнда		
флаг	#	представление
FF_0VOID	0x00000000	тип void
FF_0NUMH	0x00100000	шестнадцатеричное представление
FF_0NUMD	0x00200000	десятичное представление
FF_0CHAR	0x00300000	символьное представление
FF_0SEG	0x00400000	представление в виде базового адреса сегмента
FF_0OFF	0x00500000	представление в виде смещения в сегменте
FF_0NUMB	0x00600000	бинарное представление
FF_0NUMO	0x00700000	восьмеричное представление
FF_0ENUM	0x00800000	представление в виде перечисления
FF_0FOP	0x00900000	пользовательское представление
FF_0STRO	0x00A00000	представление в виде смещения в структуре
FF_0STK	0x00B00000	представление в виде стековой переменной
представление второго слева операнда		
флаг	#	представление
FF_1VOID	0x00000000	тип void
FF_1NUMH	0x00100000	шестнадцатеричное представление
FF_1NUMD	0x00200000	десятичное представление
FF_1CHAR	0x00300000	символьное представление
FF_1SEG	0x00400000	представление в виде базового адреса сегмента
FF_1OFF	0x00500000	представление в виде смещения в сегменте
FF_1NUMB	0x00600000	бинарное представление
FF_1NUMO	0x00700000	восьмеричное представление
FF_1ENUM	0x00800000	представление в виде перечисления
FF_1FOP	0x00900000	пользовательское представление
FF_1STRO	0x00A00000	представление в виде смещения в структуре
FF_1STK	0x00B00000	представление в виде стековой переменной

**Таблица 13 возможные представления непосредственных операндов элементов типа данные и код**

### Сводная таблица функций

функции, изменяющие отображение операндов	
название функции	краткое описание
success OpBinary(long ea,int n)	отображает операнд (операнды) в двоичном виде
success OpOctal(long ea,int n)	отображает операнд (операнды) в восьмеричном виде
success OpDecimal(long ea,int n)	отображает операнд (операнды) в десятичном виде
success OpHex(long ea,int n)	отображает операнд (операнды) в шестнадцатеричном виде
success OpChr (long ea,int n)	отображает операнд (операнды) в символьном виде
success OpNumber(long ea,int n)	отображает операнд (операнды) в систем исчисления принятой по умолчанию
success OpOff (long ea,int n,long base)	отображает операнд (операнды) в виде смещения, отсчитываемого относительно начала сегмента

success OpOffEx(long ea,int n,long reftype,long target,long base,long tdelta)	отображает операнд (операнды) в виде смещения, отсчитываемого относительно любого адреса, принадлежащего сегменту
success OpSeg(long ea,int n)	отображает операнд (операнды) в виде имени сегмента, базовый адрес которого равен значению операнда
success OpAlt(long ea,long n,char str)	отображает операнд (операнды) в виде символьной строки, заданной пользователем
success OpSign(long ea,int n)	отображает операнд (операнды) в знаковой или целочисленной форме (функция работает как триггер)
success OpStkvar(long ea,int n)	отображает непосредственное значение, использующее для базовой адресации в виде имени локальной переменной
<b>функции, возвращающие операнды</b>	
название функции	краткое описание
char GetOpnd(long ea,long n)	возвращает операнд в символьном представлении
long GetOpType(long ea, long n)	возвращает тип операнда
long GetOperandValue (long ea,long n)	возвращает значение операнда
char AltOp (long ea,long n)	возвращает операнд, определенный пользователем
<b>функции, обеспечивающие поиск операндов</b>	
название функции	краткое описание
long FindVoid(long ea, long flag)	возвращает линейный адрес очередного операнда неопределенного типа
long FindImmediate (long ea, long flag, long value);	возвращает линейный адрес очередного элемента с операндами, имеющими указанное значение
char Demangle (char name, long disable_mask)	возвращает незамангленое имя метки

### success OpBinary(long ea,int n)

Функция отображает операнд (операнды) в двоичном виде, добавляя в его конце суффикс 'b'.

Пример использования:

```
seg000:0000          mov    ax,41h
```

а) исходные данные

```
OpBinary(SegByName("seg000"),1);
```

б) вызов функцию OpBinary для преобразования второго слева операнда в двоичный вид.

```
seg000:0000          mov    ax,1000001b
```

с) результат – второй слева операнд преобразован в двоичный вид

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	операнд
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** OpOcatl, OpDecimal, OpHex, OpChr, OpNumer

**Интерактивный анлог:** “~Edit\Operand types\Binary”; <B>

success OpOctal(long ea,int n)

Функция отображает операнд (операнды) в восьмеричном виде, добавляя в его конце суффикс ‘o’.

Пример использования:

```
seg000:0000 mov ax,41h
```

а) исходные данные

```
OpOctal (SegByName ("seg000"), 1);
```

б) вызов функции OpOctal для преобразования второго слева операнда в восьмеричный вид.

```
seg000:0000 mov ax,101o
```

с) результат – второй слева операнд преобразован в восьмеричный вид

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	операнд
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** OpBinary, OpDecimal, OpHex, OpChr, OpNumer

**Интерактивный анлог:** «~Edit\Operand types\Octal»

success OpDecimal(long ea,int n)

Функция отображает операнд (операнды) в десятичном виде. Пример использования:

```
seg000:0000 mov ax,41h
```

a) исходные данные

```
OpDecimal (SegByName ("seg000"), 1);
```

b) вызов функцию OpDecimal для преобразования второго слева операнда в десятичный вид.

```
seg000:0000 mov ax, 65
```

c) результат – второй слева операнд преобразован в десятичный вид

??? #верстальщику – change table

аргумент	пооянения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	операнд
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пооянения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** OpBinary, OpOctal, OpHex, OpChr, OpNumer

**Интерактивный анлог:** «Edit\Operand types\Decimal»; <H>

success OpHex(long ea,int n)

Функция отображает операнд (операнды) в шестнадцатеричном виде, добавляя в его конце суффикс 'h'.

Пример использования:

```
seg000:0000 mov ax, 65
```

a) исходные данные

```
OpHex (SegByName ("seg000"), 1);
```

b) вызов функцию OpHex для преобразования второго слева операнда в шестнадцатеричный вид.

```
seg000:0000 mov ax, 41h
```

c) результат – второй слева операнд преобразован в шестнадцатеричный вид

??? #верстальщику – change table

аргумент	пооянения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	операнд
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пооянения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** OpBinary, OpOctal, OpDecimal, OpChr, OpNumer

**Интерактивный анлог:** «~Edit\Operand types\Hexadecimal»; <Q>

success OpChr(long ea,int n)

Функция отображает операнд (операнды) в символьном виде, заключая его в кавычки. Если операнд содержит один или больше нечитаемых байт, функция возвратит ошибку. Перечень читаемых символов содержится в поле "AsciiStringChars" конфигурационного файла <ida.cfg>. По умолчанию содержимое поля "AsciiStringChars" для кодировки *cp866* следующее:

```
"\r\n\a\v\b\t\x1B"  
"!\"#$%&'()*+,-./0123456789:;<=>?"  
"@ABCDEFGHIJKLMNORSTUVWXYZ[\]^_`"  
"abcdefghijklmnopqrstuvwxyz{|}~"  
"АВВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"  
"абвгдежзийклмнопqrstuvwxyz{|}~"  
"рстуфхцчшщъыьэюя";
```

**Замечание:** порядок следования старших и младший байт зависит от выбранного типа процессора. У микропроцессоров серии Intel 80x86 младший байт располагается по меньшему адресу, а старший, соответственно, наоборот.

Пример использования:

### 1. Эксперимент

```
seg000:0000 mov ax, 65
```

а) исходные данные

```
OpChr (SegByName ("seg000"), 1);
```

б) вызов функцию OpChar для преобразования второго слева операнда в символьный вид.

```
seg000:0000 mov ax, 'A'
```

с) результат – второй слева операнд преобразован в шестнадцатеричный вид

### 2. Эксперимент

```
seg000:0000 dq 4944412050726F21h
```

а) исходные данные

```
OpChr (SegByName ("seg000"), 0);
```

б) вызов функции OpChr для преобразования первого слева операнда в символьный вид

```
seg000:0000 dq 'IDA Pro!'
```

с) результат – успешное преобразование

??? #верстальщику – change table

аргумент	пояснения
ea	линейный адрес элемента, которому принадлежит операнд
n	=n операнд

	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** OpBinary, OpOctal, OpDecimal, OpHex, OpNumer

**Интерактивный анлог:** «Edit\Operand types\Chaster»; <R>

success OpNumber(long ea,int n)

Функция отображает операнд (операнды) в форме исчисления принятой по умолчанию. По умолчанию системой исчисления по умолчанию назначена шестнадцатеричная система исчисления.

Пример использования:

```
seg000:0000                                mov    ax,65
```

a) исходные данные

```
OpNumber (SegByName ("seg000"), 1);
```

b) вызов функции OpNumber для преобразования второго слева операнда в систему исчисления по умолчанию.

```
seg000:0000                                mov    ax,41h
```

c) результат – второй слева операнд преобразован в шестнадцатеричный вид

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	операнд
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** OpBinary, OpOctal, OpDecimal, OpHex, OpChr.

**Интерактивный анлог:** «Edit\Operand types\ Number»; <#>

success OpOff(long ea,int n,long base)

Функция отображает операнд (операнды) в виде смещения относительно заданного сегмента, автоматически создавая автогенерируемую метку по целевому адресу (если целевой адрес не имеет метки) и перекрестную ссылку соответствующего типа (см. главу «Перекрестные ссылки»). Разрядность операнда, представляемого в виде смещения, должна быть равна разрядности соответствующего сегмента, иначе функция возвратит ошибку.

Аргумент **ea** задает линейный адрес элемента, которому принадлежит

операнд.

Аргумент **base** задает базовый адрес сегмента, выраженный в байтах (не параграфах!) относительно которого отсчитывается смещение.

Аргумент **n** задает операнд, отображаемый в виде смещения (см. таблицу).

Для выполнения обратной операции, т.е. преобразованию смещения к непосредственному значению, достаточно передать функции нулевой базовый адрес сегмента.

Пример использования:

```
seg000:0100 start      proc near
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, 108h
seg000:0105          int     21h
seg000:0107          retn
seg000:0107 start      endp
seg000:0107
seg000:0107 ; _____
seg000:0108          db 'Hello,World!',0Dh,0Ah,'$'
seg000:0108 seg000    ends
```

а) исходные данные

```
OpOff (SegByName ("seg000")+0x102, 1, SegByName ("seg000")) ;
```

б) вызов функции OpOff для отображения константы, загружаемой в регистр DX в виде смещения относительно текущего сегмента

```
seg000:0100 start      proc near
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset asc_0_108 ; "Hello,World!\r\n$"
seg000:0105          int     21h
seg000:0107          retn
seg000:0107 start      endp
seg000:0107
seg000:0107 ; _____
seg000:0108 asc_0_108 db 'Hello,World!',0Dh,0Ah,'$' ; DATA XREF: start+20
seg000:0108 seg000    ends
```

в) результат – константа, загружаемая в регистр DX отображена в виде смещения, предваренного директивой “offset”, автоматически создана метка и перекрестна ссылка (в тексте они выделены жирным шрифтом).

Ближайший аналог (~Edit\Operad types\Offset by any segment)

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, котрому принадлежит операнд	
n	=n	пояснения
	=0	первый слева операнд
	=1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
base	базовый адрес сегмента, выраженный в байтах (не параграфах!) относительно которого отсчитывается смещение	
return	=return	пояснения
	=1	успешное завершение операции

	==0	ошибка
--	-----	--------

**Родственные функции:** OpOffEx

**Интерактивный аналог:** “~Edit\Operad types\Offset by any segment”; <Alt-R>

success OpOffEx(long ea,int n,long reftype,long target,long base,long tdelta)

Функция отображает операнд (операнды) в виде смещения, отсчитываемого от любого заданного адреса, не обязательно совпадающего с базовым адресом сегмента. Такая необходимость возникает например в случае обращения к элементу структуры, смещение которого требуется отсчитывать относительно начала этой структуры.

Данная функция является усовершенствованным вариантом функции OpOff и поддерживает не только смещения, разрядность которых равна разрядности соответствующего сегмента, но смещения записанные в восьми или шестнадцати младших (старших) битах шестнадцати и тридцати двух разрядных операндов соответственно (см. таблицу ???). При этом остальные биты операнда маскируются операцией «логического и» AND.

Аргумент **ea** задает линейный адрес элемента, которому принадлежит операнд.

Аргумент **n** задает операнд, отображаемый в виде смещения (см. таблицу ???)

Аргумент **reftype** задает тип смещения и может принимать одно из значений, перечисленных в таблице ???

Аргумент **target** задает линейный адрес целевого смещения выраженный в байтах, относительного которого будет отсчитываться смещение операнда. Если в качестве целевого смещения передать значение BADADDR, целевое смещение будет вычислено автоматически по следующей формуле:  $target = operand\_value - tdelta + base$

Аргумент **base** задает базовый адрес сегмента, выраженный в байтах, относительно которого задается целевое смещение.

Аргумент **tdelta** задает относительное смещение, отсчитываемое относительно целевого смещения. Относительное смещение может быть как положительным, так и отрицательным. Если оно равно нулю, то данная функция становится эквивалентна функции OpOff (см. описание функции OpOff).

Значение операнда должно соответствовать следующему соотношению  $operand\_value = target + tdelta - base$ , в противном случае функция вернет ошибку.

определение	#	тип смещения
REF_OFF8	0	8-битное смещение
REF_OFF16	1	16-битное смещение
REF_OFF32	2	32-битное смещение
REF_LOW8	3	смещение представлено 8 младшими битами 16 битного непосредственного значения
REF_LOW16	4	смещение представлено 16 младшими битами 32 битного непосредственного значения
REF_HIGH8	5	смещение представлено 8 старшими битами 16 битного непосредственного значения
REF_HIGH16	6	смещение представлено 16 старшими битами 32 битного непосредственного значения



Таблица 14

Пример использования:

```

seg000:0100 start:
seg000:0100          mov     ax, 105h
seg000:0103          retn
seg000:0103 ; _____
seg000:0104 MyStruc  db  0
seg000:0105          dw  6666h
seg000:0107          dw  9999h
seg000:0107 seg000  ends
seg000:0107

```

а) исходные данные – требуется представить непосредственное значение, загружаемое в регистр AX в виде смещения, отсчитываемого относительно начала структуры MyStruc.

```

OpOffEx (SegByName ("seg000")+0x100,1,REF_OFF16,
         SegByName ("seg000")+0x104,SegByName ("seg000"),1);

```

б) вызов функции OpOffEx для представления непосредственного значения в виде смещения, отсчитываемого относительно начала структуры MyStruc.

**Пояснение:** линейный адрес структуры MyStruc равен SegByName("seg000")+0x104, следовательно, целевой адрес tagreg равен SegByName("seg000")+0x104;

базовый адрес сегмента, которому принадлежит структура, будучи выраженным в байтах равен SegNyName("seg000"), следовательно, аргумент base равен SegByName("seg000");

смещение искомого элемента относительно начала структуры равно operand\_value – offset MyStruc, т.е. в непосредственных значениях – 0x105 – 0x104 = 1, следовательно, аргумент tdelta равен 1;

операнд представляет собой 16-разрядное непосредственное значение, поэтому, тип смещения - REF\_OFF16.

```

seg000:0100 start:
seg000:0100          mov     ax, offset MyStruc+1
seg000:0103          retn
seg000:0103 ; _____
seg000:0104 MyStruc  db  0 ; DATA XREF: seg000:0100o
seg000:0105          dw  6666h
seg000:0107          dw  9999h
seg000:0107 seg000  ends

```

с) результат – непосредственное значение теперь представлено в виде смещения, отсчитываемого от начала структуры MyStruc

**Замечание:** в данном примере было допустимо использовать автоматическое определение целевого адреса, однако, для большей ясности оно было вычислено вручную.

??? #Верстальщику – chabge table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды

	== -1	все операнды	
reftype	==reftype	#	тип смещения
	==REF_OFF8	0	8-битное смещение
	==REF_OFF16	1	16-битное смещение
	==REF_OFF32	2	32-битное смещение
	==REF_LOW8	3	смещение представлено 8 младшими битами 16 битного непосредственного значения
	==REF_LOW16	4	смещение представлено 16 младшими битами 32 битного непосредственного значения
	==REF_HIGH8	5	смещение представлено 8 старшими битами 16 битного непосредственного значения
	==REF_HIGH16	6	смещение представлено 16 старшими битами 32 битного непосредственного значения
target	==target	пояснения	
	!=BADADDR	целевое смещение	
	==BADADDR	вычислять целевое смещение автоматически по следующей формуле $target = operand\_value - tdelta + base$	
base	базовый адрес сегмента, выраженный в байтах (не параграфах!)		
tdelta	относительное смещение, считаемое относительно целевого смещения; может быть как положительным, так и отрицательным		
return	=return	пояснения	
	==1	успешное завершение операции	
	==0	ошибка	

**Родственные функции:** OpOff

**Интерактивный аналог:** “~Edit\Operad types\User-defined offset”;<Ctrl-R>

### success OpSeg(long ea,int n)

Функция отображает операнд (операнды) в виде имени сегмента, базовый адрес которого равен значению операнда. Если сегмента с таким базовым адресом не существует, функция возвращает ошибку.

**Замечание:** в процессе загрузки файла IDA автоматически преобразует все перемещаемые элементы в базовые адреса соответствующих сегментов.

Пример использования:

```
seg000:0000          mov ax, 1000h
```

а) исходные данные – требуется представить непосредственный операнд, загружаемый в регистр ax в виде имени сегмента

```
OpSeg (SegByName ("seg000"), 1);
```

б) вызов функции OpSeg для преобразования непосредственного операнда в имя сегмента с соответствующим базовым адресом

```
seg000:0000          mov ax, seg seg000
```

с) результат – непосредственный операнд теперь представлен в виде имени сегмента с соответствующим базовым адресом

??? #Верстальщику – change table

аргумент	пояснения
----------	-----------

ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пояснения
	==1	операция выполнена успешно
	==0	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** "~ Edit\Operand types\ Segment" <S>

success OpAlt(long ea,long n,char str)

Функция отображающая операнды в виде символьной строки, заданной пользователем. Никаких ограничений на переданную строку не налагается – она может содержать любые символы, кроме символа с кодом нуля, служащим признаком конца строки.

Пример использования:

```
seg000:0000                mov     ax, 9
а) исходные данные
```

OpAlt(SegByName("seg000"),0,"Регистр АХ");

б) вызов функции OpAlt для переименования первого слева операнда в строку «Регистр АХ».

```
seg000:0000                mov     Регистр АХ, 9
с) результат – операнд успешно переименован
```

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пояснения
	==1	операция выполнена успешно
	==0	ошибка

**Родственные функции:** AltOp

**Интерактивный аналог:** "~ Edit\Operand types\ Enter operand manually";<Alt-F1>

success OpSign(long ea,int n)

Функция отображает операнд в знаковой или целочисленной форме, работая как триггер – если до ее вызова операнд отображался в целочисленной форме, после станет отображаться в знаковой и, соответственно, наоборот.

Пример использования:

```
seg000:0000 mov ax, 0FFFFh
```

а) исходные данные – требуется отобразить непосредственное значение, загружаемое в регистр AX в знаковой форме

```
OpSign(SegByName("seg000"), 1);
```

б) вызов функции OpSign для отображения непосредственного значения, загружаемого в регистр AX в знаковой форме

```
seg000:0000 mov ax, -1
```

с) результат - непосредственное значение, загружаемое в регистр AX теперь отображается в знаковой форме.

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
	== -1	все операнды
return	=return	пояснения
	==1	операция выполнена успешно
	==0	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** “~Edit\Operand types\ Change Sign”; <->

success OpStkvar(long ea,int n)

Функция отображает непосредственное значение, используемые для базовой адресации относительно регистров BP (EBP) и SP (ESP) в виде стековой переменной. Сама стековая переменная должна быть предварительно создана вызовом MakeLocal (см. описание функции MakeLocal).

Значение регистров BP (EBP) и SP (ESP) IDA в каждой точке программы IDA по возможности определяет автоматически, облегчая тем самым анализ кода, генерируемого оптимизируемыми компиляторами, использующими для адресации локальных переменных регистр SP (ESP) значение которого подвержено частым изменениям. Для ручного задания значения регистра SP (ESP) предусмотрена функция SetSpDiff, к вызову которой приходится прибегать в случае невозможности определить значение стекового регистра автоматическим анализатором.

**Замечание:** IDA эмулирует выполнения некоторых наиболее употребляемых инструкций, таких как PUSH, POP, ADD, SUB и т.д., для отслеживания изменения значения регистра SP (ESP). Более сложные операции с регистрами пока не поддерживаются.

Пример использования:

```
seg000:0000 start proc near
seg000:0000 mov bp, sp
seg000:0002 sub sp, 10h
```

```

seg000:0005          mov     word ptr [bp-2], 666h
seg000:000A          add     sp, 10h
seg000:000D          retn
seg000:000D start    endp

```

а) исходные данные – требуется представить непосредственное значение, вычитаемое из регистра bp в виде имени локальной переменной.

```
MakeLocal (SegByName ("seg000"), SegByName ("seg000")+0xD, "[BP-2]", "MyVar");
```

б) вызов функции MakeLocal (см. описание MakeLocal) для создания локальной переменной MyVar, расположенной двумя байтам «выше» конца кадра стека

```
OpStkvar (SegByName ("seg000"), 0);
```

с) вызов функции OpStkvar для отображения непосредственного значения в виде имени ранее созданной локальной переменной

```

seg000:0100 start    proc near
seg000:0100          = word ptr -2
seg000:0100 MyVar
seg000:0100          mov     bp, sp
seg000:0102          sub     sp, 10h
seg000:0105          mov     [bp+MyVar], 666h
seg000:010A          add     sp, 10h
seg000:010D          retn
seg000:010D start    endp

```

д) результат – непосредственное значение отображено в виде имени локальной переменной MyVar (в тексте она выделена жирным шрифтом)

**Замечание:** подробнее о поддержке локальных переменных можно прочитать в главе «Функции»

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	=0	первый слева операнд
	=1	второй слева операнд
	=-1	все операнды
return	=return	пояснения
	=1	операция выполнена успешно
	=0	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** "Edit\Operand types\ Stack variable"; <K>

char GetOpnd(long ea, long n)

Функция возвращает операнд в строковом виде, т.е. том виде, в каком дизассемблер отображает его на экране.

Пример использования:

```
seg000:0000          mov     ax, 9
```

а) исходные данные – требуется получить операнды в том виде, в котором они

отображены на экране.

```
Message (">%s, %s\n", GetOpnd (SegByName ("seg000"), 0),  
GetOpnd (SegByName ("seg000"), 1));
```

b) вызов функции GetOpnd для получения операндов в том виде, в котором они отображены на экране

```
>ax, 0
```

c) результат

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	==0	первый слева операнд
	==1	второй слева операнд
return	=return	пояснения
	==1	операция выполнена успешно
	==0	ошибка

**Родственные функции:** GetOpType, GetOperandValue

**Интерактивный аналог:** нет

char AltOp (long ea, long n)

Функция возвращает операнд, определенный пользователем (см. описание функции OpAlt).

```
seg000:0000 mov Регистр AX, 9
```

a) исходные данные

```
Message (">%s\n", AltOp (SegByName ("seg000"), 1));
```

b) вызов функции AltOp для получения операнда, определенного пользователем

```
>Регистр AX
```

c) результат – получен операнд, определенный пользователем

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	==0	первый слева операнд
	==1	второй слева операнд
return	=return	пояснения
	==1	операция выполнена успешно
	==0	ошибка

**Родственные функции:** OpAlt

**Интерактивный аналог:** нет

### long GetOpType (long ea,long n)

Функция возвращает тип операнда (см. таблицу ???), принадлежащему элементу кода (не данных!). Тип операнда, за исключением типов определенных для всех процессоров, зависит от выбранного микропроцессора.

Тип операнда определяется не его представлением на экране, а инструкциями, в состав которых он входит. Так, например, при попытке определения второго слева операнда конструкции "mov dx,offset MyLabel" функция вернет тип непосредственное значение, несмотря на то, что он представлен в виде смещения.

<b>Общие для всех процессоров</b>		
#	тип операнда	
1	регистр общего назначения	
2	ячейка памяти	
3	базовый регистр + [индексный]	
4	базовый регистр + [индексный] + смещение	
5	непосредственное значение	
6	непосредственный far-адрес	
7	непосредственный near-адрес	
<b>Intel 80x86</b>		
#	тип операнда	
8	386+ трассировочный регистр	
9	386+ отладочный регистр	
10	386+ контрольный регистр	
11	Регистр FPP (сопроцессора)	
12	MMX регистр	
<b>8051</b>		
#	тип операнда	
8		
9	бит	
10		
<b>80196</b>		
#	тип операнда	
8		
9	[внутренняя память]	
10	смещение[внутренняя память]	
<b>ARM</b>		
#	тип операнда	
8	регистр сдвига	
9	MLA-операнд	
10	регистр (для LDM/STM)	
11	регистр сопроцессора	
12		CDP LDC/STC
<b>Power PC</b>		
#	тип операнда	
8	регистр указателя стека	
9	регистры плавающей запятой	
10	SH & MB & ME	
11	CR	поле бит
<b>TMS320C5</b>		
#	тип операнда	
8	спарка регистров (A1:A0..B15:B14)	

Z8	
#	тип операнда
8	@внутренняя память
9	@Rx
Z80	
#	тип операнда
8	условие

**Таблица 15**

Пример использования:

```
seg000:0000 mov ax, 9
```

а) исходные данные – требуется определить тип обоих операндов

```
Message(">%x, %x\n", GetOpType(SegByName("seg000"), 0),
        GetOpType(SegByName("seg000"), 1));
```

б) вызов функции GetOpType для определения типов операндов

>1, 5

с) результат – по таблице ??? определяем тип операндов – регистр общего назначения и непосредственное значение соответственно.

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	==0	первый слева операнд
	==1	второй слева, третий (если он есть) и все остальные операнды
return	=return	пояснения
	>1	тип операнда (см. таблицу ???)
	==0	элемент не имеет операндов
	==BADADDR	ошибка

**Родственные функции:** GetOpnd, GetOperandValue

**Интерактивный аналог:** *нет*

longGetOperandValue(long ea, long n)

Функция возвращает значение непосредственного операнда, принадлежащему элементу кода (не данных!), т.е. типу #5 (см. описание функции GetOpType).

Пример использования:

```
seg000:0000 mov ax, 9
```

а) исходные данные – требуется получить значение непосредственного операнда

```
Message(">%x\n", GetOperandValue(SegByName("seg000"), 1));
```

б) вызов функции GetOperandValue для получения значения непосредственного операнда



>9

с) результат – значение непосредственного операнда

??? #верстальщику – change table

аргумент	пояснения	
ea	линейный адрес элемента, которому принадлежит операнд	
n	=n	пояснения
	==0	первый слева операнд
	==1	второй слева операнд
return	=return	пояснения
	==1	операция выполнена успешно
	==0	ошибка

**Родственные функции:** GetOpnd, GetOpType

**Интерактивный аналог:** *нет*

long FindVoid (long ea,long flag)

Функция ищет ближайший к переданному ей линейному адресу **ea** операнд типа “void”, возвращая в случае успешного завершения поиска адрес головы элемента кода, которому он принадлежит. В зависимости от флага направления поиск может идти как вперед (от младших адресов к старшим), так и назад (от старших адресов к младшим). Переданный функции линейный адрес в этот диапазон поиска **не входит** и не обязательно должен принадлежать какому-нибудь сегменту.

Аргумент **flag** задает направление поиска – если его младший бит установлен поиск идет от младших адресов к старшим и, соответственно, наоборот.

Пример использования:

```
seg000:0100          mov     ax, 9
seg000:0103          mov     dx, 133h
```

а) исходные данные – требуется получить линейный адрес элемента, содержащего операнд типа “void”

```
Message(">%s\n", atoa (FindVoid(0,1)) );
```

б) вызов функции FindVoid – адрес начала поиска равен нулю, единичное значение флага направление указывает вести поиск с увеличением адресов

```
>seg000:0103
```

результат – линейный адрес элемента, содержащего операнд типа void, найден

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес начала поиска, не обязательно принадлежащий какому-нибудь сегменту	
flag	=flag	пояснения
	==1	прямое направление поиска
	==0	обратное направление поиска
return	=return	пояснения
	!=BADADDR	линейный адрес элемента, которому принадлежит найденный операнд

	==BADADDR	ошибка
--	-----------	--------

**Родственные функции:** FindImmediate

**Интерактивный аналог:** "~Nabigate\Search for\Next void"; <Ctrl-V>

long FindImmediate(long ea,long flag,long value)

Функция ищет ближайший к переданному ей линейному адресу **ea** операнд типа константа со значением равным **value**. В случае успешного поиска возвращается адрес головы элемента кода, которому этот операнд принадлежит.

В зависимости от флага направления поиск может идти как вперед (от младших адресов к старшим), так и назад (от старших адресов к младшим). Переданный функции линейный адрес в этот диапазон поиска **не входит** и не обязательно должен принадлежать какому-нибудь сегменту.

Аргумент **flag** задает направление поиска – если его младший бит установлен поиск идет от младших адресов к старшим и, соответственно, наоборот.

Пример использования:

```
seg000:0100          mov     ax, 9
seg000:0103          mov     dx, 133h
```

а) исходные данные – требуется получить линейный адрес элемента, содержащего операнд типа константа, значение которой равно 9

```
Message(">%s\n", atoa(FindImmediate(0,1,9)));
```

б) вызов функции FindImmediate – адрес начала поиска равен нулю, единичное значение флага направление указывает вести поиск с увеличением адресов.

```
>seg000:0100
```

результат – линейный адрес элемента, содержащего операнд типа константа, значение которой равно 9

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес начала поиска, не обязательно принадлежащий какому-нибудь сегменту	
flag	=flag	пояснения
	==1	прямое направление поиска
	==0	обратное направление поиска
value	искомое значение константы	
return	=return	пояснения
	!=BADADDR	линейный адрес элемента, которому принадлежит найденный операнд
	==BADADDR	ошибка

**Родственные функции:** FindVoid

**Интерактивный аналог:** "~Nabigate\Search for\Immediate"; <Alt-I>, "~Nabigate\Search for\Next Immediate"; <Ctrl-I>

## ОБЪЕКТЫ

### #Definition

С каждым элементом (бестиповым байтом) могут быть связаны три **объекта** – метка, перекрестная ссылка и комментарий. IDA поддерживает два типа меток – метки, **определенные пользователем** и метки, **автоматически сгенерированные IDA**, а так же четыре типа комментариев – **постоянный** комментарий, отображаемый справа от элемента и отделяемый от него знаком «точка с запятой» (обычный ассемблерный комментарий), **повторяемый** комментарий, отображаемый справа от комментируемого элемента и возле всех ссылок на данный элемент, и два вида многострочных комментариев **предваряющих** и **закрывающих** комментируемый элемент. О перекрестных ссылках подробно рассказано в главе «Перекрестные ссылки».

Каждый элемент может иметь не более одной метки и до четырех комментариев различного типа одновременно. Метки и комментарии хранятся в отдельном виртуальном массиве, проиндексированном линейными адресами, а на наличие связанных с элементом (бестиповым байтом) объектов указывают флаги (см. таблицу 16)

В принципе без флагов, ссылающихся на объекты можно было бы и обойтись, но тогда бы пришлось при отображении каждой ячейки просматривать все виртуальные массивы на предмет поиска объектов, ассоциированных с данным линейным адресом, что отрицательно сказалось бы на производительности дизассемблера. Напротив, перенос этой информации в флаги позволяет ускорить работу – обращение к виртуальному массиву происходит только в тех случаях, когда с ячейкой заведомо связан какой-то объект

Разрушение элемента не вызывает автоматического уничтожения связанных с ним объектов – каждый объект должен быть удален по отдельности соответствующими функциями.

константа	#	пояснения
FF_COMM	0x00000800	комментарий
FF_REF	0x00001000	перекрестная ссылка
FF_LINE	0x00002000	много строчечный комментарий
FF_NAME	0x00004000	метка, определенное пользователем
FF_LABL	0x00008000	метка, автоматически сгенерированное IDA
FF_FLOW	0x00010000	перекрестная ссылка с предыдущей инструкции
FF_VAR	0x00080000	переменная

Таблица 16 Флаги, указывающие на наличие связанных объектов

### Сводная таблица функций

функции, создающие и уничтожающие объекты	
имя функции	краткое описание
success MakeName (long ea, char name)	создает метку
success JmpTable (long jmp ea, long table ea, long nitems, long is32bit)	создает таблицу переходов
success MakeComm (long ea, char comment)	создает постоянный комментарий
success MakeRptCmt (long ea, char comment)	создает повторяемый комментарий

void ExtLinA (long ea,long n, char line)	создает строку комментария перед элементом
void ExtLinB (long ea,long n, char line);	создает строку комментария за элементом
void DelExtLnA (long ea, long n)	удаляет строку комментария перед элементом
void DelExtLnB (long ea, long n)	удаляет строку комментария за элементом
void MakeVar(long ea)	помечает элемент, флажком «переменная»
<b>функции, возвращающие элементы</b>	
имя функции	краткое описание
char Name (long ea)	возвращает имя метки, при необходимости выполняя замену недопустимых символов
char GetTrueName (long ea)	возвращает имя метки
char Comment (long ea)	возвращает постоянный комментарий
char RptCmt (long ea)	возвращает повторяемый комментарий
char LineA (long ea,long num);	возвращает строку комментария, стоящего до элемента
char LineB (long ea,long num);	возвращает строку комментария, стоящего за элементом
<b>функции, поиска объектов</b>	
имя функции	краткое описание
long LocByName (char name)	возвращает линейный адрес метки с заданным именем

#### success MakeName(long ea,char name)

Функция создает метку, расположенную по линейному адресу **ea**, с именем **name**. Переданный линейный адрес должен быть либо адресом головы элемента любого вида, либо адресом бестипового байта; в противном случае функция возвратит ошибку. Имя метки должно состоять только из допустимых символов, перечень которых для каждой платформы содержится в поле "NameChars" конфигурационного файла <ida.cfg>.

платформа	перечень символов, допустимых в именах меток
PC	"\$?@" <sup>9</sup>
	"_0123456789"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz";
Java	"\$ _@?!" <sup>10</sup>
	"0123456789<>"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz"
	"АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ" <sup>11</sup> "абвгдежзийклмнопрстуфхцчшщъыьэюя";
TMS320C6	"\$ 0123456789"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz"
	"_0123456789."
PowerPC	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz"

<sup>9</sup> Служебные символы ассемблера

<sup>10</sup> Символы, определенные **только** для специальных режимов Java-ассемблера

<sup>11</sup> Национальные (русские) символы

## Таблица 17 перечень символов, допустимых в именах меток

Если по указанному адресу расположена уже существующая метка, в результате работы функции она будет переименована.

Удалить метку можно, переименовав ее, в пустую строку. Удаление возможно только в том случае, если во всем дизассемблируемом тексте на данную метку нет ни одной ссылки, в противном случае IDA Pro тут же создаст новое автогенерируемое (*dummy*) имя.

**Замечание:** “*MakeName*” помимо переименования меток, так же изменяет имена функций, если ей передать адрес начала функции (см. главу «Функции»)

Пример использования:

```
seg000:0000                                mov     ah, 9
```

а) исходные данные – требуется создать метку с именем “NoName” по адресу seg000:000

```
MakeName (SegByName (“seg000”), “NoName”);
```

б) вызов функции *MakeName* для создания метки

```
seg000:0000 NoName                        mov     ah, 9
```

с) результат – метка успешно создана

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес головы элемента любого вида или бестипового байта	
name	имя метки	
return	=return	пояснения
	==1	успешное завершение операции
	==0	ошибка

**Родственные функции:** *GetTrueName*

**Интерактивный аналог:** “~Edit\Name” <N>

### success MakeComm(long ea,char comment)

Функция создает комментарий **comment**, размещая его справа от элемента, расположенного по линейному адресу **ea**. Переданный линейный адрес должен быть либо адресом головы элемента любого вида, либо адресом бестипового байта; в противном случае функция возвратит ошибку.

Комментарий автоматически отделяется от элемента символом «точка с запятой» и в самой строке комментария его указывать не нужно. Величина отступа задается настройками IDA (см. главу «Глобальные настройки»).

Строка комментария может содержать как символы латиницы, так и символы кириллицы, однако, нормальное отображение кириллицы возможно только в той ипостаси IDA, в которой они были созданы.

Удалить комментарий можно задав в качестве нового пустую строку. Удаляются в том числе, и некоторые комментарии, автоматически создаваемые IDA.

Функция поддерживает спецификатор переноса строки '\n', автоматически создавая новую строку и перенося на нее хвост комментария.

Пример использования:

```
seg000:0000          mov     ah, 9  
а) исходные данные – требуется вставить комментарий
```

```
MakeComm(0x1275C, "Функция 0x9 – печать строки");  
б) вызов функции MakeComm для вставки комментария
```

```
seg000:0000          mov     ah, 9      ; Функция 0x9 – печать строки  
с) результат – вставленный комментарий
```

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес головы элемента любого вида или бестипового байта	
comment	строка комментария	
return	=return	пояснения
	==1	успешное завершение операции
	=0	ошибка

**Родственные функции:** MakeRptCmt, ExrLinA, ExtLinB

**Интерактивный аналог:** “~Edit\Comments\Enter comment”; <:>

success MakeRptCmt(long ea,char comment)

Функция создает повторяемый комментарий **comment**, размещая его справа от элемента, расположенного по линейному адресу **ea**. Переданный линейный адрес должен быть либо адресом головы элемента любого вида, либо адресом бестипового байта; в противном случае функция возвратит ошибку.

Комментарий автоматически отделяется от элемента символом «точка с запятой» и в самой строке комментария его указывать не нужно. Величина отступа задается настройками IDA (см. главу «Глобальные настройки»).

Строка комментария может содержать как символы латиницы, так и символы кириллицы, однако, нормальное отображение кириллицы возможно только в той ипостаси IDA, в которой они были созданы.

Удалить комментарий можно задав в качестве нового пустую строку. Функция поддерживает спецификатор переноса строки '\n', автоматически создавая новую строку и перенося на нее хвост комментария.

Отличие повторяемого комментария от постоянное заключается в том, что повторяемый комментарий автоматически отображается около всех элементов, ссылающихся на элемент, помеченный повторяемым комментарием.

**Замечание:** повторяемый комментарий может оказаться очень полезным на начальной стадии анализа программы, когда осмысленные имена переменным и функциям дать еще затруднительно, но какие-то мысли по поводу их назначения уже имеются, которые и можно высказать в комментарии, автоматически повторяемом возле всех ссылок на эту переменную (функцию), облегчая тем самым исследование кода.

Пример использования:

```
seg000:0100          mov     ah, 9
```

```

seg000:0102      mov     dx, offset aHello
seg000:0105      int     21h          ;
seg000:0107      retn
seg000:0107 ; -----
seg000:0108 aHello      db     'Hello,',0      ; DATA XREF: seg000:0102↑o
seg000:0108                                     ;

```

а) исходные данные – требуется вставить комментарий к метке aHello, автоматически повторяемый возле всех инструкций, ссылающихся на эту метку.

```
MakeRptCmt (SegByName ("seg000")+0x108,"Это повторяемый комментарий");
```

б) вызов функции MakeRptCmt для создания повторяемого комментария

```

seg000:0100      mov     ah, 9
seg000:0102      mov     dx, offset aHello ; Это повторяемый комментарий
seg000:0105      int     21h          ; DOS - PRINT STRING
seg000:0105                                     ; DS:DX -> string terminated by "$"
seg000:0107      retn
seg000:0107 ; -----
seg000:0108 aHello      db     'Hello,',0      ; DATA XREF: seg000:0102↑o
seg000:0108                                     ; Это повторяемый комментарий

```

с) результат – повторяемый комментарий создан – теперь он будет отображаться возле всех элементов, ссылающихся на метку aHello (обратите внимание на текст, выделенный в листинге жирным шрифтом)

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес головы элемента любого вида или бестипового байта	
comment	строка повторяемого комментария	
return	=return	пояснения
	==1	операция выполнена успешно
	==0	ошибка

**Родственные функции:** MakeComm, ExrLinA, ExtLinB

**Интерактивный аналог:** "Edit\Comments\Enter repeatable comment"; <;

void ExtLinA(long ea,long n,char line)

Функция создает строку (или несколько строк) комментариев, отображаемых перед элементом (бестиповым байтом), расположенном по переданному функции линейному адресу ea.

Комментарий располагается сначала строки и не предваряется символом «точка с запятой», поэтому, его необходимо указать самостоятельно.

Аргумент n задает номер строки комментария и может принимать значения от 0 до 500 включительно. IDA отображает комментарии начиная с нулевой до первой пустой строки. Т. е. если создать нулевую, первую и третью строки комментария, IDA отобразит лишь первые две из них.

Строка комментария может содержать как символы латиницы, так и символы кириллицы, однако, нормальное отображение кириллицы возможно только в той ипостаси IDA, в которой они были созданы.

Для удаления всех строк комментария достаточно присвоить нулевой строке пустое значение – остальные строки хотя физически и останутся в базе, на экране отображаться не будут..

**Замечание:** отсутствие автоматического предварения комментария символом «точка с запятой», позволяет, используя данную функцию, помещать в ассемблерный листинг директивы и инструкции, заданные пользователем.

Пример использования:

```
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0105          int     21h          ;
```

а) исходные данные – требуется разместить комментарий перед инструкцией INT 21h

```
ExtLinA(SegByName("seg000")+0x105,0,"; Строка 1");
ExtLinA(SegByName("seg000")+0x105,1,"; Строка 2");
```

б) вызов функции ExtLinA для создания двух строк комментария

```
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0105 ; Строка 1
seg000:0105 ; Строка 2
seg000:0105          int     21h          ;
```

с) результат

??? #Верстальщику – change table

аргумент	пояснения
ea	линейный адрес головы элемента (бестипового байта) перед которым должен быть размещен комментарий
n	номер строки комментария от 0 до 500 включительно.
line	строка комментария

**Родственные функции:** MakeComm, MakeRptCmt, ExtLinB, DelExtLnA

**Интерактивный аналог:** “~Edit\Comments\Edit extra anterior lines”; <Ins>

void ExtLinB(long ea,long n,char line)

Функция создает строку (или несколько строк) комментариев, отображаемых после элемента (бестипового байта), расположенного по переданному функции линейному адресу ea.

Комментарий располагается сначала строки и не предваряется символом «точка с запятой», поэтому, его необходимо указать самостоятельно.

Аргумент n задает номер строки комментария и может принимать значения от 0 до 500 включительно. IDA отображает комментарии начиная с нулевой до первой пустой строки. Т. е. если создать нулевую, первую и третью строки комментария, IDA отобразит лишь первые две из них.

Строка комментария может содержать как символы латиницы, так и символы кириллицы, однако, нормальное отображение кириллицы возможно только в той ипостаси IDA, в которой они были созданы.

Для удаления всех строк комментария достаточно присвоить нулевой строке пустое значение – остальные строки хотя физически и останутся в базе, на экране отображаться не будут.



**Замечание:** отсутствие автоматического предварения комментария символом «точка с запятой», позволяет, используя данную функцию, помещать в ассемблерный листинг директивы и инструкции, заданные пользователем.

Пример использования:

```
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0105          int     21h          ;
```

а) исходные данные – требуется разместить комментарий после инструкции MOV DX, offset aHello

```
ExtLinB(SegByName("seg000")+0x102,0,"; Строка 1");
ExtLinB(SegByName("seg000")+0x102,1,"; Строка 2");
```

б) вызов функции ExtLinB для создания двух строк комментария

```
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0102 ; Строка 1
seg000:0102 ; Строка 2
seg000:0105          int     21h          ;
```

с) результат

??? #Верстальщику – change table

аргумент	пояснения
ea	линейный адрес головы элемента (бестипового байта) перед которым должен быть размещен комментарий
n	номер строки комментария от 0 до 500 включительно.
line	строка комментария

**Родственные функции:** MakeComm, MakeRptCmt, ExtLinA, DelExtLnB

**Интерактивный аналог:** “~Edit\Comments\Edit extra posterior lines”; <Shift-Ins>

void DelExtLnA(long ea,long n)

Функция удаляет строку **n** много строчного комментария, ранее помещенного перед элементом (бестиповым байтом), расположенным по линейному адресу **ea**. При этом, все строки с номерами, превосходящими **n** (если они существуют) отображаться не будут, но физически по-прежнему будут присутствовать в базе.

Пример использования:

```
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0105 ; Строка 1
seg000:0105 ; Строка 2
seg000:0105 ; Строка 3
seg000:0105          int     21h          ;
```

а) исходные данные – требуется удалить вторую (считая от одного) строку много строчного комментария

```
DelExtLnA(SegByName("seg000")+0x105,1);
б) вызов функции DelExtLnA
```

```

seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0105 ; Строка 1
seg000:0105          int     21h          ;

```

с) результат - все строки, с номерами больше двух (считая от одного) не отображаются на экране

```

ExtLinA(SegByName("seg000")+0x105,1,"; 2");

```

д) вызов функции ExtLinA для восстановления второй строки двух строк комментария

```

seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0105 ; Строка 1
seg000:0105 ; 2
seg000:0105 ; Строка 3
seg000:0105          int     21h          ;

```

е) результат – все строки вновь отображаются на экране

??? #Верстальщику – change table

аргумент	пояснения
ea	линейный адрес элемента (бестипового байта)
n	удаляемая строка комментария (от 0 до 500 включительно)

**Родственные функции:** DelExtLnB

**Интерактивный аналог:** “~Edit\Comments\Edit extra anterior lines”; <Ins>

void DelExtLnB(long ea,long n)

Функция удаляет строку **n** много строчечного комментария, ранее помещенного после элемента (бестипового байта), расположенного по линейному адресу **ea**. При этом, все строки с номерами, превосходящими **n** (если они существуют) отображаться не будут, но физически по-прежнему будут присутствовать в базе.

Пример использования:

```

seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0102 ; Строка 1
seg000:0102 ; Строка 2
seg000:0102 ; Строка 3
seg000:0105          int     21h          ;

```

а) исходные данные – требуется удалить вторую (считая от одного) строку много строчечного комментария

```

DelExtLnB(SegByName("seg000")+0x102,1);

```

б) вызов функции DelExtLnA

```

seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0102 ; Строка 1
seg000:0105          int     21h          ;

```

с) результат - все строки, с номерами больше двух (считая от одного) не отображаются на экране

```
ExtLinB(SegByName("seg000")+0x102,1,"; 2");
```

d) вызов функции ExtLinB для восстановления второй строки двух строк  
комментария

```
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0102 ; Строка 1
seg000:0102 ; 2
seg000:0102 ; Строка 3
seg000:0105          int     21h          ;
```

e) результат – все строки вновь отображаются на экране

??? #Верстальщику – change table

аргумент	пояснения
ea	линейный адрес элемента (бестипового байта)
n	удаляемая строка комментария (от 0 до 500 включительно)

**Родственные функции:** DelExtLnB

**Интерактивный аналог:** “~Edit\Comments\Edit extra posterior lines”; <Shift-Ins>

void MakeVar(long ea)

Функция помечает элемент символом «звездочка», помещая его в начало строки. Повторный вызов функции *не* снимает пометку, и автору книги вообще не известно ни одного программного способа, позволяющего, эту пометку убрать. Интерактивно она снимается вызовом пункта «Mark item as variable» меню “~Edit\Other”, который действует как триггер.

Пример использования:

```
seg000:0000 aHelloIdaPro db 'Hello, IDA Pro! ',0Dh,0Ah
```

a) исходные данные – требуется установить пометку

```
MakeVar(SegByName("seg000"));
```

b) вызов функции MakeVar для пометки

```
seg000:0000*aHelloIdaPro db 'Hello, IDA Pro! ',0Dh,0Ah
```

c) результат – пометка установлена

??? #Верстальщику – change table

аргумент	пояснения
ea	линейный адрес элемента (бестипового байта)

**Родственные функции:** *нет*

**Интерактивный аналог:** “~Edit\Other \Mark item as variable”

char Name(long ea)

Функция возвращает имя метки или функции, расположенной по линейному адресу **ea**, если с данным линейным адресом не связано ни одно имя, функция возвращает пустую

строку, сигнализируя об ошибке.

Функция выполняет проверку на наличие недопустимых символов в имени метки (функции) и при наличии таковых, заменяет их символом, заданным в поле "SubstChar" конфигурационного файла <ida.cfg>. По умолчанию недопустимые символы заменяются знаком «прочерка». Перечень допустимых символов в именах метках определяется значением поля "NameChars" конфигурационного файла <ida.cfg> (см. таблицу 17)

**Замечание:** при отображении имен меток (функций) в окне дизассемблера, IDA Pro всегда заменяет запрещенные символы знаком «прочерка». Т.е. функция Name возвращает имена в том виде, в каком они отображаются на экране. Для получения подлинного имени метки (функции) следует воспользоваться функцией GetTrueName

платформа	перечень символов, допустимых в именах меток (функций)
PC	"\$?@" <sup>12</sup>
	"_0123456789"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz";
Java	"\$ _@?!" <sup>13</sup>
	"0123456789<>"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz"
	"АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ" <sup>14</sup> "абвгдежзийклмнопрстуфхцчшщъыьэюя";
TMS320C6	"\$ _0123456789"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz"
	"_0123456789."
PowerPC	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz"

Таблица 18 перечень символов, допустимых в именах меток

Пример использования:

```
seg000:0000 aHelloIdaPro db 'Hello, IDA Pro! ', 0Dh, 0Ah
```

а) исходные данные – требуется получить имя метки

```
Message(">%s\n", Name(SegByName("seg000")));
```

б) вызов функции Name для получения имени метки

```
> aHelloIdaPro
```

с) результат – имя метки получено

```
??? #Верстальщику – change table
```

аргумент	пояснения	
ea	линейный адрес	
return	=return	пояснения
	!=""	имя метки, в том виде, в котором оно отображено на экране

<sup>12</sup> Служебные символы ассемблера

<sup>13</sup> Символы, определенные **только** для специальных режимов Java-ассемблера

<sup>14</sup> Национальные (русские) символы

	=="	ошибка
--	-----	--------

**Родственные функции:** MakeName, GetTrueName

**Интерактивный аналог:** *имя метки (функции) отображается справа от адреса*

char GetTrueName(long ea)

Функция возвращает полное имя метки (функции), расположенной линейному адресу **ea**, не проверяя его на наличие недопустимых символов и не производя их автоматической замены (см. описание функции Name)

Пример использования:

```
seg000:0000 _HelloIdaPro db 'Hello, IDA Pro! ', 0Dh, 0Ah
```

a) исходные данные – требуется получить подлинное имя метки

```
Message(">%s\n", GetTrueName(SegByName("seg000")));
```

b) вызов функции GetTrueName для получения имени метки

```
>%HelloIdaPro
```

c) результат – подлинное имя метки получено (сравните его с отображаемым на экране)

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес	
return	=return	пояснения
	!="	подлинное имя метки (функции)
	=="	ошибка

**Родственные функции:** MakeName, Name

**Интерактивный аналог:** *нет*

char Comment(long ea)

Функция возвращает строку постоянного комментария, расположенного по линейному адресу **ea**. Если с данным адресом не связан никакой комментарий, функция возвращает пустую строку, сигнализируя об ошибке.

Пример использования:

```
seg000:0000 mov ah, 9 ; Функция 0x9 - печать строки
```

a) исходные данные – требуется получить постоянный комментарий

```
Message(">%s\n", Comment(SegByName("seg000")));
```

b) вызов функции Comment для получения постоянного комментария

```
> Функция 0x9 - печать строки
```

c) результат

??? #Верстальщику – change table

аргумент	пояснения
----------	-----------

ea	линейный адрес	
return	=return	пояснения
	!=""	строка постоянного комментария
	=="	ошибка

**Родственные функции:** MakeComment

**Интерактивный аналог:** *постоянный комментарий отображается справа от элемента*

char RptCmt(long ea)

Функция возвращает строку повторяемого комментария, расположенного по линейному адресу **ea**. Если с данным адресом не связан никакой комментарий, функция возвращает пустую строку, сигнализируя об ошибке.

Пример использования:

```

seg000:0100      mov     ah, 9
seg000:0102      mov     dx, offset aHello ; Это повторяемый комментарий
seg000:0105      int     21h           ; DOS - PRINT STRING
seg000:0105      ; DS:DX -> string terminated by "$"
seg000:0107      retn
seg000:0107 ; _____
seg000:0108 aHello      db     'Hello,',0       ; DATA XREF: seg000:0102;o
seg000:0108      ; Это повторяемый комментарий

```

a) исходные данные – требуется получить строку повторяемого комментария

```
Message (">%s\n", RptCmt (SegByName ("seg000")+0x108) ) ;
```

b) вызов функции RptCmt для получения повторяемого комментария

```
> Это повторяемый комментарий
```

c) результат – строка повторяемого комментария

**Внимание:** функция RptCmt ожидает именно адрес повторяемого комментария, а не адрес элементов, ссылающихся на элемент, связанный с повторяемым комментарием. Т.е. в приведенном выше примере вызов RptCmt (SegByName ("seg000")+0x102) вернул бы пустую строку.

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес	
return	=return	пояснения
	!=""	строка повторяемого комментария
	=="	ошибка

**Родственные функции:** MakeRptCmt

**Интерактивный аналог:** *повторяемый комментарий отображается справа от элемента, и всех ссылок на данный элемент*

char LineA(long ea,long num)

Функция возвращает строку **num** многострочного комментария, помещенного перед элементом, расположенным по линейному адресу **ea**.

Пример использования:

```
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0105 ; Строка 1
seg000:0105 ; Строка 2
seg000:0105          int     21h          ;
```

а) исходные данные – требуется получить первую строку многострочного комментария

```
Message(">%s\n", LineA(SegByName("seg000")+0x105, 0));
```

б) вызов функции LineA для получения первой строки многострочного комментария.

```
> ; Строка 1
```

с) результат

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес	
n	номер строки комментария от 0 до 500 включительно.	
return	=return	пояснения
	!=""	строка повторяемого комментария
	=="	ошибка

**Родственные функции:** LineB

**Интерактивный аналог:** многострочный комментарий отображается перед комментируемым элементом

char LineB(long ea, long num)

Функция возвращает строку **num** многострочного комментария, помещенного за элементом, расположенным по линейному адресу **ea**.

Пример использования:

```
seg000:0100          mov     ah, 9
seg000:0102          mov     dx, offset aHello
seg000:0102 ; Строка 1
seg000:0102 ; Строка 2
seg000:0105          int     21h          ;
```

а) исходные данные – требуется получить первую строку многострочного комментария

```
Message(">%s\n", LineB(SegByName("seg000")+0x102, 0));
```

б) вызов функции LineB для получения первой строки многострочного комментария.

```
> ; Строка 1
```

с) результат

??? #Верстальщику – change table

аргумент	пояснения	
ea	линейный адрес	
n	номер строки комментария от 0 до 500 включительно.	
return	=return	пояснения
	!=	строка повторяемого комментария
	==	ошибка

**Родственные функции:** LineA

**Интерактивный аналог:** *многострочечный комментарий отображается перед комментируемым элементом*

long LocByName(char name)

Функция возвращает линейный адрес метки (имени функции) с именем **name**. Если ни одной метки (функции) с указанными именем не существует, функция возвращает значение BADADDR, сигнализируя об ошибке.

Функция чувствительна к регистру символов и различает имена, набранные строчечными и прописными буквами.

**Внимание:** функции требуется передавать подлинные имена меток, а не имена, отображаемые на экране, прошедшие через фильтр замены недопустимых символов (см. описание функции *GetTrueName*)

Пример использования:

```
seg000:0000 aHelloIdaPro db 'Hello, IDA Pro! ',0Dh,0Ah
```

а) исходные данные – требуется получить адрес метки “aHelloldaPro”

```
Message(">%s\n", atoa(LocByName("aHelloIdaPro")));
```

б) вызов функции LocByName для получения адреса метки

```
>seg000:0000
```

с) результат – адрес метки “aHelloldaPro”

??? #Верстальщику – change table

аргумент	пояснения	
name	имя метки (функции) с учетом регистра	
return	=return	пояснения
	!=BADADDR	линейный адрес метки (функции)
	==BADADDR	ошибка

**Родственные функции:** *нет*

**Интерактивный аналог:** “~View\Names”

??? all – дальше начинается не переработанный вариант

## ФУНКЦИИ



## #Definition

Как только подпрограммы стали неотъемлемой конструкцией любого языка, возникли проблемы с их классификацией.

Начала всему положил BASIC, в котором операторы сплошь и рядом спутаны с переменными, функции с операторами, а подпрограммы представляют наименее развитую конструкцию языка.

Затем было предложено называть подпрограмму, не возвращающую результатов своей работы **процедурой**, а возвращающую **функцией**. Именно такая классификация и была использована в языке Pascal.

Разумеется, было много путаницы. Мало того, что трудно представить себе процедуру, **совсем** ничего не возвращающую. По крайней мере она выводит что-то на экран, или в порты ввода-вывода, пускай даже меняет значения глобальных переменных – то есть все-таки что-то **возвращает**.

Потом, любая процедура имеет доступ к аргументам, передаваемым по ссылке, и может их модифицировать, даже если она ничего **не возвращает** с точки зрения **языка**.

Иными словами выражение:

```
Resultant := MyProc (arg1, ard2);
```

С точки зрения языка Pascal бессмысленно, процедура не может ничего возвращать.

Если опуститься на уровень реализации компилятора Turbo-Pascal, то грубо говоря, функции возвращают результат своей работы в регистре AX, а процедуры оставляют его неопределенным.

То есть другими словами, функцией называется то, что возвращает результат своей работы в регистре AX.

В языке Си все совсем иначе. Не зависимо от того, возвращает что ни будь подпрограмма или нет, она называется **функцией**.

Процедур в Си нет. Функция всегда что-то возвращает, по крайней мере бестиповое значение void, которые **можно** присвоить переменной.

Не будем вспоминать сейчас те языки, где классификация подпрограмм еще более запутана или вовсе не развита.

Сейчас же важно представить себе, как будет работать со всем этим IDA. Знает ли она что-нибудь об этих языковых конструкциях?

Для ответа на этот вопрос необходимо уточнить, что нужно понимать под термином «знает».

Прежде всего она, как и любой другой дизассемблер явно поддерживает только те конструкции, которые «понимает» целевой ассемблер.

С другой стороны, автоматический анализатор способен распознавать различные языковые конструкции, которые могут напрямую и не поддерживаться ассемблером.

Популярные ассемблеры MASM и TASM кроме низкоуровневой поддержки подпрограмм, обеспечивают ряд типовых операций с ними, такими как, например, передача аргументов через стек и обращения к ним, абстрагируясь от модели памяти (все вычисления ложатся на плечи ассемблера).

Однако, эту конструкцию создатели языка почету-то окрестили процедурой, чем окончательно всех запутали. С одной стороны ассемблер не может автоматически присваивать переменным и регистрам результат, возвращенный процедурой, но с другой это еще никак не означает, что процедура не может ничего возвращать.

Воистину, «называемое дао, не есть дао». Впрочем, само по себе такой расклад вещей достаточно приемлем – поскольку, ассемблер, как и любой другой язык, имеет свои соглашения и может совсем не интересоваться, как та же конструкция называется у других.

Но ведь IDA это не просто дизассемблер, но и немного декомпилятор, то есть мотивом ее использования, зачастую служит необходимость получить хотя бы отдаленное представление об исходном тексте программы, пусть и в другой форме.

Однако, поддерживать отдельные языковые конструкции было бы нецелесообразно, да и в ряде случаев и вовсе невозможно. Поэтому было принято решение остановиться на одной универсальной конструкции, которая бы с успехом подходила для любого языка.

Названа же она была **функцией**. Но в ассемблере не существует такого понятия! Поэтому для выражения функции приходится пользоваться средствами ассемблера, в котором она (функция) называется процедурой, но ничем другими принципиально, кроме названия не отличается.

Это иногда приводит к небольшой путанице. Так, например, если дизассемблировать программу, написанную на Turbo-Pascal, то IDA автоматически распознает все процедуры, но называться теперь они будут функциями, а выделяться в ассемблерном листинге ключевым словом PROC (сокращение от procedure)

Пусть исходная программа выглядела так:

```
Procedure MyProc;
begin
    WriteLn('Hello');
end;

BEGIN
MyProc;
End.
```

Тогда результат работы IDA может выглядеть следующим образом:

```
seg000:0006 ; Attributes: bp-based frame
seg000:0006
seg000:0006 sub_0_6      proc near          ; CODE XREF:
PROGRAM+14p
seg000:0006         push     bp
seg000:0007         mov     bp, sp
seg000:0009         xor     ax, ax
seg000:000B         call    @__StackCheck$q4Word ; Stack
overflow check (AX)
seg000:0010         mov     di, offset unk_E1_166
seg000:0013         push   ds
seg000:0014         push   di
seg000:0015         mov     di, offset asc_0_0 ;
"\x05Hello"
seg000:0018         push   cs
seg000:0019         push   di
seg000:001A         xor     ax, ax
seg000:001C         push   ax
seg000:001D         call   @Write$qm4Textm6String4Word ;
Write(var f; s: String; width:
seg000:0022         call   @WriteLn$qm4Text ;
WriteLn(var f: Text)
seg000:0027         call   @__IOCheck$qv ; Exit if
error
seg000:002C         pop    bp
seg000:002D         retn
seg000:002D sub_0_6      endp
seg000:002D
seg000:002E         assume ss:seg004
```

```

seg000:002E PROGRAM      proc near
seg000:002E              call    @__SystemInit$qv ;
__SystemInit(void)
seg000:0033              call    sub_5_D
seg000:0038              push   bp
seg000:0039              mov    bp, sp
seg000:003B              xor    ax, ax
seg000:003D              call   @__StackCheck$q4Word ; Stack
overflow check (AX)
seg000:0042              call   sub_0_6
seg000:0045              pop    bp
seg000:0046              xor    ax, ax
seg000:0048              call   @Halt$q4Word      ; Halt(Word)
seg000:0048 PROGRAM      endp

```

На самом же деле никакого противоречия тут нет. Компиляция однонаправленный процесс с потерями, поэтому можно забыть о существующих конструкциях в языке-источнике.

Код, сгенерированный компилятором одинаково хорошо похож, как на процедуру, так и на функцию:

```

seg000:0006 sub_0_6      proc near
seg000:0006              push   bp
seg000:0007              mov    bp, sp

seg000:0027              call   @__IOCheck$qv
seg000:002C              pop    bp
seg000:002D              retn

```

Поэтому при дизассемблировании принято не акцентировать внимания на подобных различиях и говорить о **подпрограммах**.

Подпрограмма, оформленная особым образом, в IDA называется **функцией**. Но под этим понимается не только совокупность кода и данных, но и действия, которые над ними можно совершить.

Чувствуете разницу? Функцию можно создать, дать ей имя, потом удалить ее, внутри функции IDA может отслеживать значение регистра указателя на верхушку стека и автоматически поддерживать локальные переменные.

При этом в ассемблерном листинге функции оформляются в виде процедур.

```

seg000:0006 sub_0_6      proc near

seg000:002D sub_0_6      endp

```

Противоречия не возникнет, есть понять, что в данном случае под процедурой является один из возможных вариантов предстваления функции, средствами выбранного языка (в данном случае ассемблера MASM)

Таким образом, мы будем говорить о функции не как о совокупности кода и данных, а именно как о **методах взаимодействия** с ней.

### Сводная таблица функций

Имя функции	Назначение
-------------	------------

success MakeFunction(long start,long end);	Создает функцию
success DelFunction(long ea);	Удаляет функцию
success SetFunctionEnd(long ea,long end);	Изменяет линейный адрес конца функции
long NextFunction(long ea);	Возвращает линейный адрес начала следующей функции
long PrevFunction(long ea)	Возвращает линейный адрес начала предыдущей функции
long GetFunctionFlags(long ea);	Возвращает атрибуты функции
success SetFunctionFlags(long ea,long flags);	Устанавливает атрибуты функции
char GetFunctionName(long ea);	Возвращает имя функции
void SetFunctionCmt(long ea, char cmt, long repeatable);	Устанавливает комментарий функции (постоянный и повторяемый)
char GetFunctionCmt(long ea, long repeatable);	Возвращает комментарий функции
long ChooseFunction(char title);	Открывает диалоговое окно со списком всех функций
char GetFuncOffset(long ea);	Преобразует линейный адрес к строковому смещению от начала функции
long GetFrame(long ea);	Возвращает ID фрейма функции
long GetFrameLvarSize(long ea);	Возвращает размер фрейма функции
long GetFrameLvarSize(long ea);	Возвращает размер локальных переменных функции в байтах
long GetFrameArgsSize(long ea)	Возвращает размер аргументов функции в байтах
long GetFrameSize(long ea);	Возвращает полный размер стекового фрейма в байтах
long MakeFrame(long ea,long lvarsize,long frregs,long argsize)	Создает фрейм функции или модифицирует уже существующий
long GetSpd(long ea);	Возвращает значение регистра SP в произвольной точке функции
long GetSpDiff(long ea);	Возвращает относительное изменение регистра SP указанной инструкцией

success SetSpDiff(long ea,long delta);	Корректирует изменение регистра SP, указанной командой
long FindFuncEnd(long ea)	Определяет линейный адрес конца функции

### success MakeFunction(long start,long end);

Вызов MakeFunction позволяет создавать функцию. IDA не различает функций и процедур – в ее терминологии все это функции.

Каждая функция обладает принадлежащим ей непрерывным диапазоном адресов. В его границах может отслеживаться значения указателя стека, создаются ссылки на следующие инструкции и так далее. То есть ряд вызовов API работает исключительно с функциями.

Для создания функции достаточно только указать линейный адрес ее начала и конца. Функции могут создаваться только внутри сегментов и располагаться только с начала, а не середины машинных инструкций.

В то же время допускается в качестве конца задавать адрес, приходящейся на середину инструкции. IDA все равно его округлит до адреса конца предыдущей инструкции.

Например:

```
seg000:002A          mov     si, 211h
seg000:002D          call   sub_0_DD
seg000:0030          mov     si, 2BAh
seg000:0033          call   sub_0_DD
seg000:0036          retn
```

```
MakeFunction(0x1002A, 0x10037);
```

```
seg000:002A ; _____ S U B R O U T I N E
seg000:002A
seg000:002A
seg000:002A sub_0_2A      proc near
seg000:002A          mov     si, 211h
seg000:002D          call   sub_0_DD
seg000:0030          mov     si, 2BAh
seg000:0033          call   sub_0_DD
seg000:0036          retn
seg000:0036 sub_0_2A      endp
seg000:0036
seg000:0037
seg000:0037 ; _____ S U B R O U T I N E
```

При этом функции автоматически дается имя, вид которого зависит от настроек. По умолчанию оно предваряется префиксом 'sub' (от subroutine - то есть процедура; забавно – ведь в терминологии IDA она называется функцией) и последующим смещением внутри сегмента.

Если вместо адреса конца функции указать константу BADADDR, то IDA попытается самостоятельно определить его.

Этот механизм довольно бесхитроуен (концом функции считается инструкция get или jmp) и довольно часто приводит к ошибкам и занижает адрес.

Разберем, например, такой случай. Пусть некая функция имеет более одной точки выхода. В этом случае IDA часто принимает за конец функции первый встретившийся их них, а второй так и остается в «хвосте».

Это не упрек в несовершенстве алгоритма, а просто предостережения от всецелого доверия ему. В действительности же машинный анализ никогда не станет настолько совершенным, что бы полностью заменить человека.

Обратите внимание, что вызов MakeFunction провалится, если выделенная под функцию область будет помечена как undefined. Что и видно из следующего примера:

```
seg000:002A      db 0BEh
seg000:002B      db 11h
seg000:002C      db 2
seg000:002D      db 0E8h
seg000:002E      db 0ADh
seg000:002F      db 0
seg000:0030      db 0BEh
seg000:0031      db 0BAh
seg000:0032      db 2
seg000:0033      db 0E8h
seg000:0034      db 0A7h
seg000:0035      db 0
seg000:0036      db 0C3h

Message("0x%X \n",MakeFunction(0x1002A,0x10037));

0
```

Но в то же время, если в качестве адреса конца указать константу BADADDR, то функция будет успешно создана!

```
seg000:002A      db 0BEh
seg000:002B      db 11h
seg000:002C      db 2
seg000:002D      db 0E8h
seg000:002E      db 0ADh
seg000:002F      db 0
seg000:0030      db 0BEh
seg000:0031      db 0BAh
seg000:0032      db 2
seg000:0033      db 0E8h
seg000:0034      db 0A7h
seg000:0035      db 0
seg000:0036      db 0C3h

Message("0x%X \n",MakeFunction(0x1002A,-1));

1
```

```
seg000:002A ; _____ S U B R O U T I N E
seg000:002A
seg000:002A sub_0_2A      proc near
seg000:002A      mov     si, 211h
```

```

seg000:002D          call     sub_0_DD
seg000:0030          mov     si, 2BAh
seg000:0033          call     sub_0_DD
seg000:0036          retn
seg000:0036 sub_0_2A          endp
seg000:0036
seg000:0037
seg000:0037 ; _____ S U B R O U T I N E

```

Операнд	Пояснения	
Start	Линейный адрес начала функции. Функция не может начинаться с середины инструкции	
End	==end	Пояснения
	!=-1	Линейный адрес конца функции. Может приходиться на середину инструкции. IDA его округлит до адреса конца предыдущей инструкции.
	==-1	IDA автоматически вычисляет адрес конца функции и при необходимости преобразует undefined в инструкции
Return	Завершение	Пояснения
	0	Вызов завершился не успешно. Функция не была создана
	1	Вызов завершился Успешно

### success DelFunction(long ea);

Вызов DelFunction позволяет удалять функцию, указав любой, принадлежащий ей адрес. Вместе с функцией уничтожаются все локальные переменные, и аргументы, если они есть. Все остальное (инструкции, перекрестные ссылки, метки) остается нетронутым. Например:

```

.text:00400FFF ; _____ S U B R O U T I N E
_____
.text:00400FFF
.text:00400FFF ; Attributes: library function
.text:00400FFF
.text:00400FFF __amsg_exit      proc near                ; CODE
XREF: __setenvp+4Ep
.text:00400FFF
_____
__setenvp+7Dp ...
.text:00400FFF
.text:00400FFF arg_0          = dword ptr 4
.text:00400FFF
.text:00400FFF          cmp     dword_0_408758, 2
.text:00401006          jz     short loc_10_40100D
.text:00401008          call   __FF_MSGBANNER
.text:0040100D
.text:0040100D loc_10_40100D:                ; CODE
XREF: __amsg_exit+7j
.text:0040100D          push  [esp+arg_0]

```

```

.text:00401011      call     __NMSG_WRITE
.text:00401016      push    0FFh
.text:0040101B      call    off_0_408050
.text:00401021      pop     ecx
.text:00401022      pop     ecx
.text:00401023      retn
.text:00401023      __amsg_exit      endp

```

DelFunction(0x400FFF);

```

.text:00400FFF      __amsg_exit:      ; CODE
XREF: __setenvp+4Ep
.text:00400FFF      ;
__setenvp+7Dp ...
.text:00400FFF      cmp     dword_0_408758, 2
.text:00401006      jz     short loc_10_40100D
.text:00401008      call   __FF_MSGBANNER
.text:0040100D      loc_10_40100D:  ; CODE
XREF: .text:00401006j
.text:0040100D      push   dword ptr [esp+4]
.text:00401011      call   __NMSG_WRITE
.text:00401016      push   0FFh
.text:0040101B      call   off_0_408050
.text:00401021      pop    ecx
.text:00401022      pop    ecx
.text:00401023      retn

```

Операнд	Пояснения	
ea	Любой линейный адрес, принадлежащий функции	
Return	Завершение	Пояснения
	0	Вызов завершился не успешно. Функция не была создана
	1	Вызов завершился Успешно

### success SetFunctionEnd(long ea,long end);

Позволяет изменить линейный адрес конца функции. Для этого достаточно лишь передать любой адрес, принадлежащий функции и новое значение конца.  
Например:

```

seg000:22C0      start      proc near
seg000:22C0      call     sub_0_22DD
seg000:22C3      call     sub_0_2325
seg000:22C6      call     sub_0_235B
seg000:22C9      call     sub_0_2374
seg000:22CC      call     sub_0_23B6
seg000:22CF      call     sub_0_23F8
seg000:22D2      jnz     loc_0_22DA
seg000:22D4      nop
seg000:22D5      nop

```



```

seg000:22D6          nop
seg000:22D7          call   sub_0_2412
seg000:22DA          loc_0_22DA:
seg000:22DA          call   sub_0_2305
seg000:22DA start    endp

```

```
SetFunctionEnd(0x122C3,0x122CF);
```

```

seg000:22C0 start    proc near
seg000:22C0          call   sub_0_22DD
seg000:22C3          call   sub_0_2325
seg000:22C6          call   sub_0_235B
seg000:22C9          call   sub_0_2374
seg000:22CC          call   sub_0_23B6
seg000:22CF          call   sub_0_23F8 ; → источник
seg000:22CF start    endp
seg000:22D2          jnz   loc_0_22DA ; ← приемник
seg000:22D4          nop
seg000:22D5          nop
seg000:22D6          nop
seg000:22D7          call   sub_0_2412
seg000:22DA          loc_0_22DA:
seg000:22DA          call   sub_0_2305

```

Однако при этом не удаляется перекрестная ссылка на следующую команду, что может иметь неприятные последствия, например, при попытке пометить функцию как undefined, что и видно на следующем примере:

```
MakeUnkn(0x122C0,1);
```

```

seg000:22C0 start    db 0E8h ; ш
seg000:22C1          db 1Ah ;
seg000:22C2          db 0 ;
seg000:22C3          db 0E8h ; ш
seg000:22C4          db 5Fh ; _
seg000:22C5          db 0 ;
seg000:22C6          db 0E8h ; ш
seg000:22C7          db 92h ; Т
seg000:22C8          db 0 ;
seg000:22C9          db 0E8h ; ш
seg000:22CA          db 0A8h ; и
seg000:22CB          db 0 ;
seg000:22CC          db 0E8h ; ш
seg000:22CD          db 0E7h ; ч
seg000:22CE          db 0 ;
seg000:22CF          db 0E8h ; ш
seg000:22D0          db 26h ; &
seg000:22D1          db 1 ;
seg000:22D2          db 75h ; u
seg000:22D3          db 6 ;
seg000:22D4          db 90h ; P
seg000:22D5          db 90h ; P
seg000:22D6          db 90h ; P

```

Кроме того, если в качестве нового конца указать адрес, уже принадлежащий какой-нибудь функции (разумеется, кроме текущей), то вызов провалиться.

```

seg000:2305 sub_0_2305      proc near
seg000:2305                sti
seg000:2306                call   sub_0_1CA
seg000:2309                mov    ah, 4Ch
seg000:230B                int    21h
seg000:230B sub_0_2305      endp
seg000:230B
seg000:230D
seg000:230D ; _____ S U B R O U T I N E
seg000:230D
seg000:230D sub_0_230D      proc near
seg000:230D                mov    si, 2C51h
seg000:2310                call   sub_0_DD
seg000:2313                mov    si, 2C4Dh
seg000:2316                call   sub_0_2E2
seg000:2319                jnb   loc_0_2321
seg000:231B                nop
seg000:231C                nop
seg000:231D                nop
seg000:231E                mov    si, 2A2Dh
seg000:2321
seg000:2321 loc_0_2321:
seg000:2321                call   sub_0_DD
seg000:2324                retn
seg000:2324 sub_0_230D      endp

```

```

Message ("0x%X \n",
SetFunctionEnd(0x12305, 0x12310)
);

```

1

Если функция возвращает отличное от нуля число, то это признак не успешности завершения операции. Следовательно, адрес конца не был изменен. (Повторный дамп для экономии не приводится)

Напротив, если за концом функции расположены данные (можно даже массив), то новый адрес конца будет успешно установлен.

```

seg000:292F sub_0_292F      proc near
seg000:292F                inc    bx
seg000:2930                loop   loc_0_292F
seg000:2932                nop
seg000:2933                retn
seg000:2933 sub_0_292F      endp
seg000:2933
seg000:2933 ; -----
seg000:2934*word_0_2934      dw    0
seg000:2934*
seg000:2936*byte_0_2936     db    0

```

```
SetFuctionEnd(0x12930,0x12934);
```

```
seg000:292F sub_0_292F      proc near
seg000:292F          inc     bx
seg000:2930          loop   loc_0_292F
seg000:2932          nop
seg000:2933          retn
seg000:2933
seg000:2933 ; -----
seg000:2934*word_0_2934  dw 0
seg000:2934*
seg000:2934 sub_0_292F      endp
seg000:2936*byte_0_2936 db 0
```

Можно даже указать на середину массива или ячейки. Функция завершится успешно, адрес будет изменен, но он перестанет отображаться на экране, поскольку IDA забыла его округлить или проверить на корректность!

Это подтверждает следующий пример, проделанный над куском кода, приведенном выше.

```
Message("0x%X \n",
SetFuctionEnd(0x12930,0x12935)
);
```

0

```
seg000:292F sub_0_292F      proc near
seg000:292F          inc     bx
seg000:2930          loop   loc_0_292F
seg000:2932          nop
seg000:2933          retn
seg000:2933
seg000:2933 ; -----
seg000:2934*word_0_2934  dw 0
seg000:2934*
seg000:2936*byte_0_2936 db 0
```

Однако, в действительности, то, что конец функции не отображается на экране, еще ничего не значит. Попробуем убедиться, что IDA действительно не выполнила округления, и адрес 0x12936 не принадлежит функции.

```
Message("0x%X \n",
SetFuctionEnd(0x12936,0x12933)
);
```

1

Ага, вызов SetFuctionEnd возвратил ошибку, следовательно, адрес 0x12936 действительно не принадлежит функции. Попробуем теперь уменьшить его на единицу:

```
Message("0x%X \n",
SetFuctionEnd(0x12935,0x12933)
```

```
);
0

seg000:292F sub_0_292F      proc near
seg000:292F                inc     bx
seg000:2930                loop   loc_0_292F
seg000:2932                nop
seg000:2933                retn
seg000:2933                ; -----
seg000:2934*word_0_2934    dw 0
seg000:2934*sub_0_292F    endp
seg000:2936*byte_0_2936   db 0
```

Выходит, что как это ни парадоксально, но линейный адрес конца функции лежал посередине ячейки word\_02934, где он, разумеется, не мог быть отображен. Описание этой ошибки IDA (которая должна быть устранена в последующих версиях), вероятно, не стоило бы такого пристального внимания, если бы эти таинственные исчезновения конца функций не случались так часто.

Это не нарушает работоспособности дизассемблера, но сбивает с толку пользователя и вызывает в нем мнимое подозрение, что в базе IDA серьезные сбои или ошибки, и что лучше ее удалить и начать проект заново, чем работать с ошибками, которые еще не известно чем могут обернуться в будущем.

На самом деле никаких поводов для беспокойства нет. Необходимо поправить адрес конца функции и можно продолжить работать дальше.

Операнд	Пояснения	
ea	Любой линейный адрес, принадлежащий функции	
end	Новый линейный адрес конца функции.	
Return	Завершение	Пояснения
	0	Вызов завершился не успешно. Функция не была создана
	1	Вызов завершился Успешно

### long NextFunction(long ea);

Вызов возвращает линейный адрес начала функции следующий за 'ea'. Что бы получить адрес первой функции необходимо вызвать NextFunction(0).

Если больше ни одной функции возвратить невозможно, то функция возвращает ошибку BADADDR.

Пример использования:

```
seg000:0000 sub_0_0          proc near
seg000:0000                push   ax
seg000:0001                push   bx
.....
seg000:0027                pop    bx
seg000:0028                pop    ax
seg000:0029                retn
seg000:0029 sub_0_0          endp
```

```

seg000:0029
seg000:002A
seg000:002A ; _____ S U B R O U T I N E
_____
seg000:002A
seg000:002A
seg000:002A sub_0_2A      proc near
seg000:002A                mov     si, 211h
seg000:002D                call   sub_0_DD
seg000:0030                mov     si, 2BAh
seg000:0033                call   sub_0_DD
seg000:0036                retn
seg000:0036 sub_0_2A      endp
seg000:0036
seg000:0037
seg000:0037 ; _____ S U B R O U T I N E
_____
seg000:0037
seg000:0037
seg000:0037 sub_0_37      proc near
seg000:0037
seg000:0037
seg000:0037                push   ax
seg000:0038                push   bx

auto a;
a=0;
while ((a=NextFunction(a))!=-1)
Message("%x \n",a);

10000
1002a
10037

```

Операнд	Пояснения	
ea	Линейный адрес	
Return	Завершение	Пояснения
	!=BADADDR	Линейный адрес начала следующей функции
	BADADDR	Ошибка

### long PrevFunction(long ea)

Вызов возвращает линейный адрес предыдущий функции. Что бы получить адрес последней функции необходимо вызвать PrevFunction(BADADDR).

```

seg000:0000 sub_0_0      proc near
seg000:0000                push   ax
seg000:0001                push   bx
.....
seg000:0027                pop     bx
seg000:0028                pop     ax
seg000:0029                retn
seg000:0029 sub_0_0      endp

```

```

seg000:0029
seg000:002A
seg000:002A ; _____ S U B R O U T I N E
_____
seg000:002A
seg000:002A
seg000:002A sub_0_2A      proc near
seg000:002A                mov     si, 211h
seg000:002D                call   sub_0_DD
seg000:0030                mov     si, 2BAh
seg000:0033                call   sub_0_DD
seg000:0036                retn
seg000:0036 sub_0_2A      endp
seg000:0036
seg000:0037
seg000:0037 ; _____ S U B R O U T I N E
_____
seg000:0037
seg000:0037
seg000:0037 sub_0_37      proc near
seg000:0037
seg000:0037
seg000:0037                push   ax
seg000:0038                push   bx

auto a;
a=0x10038;
while ((a=PrevFunction(a))!=-1)
Message("%x \n",a);

10037
1002a
10000

```

Операнд	Пояснения	
Ea	Линейный адрес	
Return	Завершение	Пояснения
	!=BADADDR	Линейный адрес начала предыдущей функции
	BADADDR	Ошибка

### long GetFunctionFlags(long ea);

Вызов GetFunctionFlags позволяет узнать атрибуты функции. Назначение отдельных битов в возвращаемом значении показано в таблице ниже.

Определение	Значение	Пояснения
FUNC_NORET	0x00000001 L	Функция не возвращает управления
FUNC_FAR	0x00000002 L	FAR (далекая) функция
FUNC_LIB	0x00000004	Библиотечная функция

	L	
FUNC_STATIC	0x00000008 L	Статическая функция
FUNC_FRAME	0x00000010L	Функция использует для указателя кадра стека регистр BP
FUNC_USERFAR	0x00000020 L	Функция определена пользователем как далекая (FAR)
FUNC_HIDDEN	0x00000040 L	Скрытая функция

Подробнее о каждом атрибуте будет рассказано ниже.

## FUNC\_NORET

Этот атрибут устанавливается тем функциям, что не возвращают управления командой `ret`. Однако в большинстве случаев IDA автоматически не присваивает его.

Так, например, в следующей функции он не установлен.

```
seg000:2305 sub_0_2305      proc near
seg000:2305                sti
seg000:2306                call    sub_0_1CA
seg000:2309                mov     ah, 4Ch
seg000:230B                int     21h
seg000:230B sub_0_2305      endp
```

```
Message("%b \n",
GetFunctionFlags(0x12305)
);

0
```

Если в вашей ситуации это обстоятельство окажется критичным, то можно написать собственный скрипт, выполняющий такие проверки и устанавливающий атрибуты через вызов `SetFunctionFlags`.

## FUNC\_FAR

«Далекая» функция. IDA считает далекими все функции, оканчивающиеся командой далекого возврата – `retf`.

Этот механизм достаточно несовершенен и может давать сбои. Так, например, в следующем фрагменте эмуляцию вызова `CALL FAR \ RET` IDA интерпретировала, как далекий возврат из функции.

Это не упрек в сторону IDA, поскольку ради таких частных случаев бессмысленно вносить дополнительные усовершенствования в дизассемблер, но учитывать этот факт всегда стоит, вручную корректируя адрес конца и атрибуты функции.

```
seg000:048B sub_0_48B      proc far
seg000:048B                pushf
seg000:048B                push    cs
seg000:048C                push    offset locret_0_499
seg000:048D                push    word ptr ds:74Dh
seg000:0490                push    word ptr ds:74Bh
seg000:0494                retf
seg000:0498
```

```

seg000:0498 sub_0_48B      endp ; sp = -0Ah
seg000:0498
seg000:0499 ; -----
seg000:0499
seg000:0499 locret_0_499:
seg000:0499      retn

Message("%b \n",
GetFunctionFlags(0x1048B)
);

10

```

## FUNC\_LIB

Таким флагом отмечены стандартные «библиотечные» функции. То есть те, чьи сигнатуры известны FLIRT.

```

.text:004010FF ; Attributes: library function
.text:004010FF
.text:004010FF __amsg_exit      proc near
.text:004010FF
.text:004010FF arg_0          = dword ptr 4
.text:004010FF
.text:004010FF      cmp     dword_0_408758, 2
.text:00401106      jz     short loc_0_40110D
.text:00401108      call  __FF_MSGBANNER
.text:0040110D
.text:0040110D loc_0_40110D:
.text:0040110D      push  [esp+arg_0]
.text:00401111      call  __NMSG_WRITE
.text:00401116      push  0FFh
.text:0040111B      call  off_0_408050
.text:00401121      pop   ecx
.text:00401122      pop   ecx
.text:00401123      retn
.text:00401123 __amsg_exit      endp

Message("%b \n",
GetFunctionFlags(0x4010FF)
);

100

```

## FUNC\_FRAME

Функция использует в качестве указателя на кадр стека регистр BP (EBP). IDA определяет это отслеживая последовательность

```

PUSH BP
MOV BP, SP

```



Большинство современных оптимизирующих компиляторов отказались от такого подхода, и адресуют локальные переменные и аргументы непосредственно по регистру ESP. IDA распознает такую ситуацию и отслеживает значение ESP по ходу исполнения процедуры, освобождая пользователя от лавиной доли работы.

```

seg000:20B8 ; Attributes: bp-based frame
seg000:20B8
seg000:20B8 sub_0_20B8      proc near
seg000:20B8
seg000:20B8
seg000:20B8 var_80          = byte ptr -80h
seg000:20B8 var_6B          = byte ptr -6Bh
seg000:20B8 var_62          = byte ptr -62h
seg000:20B8
seg000:20B8          push    bp
seg000:20B9          mov     ah, 2Fh
seg000:20BB          int     21h
seg000:20BB
seg000:20BD          push    bx
seg000:20BE          mov     bp, sp
seg000:20C0          sub     sp, 80h
seg000:20C4          mov     ah, 1Ah
seg000:20C6          lea    dx, [bp+var_80]
seg000:20C9          int     21h

```

```

Message("%b \n",
GetFunctionFlags(0x4010FF)
);

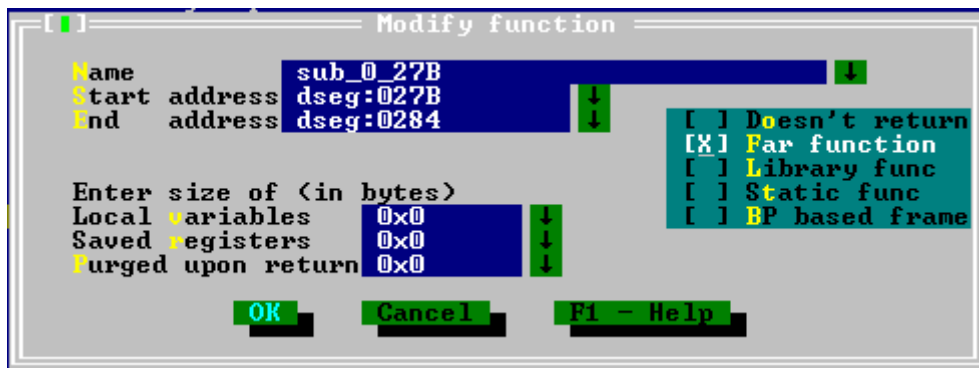
```

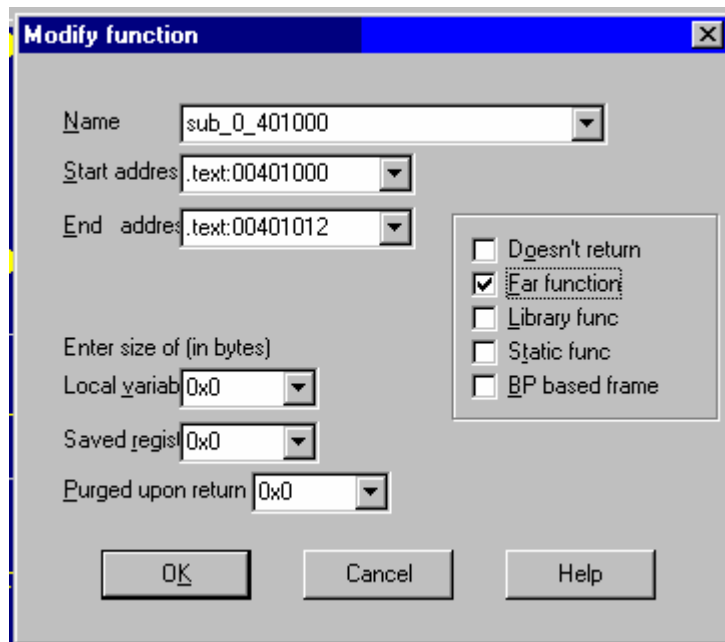
```
10000
```

Обратите внимание, что IDA распознала эту комбинацию, даже когда команды push bp и mov bp,sp оказались достаточно разнесены.

## FUNC\_USERFAR

Этот атрибут IDA устанавливает, когда пользователь вручную меняет тип функции с NEAR на FAR, вызывая диалог 'Modify Function' с помощью команды меню ~ Edit \ Function \ Edit Function.





Обратите внимание, что это не относится к вызовам SetFunctionFlags! Последняя устанавливает именно тот набор атрибутов, который ей передается.

## FUNC\_HIDDEN

Этот атрибут устанавливается у «свернутых» функций. Свернуть любую функцию можно однократным нажатием Gray '-', переместив курсор в ее границы. Кроме этого, в зависимости от настроек, IDA может автоматически сворачивать все библиотечные функции для экономии места.

```
dseg:027B ; [00000009 BYTES: COLLAPSED FUNCTION sub_0_27B. PRESS KEYPAD "+" TO EXPAND]
```

```
Message("%b \n",
GetFunctionFlags(0x4010FF)
);
```

```
100000
```

Заметим, что в результате упущения этот тип не определен в IDC.IDC и что бы им пользоваться необходимо это сделать самостоятельно, внося новую строчку:

```
#define FUNC_HIDDEN      0x00000040L      // a hidden function
```

Все атрибуты функция IDA отображает в виде комментариев, расположенных в начале функции.

```
dseg:0271 ; Attributes: library function
dseg:0271
dseg:0271 __checknull      proc near
```

```

dseg:0271                               retn
dseg:0271 __checknull                    endp

dseg:0272 ; Attributes: library function bp-based frame
dseg:0272
dseg:0272 __terminate                    proc

```

Операнд	Пояснения	
Ea	Линейный адрес, принадлежащий функции	
Return	Завершение	Пояснения
	!=BADADDR	Набор атрибутов функции (смотри таблицу выше)
	BADADDR	Ошибка

### success SetFunctionFlags(long ea,long flags);

Позволяет устанавливать атрибуты функции. Подробнее об этом было сказано в описании функции GetFunctionFlags.

Операнд	Пояснения	
Ea	Линейный адрес, принадлежащий функции	
flag	Атрибуты функции (смотри таблицу в описании GetFunctionFlags)	
Return	Завершение	Пояснения
	!=BADADDR	Набор атрибутов функции (смотри таблицу выше)
	BADADDR	Ошибка

Как уже отмечалось в описании функции SetFunctionFlags, часто IDA автоматически не распознает функции, не возвращающие управления (нет команды ref в завершении функции).

Если это критично, то нужный атрибут можно установить вручную. Покажем это на следующем примере:

```

dseg:0272 ; Attributes: library function bp-based frame
dseg:0272
dseg:0272 __terminate                    proc near                    ; COD
dseg:0272
dseg:0272 arg_0                          = byte ptr 2
dseg:0272
dseg:0272                               mov     bp, sp
dseg:0274                               mov     ah, 4Ch ; 'L'
dseg:0276                               mov     al, [bp+arg_0]
dseg:0279                               int     21h                    ; DOS
dseg:0279 __terminate                    endp                          ; AL

```

```

SetFunctionFilegs
(
    0x10272,
    GetFunctionFlags(0x10272) + 1
)

```

```

dseg:0272 ; Attributes: library function noreturn bp-based frame
dseg:0272
dseg:0272 __terminate      proc near                ; CODE XREF: sub_0_3C7+44p
dseg:0272
dseg:0272 arg_0           = byte ptr 2
dseg:0272
dseg:0272             mov     bp, sp
dseg:0274             mov     ah, 4Ch ; 'L'
dseg:0276             mov     al, [bp+arg_0]
dseg:0279             int     21h                ; DOS - 2+ - QUIT WITH EXIT
dseg:0279 __terminate      endp                    ; AL = exit code

```

В большинстве случаев атрибуты никакого влияния на функции не оказывают. Так, например, если в приведенном примере сбросить флаг `FUNC_FRAME`, то это не повлечет за собой автоматического удаления всех локальных переменных, адресуемых через `BP`.

```
SetFunctionFilegs
```

```

(
    0x10272,
    GetFunctionFlags(0x10272) - 0x10;
)

```

```

dseg:0272 ; Attributes: library function
dseg:0272
dseg:0272 __terminate      proc near                ; CODE XREF: sub_0_3C7+44p
dseg:0272
dseg:0272 arg_0           = byte ptr 2
dseg:0272
dseg:0272             mov     bp, sp
dseg:0274             mov     ah, 4Ch ; 'L'
dseg:0276             mov     al, [bp+arg_0]
dseg:0279             int     21h                ; DOS - 2+ - QUIT WITH EXIT
dseg:0279 __terminate      endp                    ; AL = exit code

```

Но вот установка флага `FUNC_HIDDEN` приведет к незамедлительному сворачиванию функции и сброс соответственно, наоборот.

```
SetFunctionFilegs
```

```

(
    0x10272,
    GetFunctionFlags(0x10272) + 0x40;
)

```

```
dseg:0272 ; [00000009 BYTES: COLLAPSED FUNCTION __terminate. PRESS KEYPAD "+" TO EXPAND]
```

```
char GetFunctionName(long ea);
```

Возвращает имя функции. Если указанный адрес не принадлежит ни одной из функций, то возвращается пустая строка.

Поскольку функции без имени не бывает, то неоднозначной ситуации не возникает.

Операнд	Пояснения	
Ea	Любой линейный адрес, принадлежащий функции	
Return	Завершение	Пояснения

	!=""	Имя функции
	""	Ошибка

Пример использования:

```
dseg:025E __cleanup      proc near
dseg:025E                mov     es, cs:DGROUP@
dseg:0263                push   si
dseg:0264                push   di

Message("%s \n",
GetFunctionName(0x10263)
);

__cleanup
```

**void SetFunctionCmt(long ea, char cmt, long repeatable);**

Задаёт комментарий, который размещается впереди функции. IDA поддерживает символ переноса, поэтому комментарий может располагаться на нескольких строках.

Существует возможность повторять тот же комментарий в точке вызова функции (так называемый repeatable comment). Для этого необходимо установить флаг 'repeatable' равным единице.

Например:

```
SetFunctionCmt(0x10271, "Hello IDA 4.0", 1);

dseg:0271 ; Hello IDA 4.0
dseg:0271 ; Attributes: static
dseg:0271
dseg:0271 __checknull      proc near                ; CODE XREF:
sub_0_3C7+2Cp
dseg:0271                retn
dseg:0271 __checknull      endp
```

Если перейти по перекрестной ссылке к месту вызова функции, то там будет можно обнаружить следующее:

```
dseg:03F0                call   __restorezero
dseg:03F3                call   __checknull      ; Hello IDA
4.0
dseg:03F6                cmp    [bp+arg_2], 0
dseg:03FA                jnz   loc_0_40F
```

Если в строке комментария будет присутствовать символ переноса, то экран будет выглядеть так:

```
SetFunctionCmt(0x10271, "Hello \nIDA 4.0", 1);
```

```

dseg:0271 ; Hello
dseg:0271 ; IDA 4.0
dseg:0271 ; Attributes: static

dseg:03F3          call    __checknull    ; Hello
dseg:03F3          ; IDA 4.0
dseg:03F6          cmp     [bp+arg_2], 0

```

Не рекомендуется перегружать листинг повторяемыми комментариями. Ведь всегда можно обратиться за разъяснениями к самой функции.

Наиболее полезны они на начальной стадии дизассемблирования, когда назначение большинства функций плохо понятны и дать им осмысленное имя никак не удается. Тогда в повторяемом комментарии отражают все, что на данный момент известно о каждой из функций и по мере исследования текста, уточняют. На финальной же стадии дизассемблирования, повторяемые комментарии обычно убирают.

Обратите внимание, каждая функция может обладать комментариями сразу двух типов, но в заголовке будет отображаться только один из них – ‘regular’.

Например:

```

SetFunctionCmt(0x10271,"Hello IDA 4.0",1);
SetFunctionCmt(0x10271,"Hello World",0);

dseg:0271 ; Hello World
dseg:0271 ; Attributes: static

dseg:03F3          call    __checknull    ; Hello IDA 4.0
dseg:03F6          cmp     [bp+arg_2], 0

```

Операнд	Пояснения	
Ea	Любой линейный адрес, принадлежащий функции	
Cmp	Строка комментария, включая символ переноса	
Repeatable	Флаг	Пояснения
	0	Неповторяемый комментарий
	1	Повторяемый комментарий

### char GetFunctionCmt(long ea, long repeatable);

Позволяет получить как повторяемый, так и постоянный комментарии. Для этого необходимо задать любой линейный адрес, принадлежащий функции.

Подробнее о повторяемых комментариях можно прочитать в описании функции

SetFunctionCmt

Например:

```

dseg:0271 ; Hello IDA 4.0
dseg:0271 ; Attributes: static
dseg:0271
dseg:0271 __checknull    proc near
dseg:0271          retn
dseg:0271 __checknull    endp

```

```

Message("%s \n",
GetFunctionCmt(0x010271,1)

```

```
);

Hello, IDA 4.0

Message("%s \n",
GetFunctionCmt(0x010271,0)
);
```

Обратите внимание, что необходимо правильно указывать тип комментария (повторяемый или нет) иначе функция вернет совсем не то, что от нее ожидается.

Операнд	Пояснения	
Ea	Любой линейный адрес, принадлежащий функции	
Repeatable	Флаг	Пояснения
	0	Неповторяемый комментарий
	1	Повторяемый комментарий
Return	Завершение	Пояснения
	!=""	Комментарий
	""	Ошибка

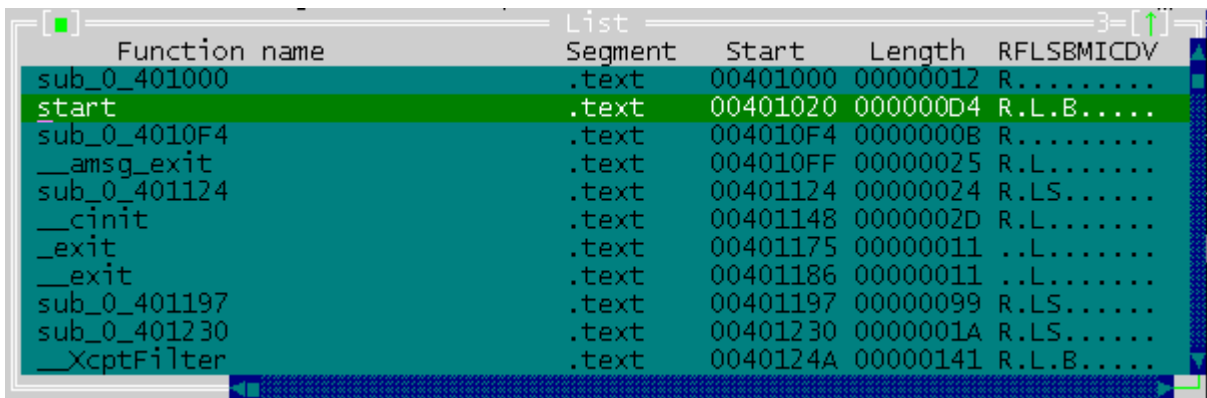
### long ChooseFunction(char title);

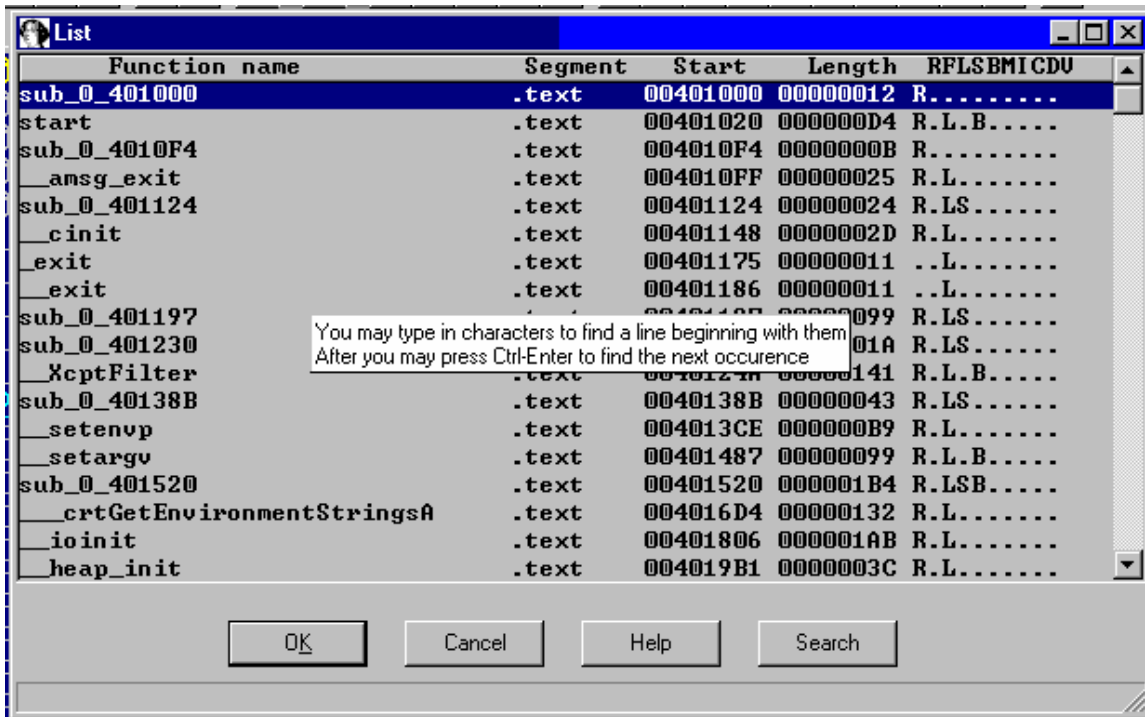
Создает диалоговое окно содержащие список всех существующих функций с краткой сводной информацией о каждой из них.

Возвращает линейный адрес начала выбранной функции или BADADDR, если ни одна функция не была выбрана.

Пример использования:

```
Message("0x%X \n",
ChooseFunction("List")
);
```





0x401020

Поле				
Function Name	Имя функции			
Segment	Сегмент, владеющий функцией			
Start	Линейный адрес начала			
Length	Длина функции в байтах			
RFLSBMICDV	Атрибут	Определение	Пояснение	
	R	IFUNC_NORET	Функция, возвращающая управление	
	F	FUNC_FAR	FAR (Далекая) функция	
	L	FUNC_LIB	Библиотечная функция	
	S	FUNC_STATIC	Static – функция	
	B	FUNC_FRAME	BP используется как указатель на кадр стека	
	*	M	FUNC_MEMBER	member function
	*	I	FUNC_VIRTUAL	Виртуальная функция
	*	C	FUNC_CTR	Конструктор
	*	D	FUNC_DTR	Деструктор
*	V	FUNC_VARARG	Функция с переменным числом аргументов	

\* Не поддерживается в текущих версиях. Зарезервировано для будущего использования.

Подробнее узнать об атрибутах функции можно в описании SetFunctionFlags.

Операнд	Пояснения
title	Заголовок диалогового окна



Return	Завершение	Пояснения
	!=BADADDR	Линейный адрес начала выбранной функции
BADADDR	Ошибка	

По умолчанию подсвечивается функция, в границах которой находится курсор. В противном случае подсвечивается ближайшая функция, лежащая «выше» курсора (то есть в более младших адресах)

Для поиска подстроки в именах функции предусмотрена специальная клавишная комбинация <Alt-T> Регистр символов при этом будет игнорироваться. Для продолжения поиска необходимо нажать <Ctrl-T>.

В контекстной помощи сообщается, что с помощью клавиш <ins> и <delete> можно соответственно добавлять или удалять функции. Но на самом деле в данном случае эти возможности недоступны.

<Enter> или двойной щелчок мышью выбирают функцию и возвращают управление, закрывая диалоговое окно.

### char GetFuncOffset(long ea);

Преобразует линейный адрес к строковому представлению в следующем формате: **ИмяФункции+СмещениеОтносительноНачалаФункции**. Смещение представлено в шестнадцатеричном виде без префиксов и постфиксов.

Например:

```
.text:004010FF __amsg_exit      proc near
.text:004010FF
.text:004010FF
.text:004010FF arg_0          = dword ptr 4
.text:004010FF
.text:004010FF                cmp     dword_0_408758, 2
.text:00401106                jz     short loc_0_40110D
.text:00401108                call   __FF_MSGBANNER
.text:0040110D
.text:0040110D loc_0_40110D:
.text:0040110D                push  [esp+arg_0]
.text:00401111                call  __NMSG_WRITE
.text:00401116                push  0FFh
.text:0040111B                call  off_0_408050
.text:00401121                pop   ecx
.text:00401121 __amsg_exit      endp
.text:00401121
.text:00401122                pop   ecx
.text:00401123                retn
```

```
Message("%s \n",
GetFuncOffset(0x401108)
);
```

\_\_amsg\_exit+9

Операнд	Пояснения	
ea	Линейный адрес, принадлежащий хотя бы одной функции	
Return	Завершение	Пояснения
	!=""	Смещение относительно начала функции (строка)

	“”	Ошибка
--	----	--------

Если смещение относительно функции равно нулю, то вызов GetFuncOffset возвратит только одно имя.

### long FindFuncEnd(long ea);

Описание этой функции, приведенное в контекстной помощи, немного запутанное и с первого взгляда назначение этой функции не ясно.

Но на самом деле она очень находит широкое применение в автономных скриптах. Основное ее назначение – определение линейного адреса при создании функции.

Это сопряжено со следующими трудностями – прежде всего необходимо, что бы адрес конца не превышал линейного адреса следующей за ней функции, поскольку функции перекрываться не могут.

Например:

```

seg000:22C0 start:
seg000:22C0          call    sub_0_22DD
seg000:22C3          call    sub_0_2325
seg000:22C6          call    sub_0_235B
seg000:22C9          call    sub_0_2374
seg000:22CC          call    sub_0_23B6
seg000:22CF          call    sub_0_23F8
seg000:22D2          jnz    loc_0_22DA
seg000:22D4          nop
seg000:22D5          nop
seg000:22D6          nop
seg000:22D7          call    sub_0_2412
seg000:22DA
seg000:22DA loc_0_22DA:          ;
seg000:22DA          call    halt
seg000:22DD
seg000:22DD ; _____ S U B R O U T I N E _____
seg000:22DD
seg000:22DD
seg000:22DD sub_0_22DD      proc near          ;
seg000:22DD          call    sub_0_28CC

```

Функция start не завершается командой возврата ret. Вместо этого она прерывает выполнение программы, процедурой Halt.

Если пытаться создать функцию, определяя линейный адрес ее конца поиском ret, то полученный адрес будет принадлежать функции sub\_0\_22DD!

Следовательно, адрес конца не может превышать линейный адрес следующей функции.

Вторая проблема заключается в отождествлении инструкции возврата. Это может быть все что угодно. И RETN, и RETF...

Таким образом, определение конца функции «вручную» оказывается слишком утомительно. И тогда стоит прибегнуть к вызову FindFuncEnd.

Что она делает? Она возвращает линейный адрес на единицу больший линейного адреса конца функции, которая может быть успешно создана.

Таким образом, задача создания определения адреса конца функции для ее создания упрощается, тем более, что FindFuncEnd ищет не первую встретившуюся ей инструкцию возврата, а последнюю в цепочке перекрестных ссылок на следующую команду (подробнее об этом рассказано в главе «Перекрестные ссылки»).

Отсюда следует тот замечательный факт, что она поддерживает функций со множественными возвратами (а таким, как правило большинство).

Например:

```
seg000:0100 start:
seg000:0100          mov     ax, 3D01h
seg000:0103          mov     dx, 10Fh
seg000:0106          int     21h
seg000:0106
seg000:0106
seg000:0106
seg000:0108          jb     loc_0_10B
seg000:010A          retn
seg000:010B ; -----
seg000:010B
seg000:010B loc_0_10B:
seg000:010B          mov     ax, 0FFFFh
seg000:010E          retn
seg000:010E ; -----
seg000:010F aMyfile  db     'MyFile',0
```

```
Message("0x%X \n",
FindFuncEnd(0x10103)
);
```

0x1010F

Обратите внимание, что IDA эмулировала выполнение команды условного перехода и правильно определила точку выхода из функции.

Однако, если быть уж совсем «буквоедом», то можно заметить, что строка aMyFile вероятнее всего принадлежала функции, но IDA автоматически не смогла это распознать. Поэтому **иногда** результат работы функции все же приходится корректировать.

Очень важный факт – линейный адрес должен указывать на начало команды, иначе вызов провалиться.

Например:

```
Message("0x%X \n",
FindFuncEnd(0x10102)
);
```

0xFFFFFFFF

То же самое произойдет если по указанному адресу будет расположены данные, так что функцию будет создать невозможно.

Если же функция уже существует, то вызов FindFuncEnd так же возврат адрес ее конца:

```
seg000:0100 start          proc near
seg000:0100          mov     ax, 3D01h
seg000:0103          mov     dx, 10Fh
seg000:0106          int     21h
seg000:0106
seg000:0106
seg000:0106
seg000:0106
seg000:0108          jb     loc_0_10B
seg000:010A          retn
```

```

seg000:010B ; -----
seg000:010B
seg000:010B loc_0_10B:
seg000:010B          mov     ax, 0FFFFh
seg000:010E          retn
seg000:010E ; -----
seg000:010F aMyfile      db 'MyFile',0
seg000:010F start      endp

```

```

Message("0x%X \n",
FindFuncEnd(0x10103)
);

```

0x10116

То, что последний отображаемый адрес равен 0x10F это небольшой баг IDA. На самом деле это адрес начала строки, но не ее конца. Как нетрудно вычислить, адрес конца строки равен 0x115, следовательно функция FindFuncEnd работает правильно.

В описании этой функции в idc.idc утверждается, что требуется передать линейный адрес начала функции, но это не так. С таким же успехом функция принимает **любой**, принадлежащий ей адрес, если он приходится на начало инструкции.

Операнд	Пояснения	
ea	Линейный адрес, конца функции	
Return	Завершени е	Пояснения
	!=BADADD R	ID структуры обеспечивающий доступ к локальным переменным и аргументам
	BADADDR	Ошибка

### long GetFrame(long ea);

Возвращает ID фрейма функции (если он есть) или BADADDR в противном случае. Это значение может интерпретироваться только IDA, и с точки зрения пользователя лишено всякого смысла (как и всякий дескриптор)

Все локальные переменные и аргументы объединены в одну структуру, с которой можно работать, как и с любой с помощью функций, описанных в разделе «Структуры»

Если функция не содержит ни одной локальной переменной и не имеет ни одного аргумента, то вызов GetFrame возвратит ошибку BADADDR.

Пример использования:

```

.text:004010FF __amsg_exit      proc near
.text:004010FF
.text:004010FF arg_0          = dword ptr 4
.text:004010FF
.text:004010FF          cmp     dword_0_408758, 2
.text:00401106          jz     short loc_0_40110D
.text:00401108          call   __FF_MSGBANNER
.text:0040110D
.text:0040110D loc_0_40110D:
.text:0040110D          push  [esp+arg_0]
.text:00401111          call  __NMSG_WRITE

```

```

.text:00401116          push    0FFh
.text:0040111B          call   off_0_408050
.text:00401121          pop     ecx
.text:00401122          pop     ecx
.text:00401123          retn
.text:00401123  __amsg_exit  endp

```

```

Message("%x \n",
GetFrame(0x40110D)
);

```

ff000162

Операнд	Пояснения	
ea	Линейный адрес, принадлежащий функции	
Return	Завершение	Пояснения
	!=BADADDR	ID структуры обеспечивающий доступ к локальным переменным и аргументам
	BADADDR	Ошибка

### long GetFrameLvarSize(long ea);

Возвращает размер локальных переменных функции (в байтах). Если функция не имеет локальных переменных, то возвращается ноль. Если указанный адрес не принадлежит ни одной функции, возвращается ошибка BADADDR.

Например:

```

.text:00401806  __ioinit      proc near
.text:00401806  var_44        = byte ptr -44h
.text:00401806  var_12        = word ptr -12h
.text:00401806  var_10        = dword ptr -10h
.text:00401806
.text:00401806          sub     esp, 44h
.text:00401809          push   ebx
.text:0040180A          push   ebp

```

```

Message("0x%X \n",
GetFrameLvarSize(0x401809)
);

```

0x44

Операнд	Пояснения	
Ea	Линейный адрес, принадлежащий функции	
Return	Завершение	Пояснения
	!=0 !=BADADDR	Размер локальных переменных функции в байтах
	0	Функция не имеет локальных переменных
	BADADDR	Ошибка

## long GetFrameRegsSize(long ea);

Возвращает размер сохраненных в стековом фрейме регистров. Для 32-разрядных программ он равен четырем (четыре байта на регистр) и для 16-разрядных соответственно двум (два байта на регистр)

Если функция не имеет кадра стека, то возвращается ноль и BADADDR если указанный адрес не принадлежит ни одной функции.

Пример использования:

```
.text:0040124A __XcptFilter    proc near
.text:0040124A
.text:0040124A arg_0              = dword ptr  8
.text:0040124A arg_4              = dword ptr  0Ch
.text:0040124A
.text:0040124A                push     ebp
.text:0040124B                mov     ebp, esp
.text:0040124D                push     ebx
.text:0040124E                push     [ebp+arg_0]
```

```
Message("0x%X \n",
GetFrameRegsSize(0x40124A)
);
```

4

```
seg000:2092 sub_0_2092    proc far
seg000:2092
seg000:2092 var_40          = byte ptr -40h
seg000:2092
seg000:2092                push    bp
seg000:2093                mov     bp, sp
```

```
Message("0x%X \n",
GetFrameRegsSize(0x12093)
);
```

2

Операнд	Пояснения	
Ea	Линейный адрес, принадлежащий функции	
Return	Завершени е	Пояснения
	!=0 !=BADADD R	Размер сохраненных регистров в стековом фрейме
	0	Функция не имеет кадра стека
	BADADDR	Ошибка

## **long GetFrameArgsSize(long ea);**

Возвращает размер (в байтах) аргументов, передаваемых функции. IDA определяет эту величину на основании анализа команд, очищающих стек от локальных переменных. Обычно компиляторы используют два принципиально различных соглашения для этого.

Паскаль – соглашение предписывает функции самой очищать стек от локальных переменных перед возвратом из функции. Независимо от способа реализации после возврата из функции стек должен быть сбалансирован. На платформе Intel практически всегда для этого используют команду RET N, где N число байт, которые нужно вытолкнуть с вершины стека после возврата. В этом случае IDA просто возвращает аргумент, стоящий после RET.

Например:

```
Pascal_func:
    Push    bp
    Mov     bp, sp
    Mov     ax, [BP+4]
    RET     2
Endp

PUSH    10
CALL    Pascal_func
```

Си – соглашение предписывает очищать локальные переменные вызываемому коду. При выходе из функции стек остается несбалансированным. Поэтому необходимо скорректировать его вершку.

Долгое время не оптимизирующие компиляторы использовали для этого команду ADD SP, N. Где N размер аргументов в байтах. Очевидно, что IDA так же без проблем могла распознать такую ситуацию.

Например:

```
C_func:
    Push    bp
    Mov     bp, sp
    Mov     ax, [BP+4]
    RET
Endp

PUSH    10
CALL    C_func
ADD     SP, 2
```

Но с появлением оптимизирующих компиляторов все изменилось. Они могли выталкивать аргументы командой POP в неиспользуемые регистры или вовсе оставлять стек несбалансированным на то время пока к нему нет обращений. Поэтому в определении размера аргументов стали возможны ошибки.

```
C_optimize_func:
    Push    bp
    Mov     bp, sp
    Mov     ax, [BP+4]
    RET
Endp

PUSH    10
CALL    C_optimize_func
```

```

OR    AX,AX
JZ    xxx
MOV   AX,[BX]
Xxx:
POP   AX
RET

```

Даже для человека с первого взгляда не очевидно назначение команды POP AX. Кроме того, современные компиляторы поддерживают «совмещенные аргументы», что делает задачу определения их размера практически невозможной.

Допустим, что по ходу программы необходимо передать один и то же аргумент нескольким функциям.

```

H=open("MyFile","rb");
read(buff,10,H);
seek(20,1,H);

```

По идее Си-компилятор должен был бы сгенерировать следующий код.

```

PUSH offset arb
PUSH offset aMyFile
CALL open
ADD SP,4

MOV [offset H],AX

PUSH [offset H]
PUSH [10]
PUSH buff
CALL read
ADD SP,6

PUSH [offset H]
PUSH 1
PUSH 20
CALL seek
ADD SP,6

```

Как нетрудно видеть один и тот же аргумент – дескриптор файла многократно заносится в стек, а потом выталкивается из него. Поэтому оптимизирующие компиляторы поступят, скорее всего, приблизительно так:

```

PUSH offset arb
PUSH offset aMyFile
CALL open
PUSH AX

PUSH [10]
PUSH buff
CALL read
ADD SP,4

PUSH 1
PUSH 20
CALL seek
ADD SP,10

```



Разобраться сколько аргументов принимает каждая функция одним лишь анализом балансировки стека абсолютно невозможно!

После выхода из первой функции стек остается несбалансированным. Вторая функция очищает только два аргумента, оставляя в стеке дескриптор файла для последующих функций.

И «отдуться» за все это приходится третьей функции, которая выталкивает из стека аж 5 слов! Но на самом деле размер его аргументов гораздо скромнее.

Можно, конечно, попытаться отслеживать аргументы, к которым функция обращается и, выбрав из них тот, что лежит «внизу» всех, вычислить на основе этого размер аргументов.

Однако такой подход предполагает, что функции известно число переданных ей аргументов, что в случае с Си - компиляторами неверно. Ведь перенос заботы об очистке стека с самой функции на вызывающий ее код объясняется как раз тем, что в Си функции **не знают**, сколько точно им параметров было передано.

Поэтому можно только удивляться, что даже на оптимизированном коде IDA сравнительно редко ошибается.

Операнд	Пояснения	
Ea	Линейный адрес, принадлежащий функции	
Return	Завершение	Пояснения
	!=0 !=BADADDR	Размер аргументов в байтах
	0	Функция не имеет кадра стека
	BADADDR	Ошибка

### long GetFrameSize(long ea);

Возвращает полный размер стекового фрейма в байтах. Он вычисляется по следующей формуле:

$$\text{FrameSize} == \text{FrameLvarSize} + \text{FrameArgsSize} + \text{FrameRegsSize} + \text{ReturnAddressSize}$$

То есть сумме размеров локальных переменных, аргументов, сохраненных в стеке регистров и адреса возврата всех вместе взятых.

Подробнее о каждой из них можно прочитать в описании функций GetFrameLvarSize, GetFrameArgsSize, GetFrameRegsSize.

Специальной функции, возвращающий значение размера адреса возврата не существует, однако, он может быть вычислен по следующей формуле:

$$\text{ReturnAddressSize} == \text{FrameSize} - \text{FrameLvarSize} + \text{FrameArgsSize} + \text{FrameRegsSize}$$

Поскольку в стековый фрейм входит и адрес возврата, то независимо от того, имеет ли функция локальные переменные или нет, он не может быть равен нулю.

Примеры использования:

```
seg000:0000 start          proc near
seg000:0000                call    sub_0_A
seg000:0003                call    sub_0_10
seg000:0006                call    sub_0_16
seg000:0009                retn
seg000:0009 start          endp
```

```
Message("0x%X \n",
GetFrameSize(0x10000)
);
```

2

```
seg000:0010 sub_0_10       proc near
seg000:0010                push   bp
seg000:0011                push   ax
seg000:0012                mov    bp, sp
seg000:0014                pop    bp
seg000:0015                retn
seg000:0015 sub_0_10       endp ; sp = -2
```

```
Message("0x%X \n",
GetFrameSize(0x10010)
);
```

6

```
Message("0x%X \n",
GetFrameRegsSize(0x10010)
);
```

4

Как видно, в последнем случае стековый фрейм состоял из адреса возврата и сохраненных в стеке регистров. Однако, если команды расположить по другому, то результат изменится:

```
seg000:000A sub_0_A       proc near
seg000:000A                push   bp
seg000:000B                mov    bp, sp
seg000:000D                push   ax
seg000:000E                pop    bp
seg000:000F                retn
seg000:000F sub_0_A       endp ; sp = -2
```

```
Message("0x%X \n",
GetFrameSize(0x1000A)
);
```

4

```
Message("0x%X \n",
GetFrameRegsSize(0x1000A)
);
```

2

Все команды, лежащие «ниже» (то есть в более старших адресах) относительно команды `mov bp, sp` (которая и определяет стековый фрейм) в него не попадают и можно безболезненно заносить (выталкивать) команды из стека, не боясь разрушить стековый фрейм.

Операнд	Пояснения	
Ea	Линейный адрес, принадлежащий функции	
Return	Завершение	Пояснения
	!=BADADDR	Размер стекового фрейма в байтах
	BADADDR	Ошибка

### **long MakeFrame(long ea,long lvsz,long frregs,long argsize);**

Создает фрейм стека функции или модифицирует уже существующий. Для этого достаточно указать любой адрес, принадлежащий функции и размеры области для локальных переменных, сохраненных регистров и аргументов.

Они расположены во фрейме в следующей последовательности.

Стековый фрейм
Локальные переменные
Сохраненные регистры
Аргументы, переданные функции
Адрес возврата из функции

При успешном завершении функция возвращает ID структуры, обеспечивающий доступ ко всем вышеперечисленным элементам. В противном случае функция вернет ошибку BADADDR.

Операнд	Пояснения	
Ea	Любой линейный адрес, принадлежащий функции	
lvsz	Размер локальных переменных в стековом фрейме	
frregs	Размер сохраненных регистров в стековом фрейме	
argsz	Размер аргументов, передаваемых функции	
Return	Завершение	Пояснения
	!=BADADDR	ID структуры, обеспечивающей доступ ко всем элементам стекового фрейма
	BADADDR	Ошибка

Модифицирование уже существующего фрейма стека может повлечь за собой удаление локальных переменных.

Например:

```
.text:00401487 __setargv      proc near
```

```

.text:00401487
.text:00401487 var_8      = dword ptr -8
.text:00401487 var_4      = dword ptr -4
.text:00401487
.text:00401487          push    ebp
.text:00401488          mov     ebp, esp
.text:0040148A          push    ecx
.text:0040148B          push    ecx

```

MakeFrame(0x401487,0,0,0);

```

.text:00401487 __setargv  proc near
.text:00401487          push    ebp
.text:00401488          mov     ebp, esp
.text:0040148A          push    ecx
.text:0040148B          push    ecx
.text:0040148C          push    ebx

```

Аргументы же функции никогда не удаляются из стекового фрейма, даже при уменьшении размера выделенного для них региона до нуля. Однако, это нарушает целостность всего фрейма и локальных переменных, лежащих «выше»

```

.text:00401520 sub_0_401520  proc near
.text:00401520
.text:00401520
.text:00401520 arg_0        = dword ptr 8
.text:00401520 arg_4        = dword ptr 0Ch
.text:00401520 arg_8        = dword ptr 10h
.text:00401520 arg_C        = dword ptr 14h
.text:00401520 arg_10       = dword ptr 18h
.text:00401520
.text:00401520          push    ebp
.text:00401521          mov     ebp, esp
.text:00401523          mov     ecx, [ebp+arg_10]
.text:00401526          mov     eax, [ebp+arg_C]
.text:00401529          push    ebx

```

MakeFrame(0x401520,0,0,0);

```

.text:00401520 sub_0_401520  proc near
.text:00401520
.text:00401520
.text:00401520 arg_0        = dword ptr 8
.text:00401520 arg_4        = dword ptr 0Ch
.text:00401520 arg_8        = dword ptr 10h
.text:00401520 arg_C        = dword ptr 14h
.text:00401520 arg_10       = dword ptr 18h
.text:00401520
.text:00401520          push    ebp
.text:00401521          mov     ebp, esp
.text:00401523          mov     ecx, [ebp+arg_10]
.text:00401526          mov     eax, [ebp+arg_C]
.text:00401529          push    ebx

```

## long GetSpd(long ea);

Возвращает значение регистра SP (ESP) в произвольной точке функции относительно его оригинального значения.

IDA использует достаточно простой алгоритм, отслеживающий только основные команды модифицирующие стековый регистр.

Для специфичных случаев предусмотрена ручная коррекция (смотри описание функции SetSpDiff) но в большинстве случаев IDA и сама справляется с этой задачей.

Пример использования:

```
.text:004010FF __amsg_exit      proc near
.text:004010FF
.text:004010FF
.text:004010FF arg_0          = dword ptr  4
.text:004010FF
.text:004010FF                cmp     dword_0_408758, 2
.text:00401106                jz     short loc_0_40110D
.text:00401108                call   __FF_MSGBANNER
.text:0040110D
.text:0040110D loc_0_40110D:
.text:0040110D                push  [esp+arg_0]
.text:00401111                call  __NMSG_WRITE
.text:00401116                push  0FFh
.text:0040111B                call  off_0_408050
.text:00401121                pop   ecx
.text:00401122                pop   ecx
.text:00401123                retn
.text:00401123 __amsg_exit      endp
.text:00401123
```

```
Message("%d \n",
GetSpd(0x4010FF)
);
```

0

```
Message("%d \n",
GetSpd(0x401111)
);
```

-4

```
Message("%d \n",
GetSpd(0x401116)
);
```

-8

```
Message("%d \n",
GetSpd(0x401122)
);
```

-4

```
Message("%d \n",
```

### GetSpd (0x401123)

```
);  
  
0
```

В точке входа в функцию значение SP (ESP) всегда равно нулю. Затем, в нашем примере, оно изменяется командой push, заносащей в стек двойное слово.

Обратите внимание, что значение ESP изменяется только после завершения команды – то есть с адреса начала следующей.

Относительное значение ESP	Адрес	Инструкция
0	.text:0040110D	push [esp+arg_0]
-4	.text:00401111	call __NMSG_WRITE

В точке выхода из функции значение SP (ESP) так же должно равняться нулю. В противном случае стек был бы несбалансированным, и команда возврата вытолкнула из стека не адрес возврата, а что-то совсем иное.

В таком случае вероятнее всего, что IDA не смогла отследить все инструкции, модифицирующие значения стекового регистра (или сделала это неправильно). Рекомендуется обнаружить это место и скорректировать его вызовом SetSpDiff.

Операнд	Пояснения
Ea	Линейный адрес в теле функции
Return	Относительное значение стекового регистра SP (ESP)

### long GetSpDiff(long ea);

Возвращает относительное изменение стекового регистра SP (ESP) командой, расположенной по линейному адресу 'ea'.

Например:

```
.text:004010FF __amsg_exit    proc near  
.text:004010FF  
.text:004010FF  
.text:004010FF arg_0      = dword ptr 4  
.text:004010FF  
.text:004010FF          cmp     dword_0_408758, 2  
.text:00401106          jz     short loc_0_40110D  
.text:00401108          call   __FF_MSGBANNER  
.text:0040110D  
.text:0040110D loc_0_40110D:  
.text:0040110D          push  [esp+arg_0]  
.text:00401111          call  __NMSG_WRITE  
.text:00401116          push  0FFh  
.text:0040111B          call  off_0_408050  
.text:00401121          pop   ecx  
.text:00401122          pop   ecx  
.text:00401123          retn  
.text:00401123 __amsg_exit    endp
```

```
Message("%d \n",
```

```

GetSpd (0x4010FF)
);

0

Message ("%d \n",
GetSpd (0x401111)
);

-4

Message ("%d \n",
GetSpd (0x401116)
);

-8

Message ("%d \n",
GetSpd (0x401122)
);

-4

Message ("%d \n",
GetSpd (0x401123)
);

0

```

Относительное значение ESP	Адрес	Инструкция
0	.text:0040110D	push [esp+arg_0]
-4	.text:00401111	call __NMSG_WRITE

Как и в случае с GetSpd необходимо задавать адрес начала следующей команды или точнее, конца текущей.

Операнд	Пояснения
Ea	Линейный адрес конца команды в теле функции
Return	Относительное изменение стекового регистра SP (ESP)

### success SetSpDiff(long ea,long delta);

Задаёт изменение стекового регистра SP (ESP) командой, лежащей по указанному линейному адресу. Дело в том, что IDA использует достаточно простой алгоритм, отслеживания SP (ESP), который не учитывает ряда особенностей некоторых экзотических команд.

Однако, в настоящее время этот механизм настолько усовершенствован, что практически невозможно придумать в каком случае команда SetSpDiff могла бы оказаться полезной.

Возьмем следующий, достаточно надуманный пример:

```

seg000:0000 000          public start
seg000:0000          start  proc near
seg000:0000          push   ax
seg000:0001 002          push   ax
seg000:0002 004          push   bp
seg000:0003 006          mov    bp, sp
seg000:0005 006          mov    word ptr [bp+2], 2
seg000:000A 006          pop    bp
seg000:000B 004          mov    bp, sp
seg000:000D 004          mov    cx, [bp+0]
seg000:0010 004          pop    ax
seg000:0011 002          add    sp, cx
seg000:0013 002          push   ax
seg000:0014 004          add    sp, cx
seg000:0016 004          retn
seg000:0016          start  endp ; sp = -4

```

```

Message("%d \n",
GetSpDiff(0x10013)
);

```

0

```

Message("%d \n",
GetSpDiff(0x10016)
);

```

0

Что бы узнать значение SP после завершения команды `add sp, cx` IDA, очевидно, должна знать чему равен регистр CX. Что бы его отследить пришлось бы включить в дизассемблер полноценный эмулятор 0x86 процессора. Пока это еще не реализовано и IDA предполагает, что значение CX равно нулю и, таким образом, уже неправильно определяет значение SP во всех нижележащих точках функции.

Исправить положение можно ручной коррекцией значения SP. Функция `SetSpDiff` задает изменение регистра SP после выполнения команды. Для этого необходимо передать линейный адрес **конца**, а не начала команды.

В нашем случае необходимо скорректировать величину изменения SP командами `ADD SP, CX` расположенными по адресам `seg000:0011` и `seg000:0014`. Линейные адреса команд соответственно равны `seg000:0013` и `seg000:0016`. Их и необходимо передать функции вместе с действительной величиной изменения SP.

```

SetSpDiff(0x10013, 2);
SetSpDiff(0x10016, 2);

```

```

seg000:0000 000          public start
seg000:0000          start  proc near
seg000:0000          push   ax
seg000:0001 002          push   ax
seg000:0002 004          push   bp

```



```

seg000:0003 006          mov     bp, sp
seg000:0005 006          mov     word ptr [bp+2], 2
seg000:000A 006          pop     bp
seg000:000B 004          mov     bp, sp
seg000:000D 004          mov     cx, [bp+0]
seg000:0010 004          pop     ax
seg000:0011 002          add     sp, cx
seg000:0013 000          push   ax
seg000:0014 002          add     sp, cx
seg000:0016 000          retn
seg000:0016          start      endp

```

Операнд	Пояснения	
Ea	Линейный адрес конца команды	
delta	Величина изменения SP указанной командой	
Return	Завершение	Пояснения
	1	Успешно
	0	Ошибка

success MakeLocal(long start,long end,char location,char name)

версия 3.74 и старше

С версии 3.74 IDA поддерживает локальные переменные, которые в большинстве же случаев распознает и создает автоматически. Но иногда она не способна правильно их распознать, и тогда эта миссия ложиться на плечи пользователя. Подробнее о локальных переменных можно прочитать в специальной главе «Локальные переменные» посвященной непосредственно им.

'MakeLocal' полный аналог («~Edit\Functions\Stack variables»). В прототипе функции 'MakeLocal' указывается область видимости локальной переменной ('start' и 'end'), однако существующие версии IDA (вплоть до IDA 4.0) не поддерживает такой возможности и область видимости локальной переменной распространяется целиком на всю функцию.

Функция принимает следующие операнды:

операнд	Пояснения
end	Этот операнд игнорируется. Обычно его оставляют равным нулю, но из соображений совместимости с последующими версиями рекомендуются задавать конец функции или константу 'BADADDR' - тогда область локальной переменной будет определена IDA автоматически.
start	Этот операнд в существующих версиях должен совпадать с началом функции, иначе MakeLocal возвратит ошибку (в последующих версиях start должен определять адрес начала видимости локальной переменной)
location	Смещение переменной в кадре стека, задаваемое в виде строкового выражения "[BP+XX]", где "xx" представлено в шестнадцатеричном исчислении. Спецификатор 'x' можно ставить, а можно не ставить - все равно значение будет трактоваться как шестнадцатеричное. Интересной недокументированной особенностью является

	возможность задавать другие регистры, помимо BP, например 'AX', однако это не возымеет никакого значения, все равно будет трактоваться как 'BP'	
name	Это есть суть имя создаваемой переменной со всеми ограничениями, наложенными на имена и метки. Признаком хорошего тона является выбор такой нотации, что бы локальные переменные легко визуально отличались от остальных. IDA всем автоматически всем создаваемым локальным переменным присваивает имя 'var_xx'.	
Return	==return	пояснения
	==1	Локальная переменная успешно создана
	==0	Ошибка

Hot Key	Menu
<Ctrl-K>	Edit\Functions\Stack variables

Кроме локальных переменных этой же функцией можно создавать и размещенные в стеке аргументы, т.к. фактически это те же локальные переменные, только размещенные по другую сторону кадра стека (с положительным смещением, а не отрицательным).

IDA в большинстве случаев самостоятельно автоматически определяет аргументы функций (называя их 'arg\_xx') и вмешательство пользователя обычно не требуется.

Пример:

```
MakeLocal (ScreenEA(), 0, "[bp+0x4]", "MyVar");

.text:00401124 sub_0_401124    proc near
.text:00401124
.text:00401124 MyVar      = dword ptr 4
.text:00401124
.text:00401124                push    [esp+MyVar]
```

### success SetReg (long ea,char reg,long value);

Функция устанавливает значение сегментных регистров. IDA автоматически отслеживает их значение (и изменение) и хранит его для каждой точки кода.

Этот механизм достаточно совершенен и обычно вмешательства не требуется, однако в некоторых случаях IDA неправильно вычисляет значение сегментных регистров, например, если модификацией управляет отдельный поток и тогда требуется вмешательство пользователя.

SetReg генерирует директиву ассемблера ASSUME, помещаемую в исходный код. При этом регистр должен указывать на начало сегмента.

Все существующие ассемблеры поддерживают именно такой режим, но программистам иногда требуется установить сегментный регистр по произвольному адресу внутри сегмента (например, для организации плавающего кадра окна для преодоления 64 КБ барьера реального режима на сегмент) SetReg нормально принимает такие значения.

Например:

```
dseg:0000 start                proc near
dseg:0000                    mov     ax, seg dseg
dseg:0003                    mov     ds, ax
dseg:0005                    assume ds:dseg
dseg:0005                    mov     dx, offset aHelloSailor ;
dseg:0008                    call    WriteLn
```

```

dseg:000B      mov     ax, ds
dseg:000D      inc     ax
dseg:000E      mov     ds, ax
dseg:0010      assume ds:nothing
dseg:0010      mov     dx, 2Fh ; '/'
dseg:0013      call    WriteLn
dseg:0016      mov     ah, 4Ch
dseg:0018      int     21h
dseg:0018      start  endp

dseg:0020      aHelloSailor  db 'Hello,Sailor',0Dh,0Ah,'$'
dseg:002F      db '$$$$$$$$$$$$$$$$'
dseg:003F      aHelloIda    db 'Hello,IDA!',0Dh,0Ah,'$'
dseg:003F      dseg          ends
dseg:003F
dseg:003F
dseg:003F      end start

```

Смещение 0x2F в строке dseg:0x10 на самом деле указывает на строку dseg:0x3F, т.к. перед этим значение DS было увеличено на единицу (один параграф равен шестнадцати байтам) Как «объяснить» это дизассемблеру?

Переведем курсор на строку 'dseg:0x10' и используем следующую команду:

```
SetReg (ScreenEA (), "DS", 0x1001);
```

Теперь если преобразовать операнд в смещение получится следующее:

```

dseg:0010      loc_0_10:                                     ; DATA XREF: start+10o
dseg:0010      mov     dx, offset aHelloIda - offset loc_0_10 ; "Hello,IDA!\r\n$"

```

Заметим по комментарию, что теперь IDA правильно определила ссылку. Однако, сгенерировала *неверный* код.

«offset aHelloIda - offset loc\_0\_10» будет работать только до тех пор, пока метка loc\_0\_10 будет расположена по смещению 0x10, и нам необходимо заменить ее константой 0x10. Для этого воспользуемся, например, функций OpAlt.

SetReg изменяет значение сегментного регистра в каждой точке до следующего 'ASSUME' или конца сегмента.

Операнд	пояснения	
'ea'	линейный адрес	
'reg'	символьное название регистра. ("CS", "DS", "ES" и т.д.)	
'value'	значение регистра в параграфах	
Return	==return	пояснения
	==1	Операция была выполнена успешно
	==0	Ошибка

Функция SetReg эквивалентна команде меню «~EDIT\Segments\Change segment register value».

### long GetReg (long ea,char reg);

Возвращает значение сегментных регистров в произвольной точке программы. Подробнее об этом можно прочитать в описании функции SetReg.

операнд	Пояснения	
'ea'	линейный адрес, в котором необходимо определить значение регистра	
'reg'	символьное имя регистра. Например "DS", "GS" и так далее	
Return	==return	Пояснения
	!=0xFFFF	Значение сегментного регистра в параграфах
	==0xFFFF	Ошибка

Функция возвращает 16-битное целое, содержащие значение сегментного регистра в параграфах. В 32-битных программах функция обычно возвращает не непосредственное значение, а селектор.

Для получения искомого адреса необходимо воспользоваться функцией AskSelector. Поскольку селекторы «визуально» неотличимы от адресов сегментов, то для уверенности необходимо вызывать AskSelector всякий раз для проверки на принадлежность возвращаемого значения к селекторам. Если селектор с указанным номером не существует, то это непосредственное значение.

Если регистр не существует (например "MS") или не определен, то функция и в том и другом случае вернет ошибку 0xFFFF, а не BADADDR, как утверждает прилагаемая к IDA документация.

Пример использования:

```

seg000:0000 seg000          segment byte public 'CODE' use16
seg000:0000                assume cs:seg000

Message ("%x \n",
GetReg (0x1000, "CS")
);

1000

.text:00401000 _text        segment para public 'CODE' use32
.text:00401000                assume cs:_text

Message ("%x \n",
GetReg (ScreenEA (), "CS")
);

1

Message ("%x \n",
AskSelector (1)
);

0

```

## ПЕРЕКРЕСТНЫЕ ССЫЛКИ

## ЧТО ТАКОЕ ПЕРЕКРЕСТНЫЕ ССЫЛКИ?

Долгое время SOURCER лидировал среди других дизассемблеров в умении находить и восстанавливать перекрестные ссылки. На этом, правда, его основные достоинства и оканчивались, но все равно он активно использовался для исследования программного обеспечения.

Что же такое перекрестные ссылки и почему они так важны для популярности дизассемблера? Покажем это на следующем примере. Рассмотрим простейший случай. Допустим, исходный файл выглядел так:

```
.MODEL TINY
.CODE
ORG 100h
Start:
    MOV     AH, 9
    LEA    DX, s0
    INT    21h
    RET
s0      DB "Hello, Sailor!", 0Dh, 0Ah, '$'
END Start
```

После ассемблирования он будет выглядеть следующим образом:

```
seg000:0100 start          proc near
seg000:0100                mov     ah, 9
seg000:0102                mov     dx, offset aHelloSailor ;
"Hello, Sailor!\r\n$"
seg000:0105                int     21h
seg000:0105
seg000:0107                retn
seg000:0107 start          endp
seg000:0107
seg000:0107 ; -----
-----
seg000:0108 aHelloSailor   db 'Hello, Sailor!', 0Dh, 0Ah, '$'
seg000:0108 seg000        ends
```

Допустим, мы хотим узнать, какой код выводит эту строку на экран. Когда-то для этого приходилось кропотливо изучать весь листинг, и то не было шансов, что с первого раза выводящий строку код удастся рассмотреть.

Поэтому, эту задачу возложили на плечи дизассемблера. Так, что бы машина сама анализировала выполнение программы и восстанавливала все связи. Это невероятно упростило анализ программ, как впрочем, и взлом защит.

Стало достаточно только найти в строку, которая защита выводит в случае неудачного завершения проверки (ключевой дискеты ли, или пароля – совершенно не важно), как дизассемблер поможет мгновенно найти, вводящий ее код, а значит, и локализовать защиту каким бы длинной программа не оказалась.

Как правило, где-то неподалеку расположен условный переход, передающий управление этой ветви. Стоит только изменить его на противоположный, как защиту можно считать взломанной.

Но ведь же не для хакерства же были придуманы перекрестные ссылки! Разумеется, нет! Помощь хакерам это только побочный эффект (хотя и очень приятный для них). Значительно важнее поддержка перекрестных ссылок чтобы правильно дизассемблировать код!

Покажем это на следующем примере:

```

.MODEL TINY
.CODE
ORG 100h
Start:
    LEA    AX,s0
    PUSH  AX
    CALL  Print
    RET
s0      DB  'Hello, Sailor!',0Dh,0Ah,'$'
Print:
    POP   AX
    POP   DX
    PUSH  AX
    MOV   AH,9
    INT   21h
    RET
END Start

```

Первые четыре строки любой дизассемблер разберет без труда. Но вот дальше начнутся сложности. Как узнать, что следом идет текстовая строка, а не исполняемый код? Если же дизассемблировать как код из соображений одной лишь надежды на это, то полученный результат станет похожим на бред и вся программа окажется дизассемблированной неправильно.

Поэтому приходится эмулировать ее исполнение и отслеживать все косвенные и непосредственные переходы. Если ассемблер сумеет распознать вызов подпрограммы, то ссылку на строку он восстановит с куда большей легкостью.

Однако, тут скрывается один подводный камень. Эмуляция (даже частичная) требует больших накладных расходов, и если каждый раз ее выполнять «налету», то никаких вычислительных ресурсов не хватит для нормальной работы!

Поэтому приходится прибегать к компиляции. Да, именно к компиляции. Ведь компиляция это только перевод с одного языка в другой, а не обязательно в машинные коды. В данном случае, как раз запутанный язык ассемблера (а точнее одних лишь ссылок) преобразуется к максимально производительной и компактной форме записи. Или другими словами можно сказать, что перекрестные ссылки – это сохраненный результат работы эмулятора.

Кроме того, если выполнение программы однонаправлено, то есть часто невозможно сказать, выполнение какой инструкции предшествовало текущей, то перекрестные ссылки предоставляют такую возможность!

Можно начать изучение программы с любой точки, свободно двигаясь по ходу ее исполнения как назад, так и вперед. Это, в самом деле, очень удобно. Ведь в большинстве случаев не требуется изучить всю программу целиком, а только один из ее компонентов. Например, тот, что отвечает за взаимодействие с интересующим вас форматом файла. Предположим, что в заголовке находится некая сигнатура, которая проверятся исследуемой программой и находится в дизассемблируемом листинге в «прямом виде».

Тогда можно по перекрестным ссылкам, перейти на процедуру загрузки файла и начать ее изучение. И это не единственный пример. Перекрестные ссылки активно используются при дизассемблировании программ любой сложности и относятся к незаменимым компонентам дизассемблера.

Однако, как нетрудно догадаться, что гарантировано можно отслеживать только непосредственные ссылки, такие как CALL 0x666, а уже MOV DX,0x777 может быть как смещением, так и константой, а про инструкции типа CALL BX говорить и вовсе не приходится – для вычисления значения регистра BX потребуется весьма не хилый эмулятор процессора.

Поэтому не удивительно, что большинство ассемблеров отслеживало перекрестные ссылки из рук вон плохо. Даже первые версии IDA не были совершенны в этом плане.

То есть поддержка перекрестных ссылок имела, но их созданием приходилось заниматься человеку (ведь IDA изначально планировалась как интерактивная среда!), а не машине.

Но с течением времени ситуация изменялась и интеллектуальные механизмы IDA улучшались. С версии 3.7 она уже значительно превосходила в этом отношении все остальные существующие в мире ассемблеры, включая SOURCER, и продолжала совершенствоваться!

## ALMA MATER

Из предыдущей главы можно сделать вывод, что реально поддержка того, что подразумевается под термином «перекрестные ссылки» состоит как минимум из механизма отслеживания перекрестных ссылок и механизма работы с перекрестными ссылками, сохраненными в некотором внутреннем формате дизассемблера.

Устройство и способности интеллектуального анализатора IDA тема для отдельного разговора, здесь же будет говориться исключительно о работе с уже созданными перекрестными ссылками.

С первого взгляда возможно будет даже не понятно, о чем идет речь. Если перекрестная ссылка уже создана, то какие могут быть проблемы? На самом деле все не так просто. Перекрестная ссылка может быть создана (и при том не в единственном числе), а может быть, и нет. Как узнать это наверняка? И как получить адрес, на который перекрестная ссылка ссылается?

Вот для этого и предусмотрено почти два десятка функций, поддерживающих работоспособность IDA. Все они ниже будут подробно рассмотрены, но сначала рассмотрим механизмы взаимодействия с перекрестными ссылками, не углубляясь в детали.

Итак, любая перекрестная ссылка состоит из **источника** и **приемника**. В контекстной помощи IDA источник обозначается как 'from', а приемник – как 'to'.

Источником называется операнд, ссылающийся на приемник, но не наоборот! Разберем в свете этого приведенный выше пример:

```
1. .MODEL TINY
2. .CODE
3. ORG 100h
4. Start:
5. MOV     AH, 9                ;
6. LEA    DX, s0                ; =>
7. INT    21h                   ;
8. RET                                ;
9. s0     DB "Hello, Sailor!", 0Dh, 0Ah, '$' ; <=
10. END Start
```

Так в строке 8 должна быть создан источник перекрестной ссылки, а в строке 11 – приемник.

Часто вызывает путаницу, что IDA создает комментарий к перекрестной ссылке только возле приемника. Это, разумеется, правильно, потому что источник в комментариях не нуждается – в большинстве случаев очевидно на что ссылается операнд (хотя в случае инструкций подобных CALL BX этого сказать нельзя), а вот по виду приемника источник установить невозможно.

Вот IDA и отображает его в виде комментария:

```

seg000:0100      public start
seg000:0100 start proc near
seg000:0100      mov     ah, 9
seg000:0102      mov     dx, offset aHelloSailor
seg000:0105      int     21h
seg000:0105
seg000:0107                      retn
seg000:0107 start                      endp
seg000:0107
seg000:0107 ; -----
seg000:0108 aHelloSailor      db 'Hello, Sailor!',0Dh,0Ah,'$' ; DATA
XREF: start+2o
seg000:0108 seg000                      ends

```

Обратите внимание, что IDA ничем не выделила строку 0x102 – создается иллюзия, что здесь ничего нет. Но на самом деле именно с этим адресом и связана перекрестная ссылка, а точнее, ее источник.

Практически для всех манипуляций с перекрестными ссылками нужно знать пару значений – линейные адреса источника и приемника. Впрочем, есть и такие функции, что возвращают источник (источники) для указанного адреса приемника. Но об них поговорим позднее, а пока остановимся на том факте, что одновременно по одному и тому же адресу может располагаться несколько как приемников, так и источников.

Начнем в первую очередь, как более простого для понимания.

```

seg000:0100      org 100h
seg000:0100      assume es:nothing, ss:nothing, ds:seg000, fs:nothing, gs:nothing
seg000:0100
seg000:0100      public start
seg000:0100 start:                      ; "Hello, World!\r\n$"
seg000:0100      push   offset aHelloWorld
seg000:0103      call   Print
seg000:0106      push   offset aHelloSailor ; "Hello, Sailor!\r\n$"
seg000:0109      call   Print
seg000:010C      retn
seg000:010D
seg000:010D; _____ S U B R O U T I N E _____
seg000:010D
seg000:010D Print      proc near                      ; CODE XREF: seg000:0103p
seg000:010D                      ; seg000:0109p
seg000:010D      pop     ax
seg000:010E      pop     dx
seg000:010F      push   ax
seg000:0110      mov     ah, 9
seg000:0112      int     21h                      ; DOS - PRINT STRING
seg000:0112                      ; DS:DX -> string terminated by "$"
seg000:0114      retn
seg000:0114 Print      endp ; sp = 2
seg000:0114
seg000:0114 ; -----
seg000:0115 aHelloWorld      db 'Hello, World!',0Dh,0Ah,'$' ; DATA XREF: seg000:0100o
seg000:0125 aHelloSailor      db 'Hello, Sailor!',0Dh,0Ah,'$' ; DATA XREF: seg000:0106o
seg000:0125 seg000                      ends
-0001010D: sub_0_10D

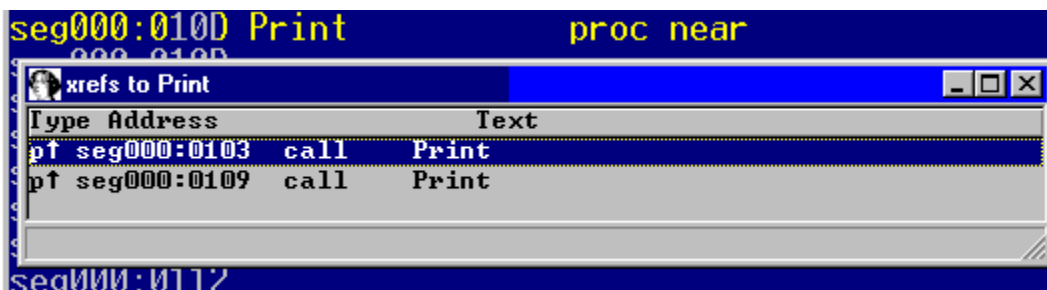
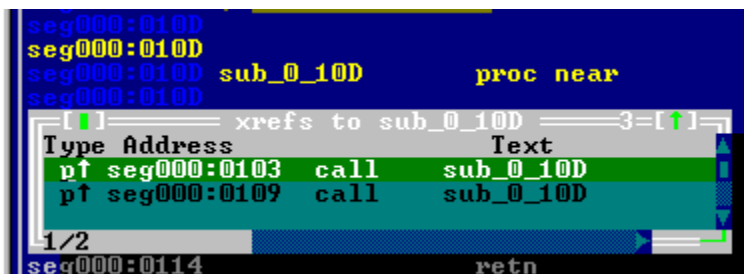
```

В этом примере процедура Print вызывалась из двух точек кода, о чем свидетельствуют две перекрестные ссылки, проставленные IDA как комментарии. Следовательно, один приемник может иметь и более одного источника.

Что бы изучить вызывающий эту процедуру код достаточно только подвести курсор в границы адреса, указанного в перекрестной ссылке и нажать Enter и при желании возвратиться назад по <Esc>.



Так же можно переместить курсор в любое место строки 0x10D и выбрать пункт меню ~ View \ Cross references. Появится окно следующего вида:



Поскольку довольно часто встречается, что на один приемник ссылаются десятки (а то и больше!) различных источников, то IDA считает не рациональным отображать их в виде комментариев и показывает по умолчанию лишь две первые из них (перекрестные ссылки отсортированы по линейным адресам источников – от младших адресов, к старшим), то просматривать остальные приходится именно в таком окне.

Как получить программно линейный адрес источника будет рассказано несколько позже, поскольку это делает по разному в зависимости от типов перекрестных ссылок.

Сейчас же рассмотрим ту ситуацию, когда один источник имеет более одного приемника. С первого взгляда это абсурдно, но выйти за рамки непосредственных операндов, то можно вообразить себе следующую ситуацию:

```

seg000:0002      mov     ds, ax
seg000:0004      assume ds:seg000
seg000:0004      mov     ah, 6
seg000:0006      mov     di, offset off_0_25
seg000:0009      jmp     short Print
seg000:0009      ; -----
seg000:000C      Def_1:                ; CODE XREF: start+1Bu
seg000:000C                        ; DATA XREF: seg000:0025o
seg000:000C      mov     dl, 31h ; '1'
seg000:000E      retn
seg000:000F      ; -----
seg000:000F      Def_2:                ; CODE XREF: start+1Bu
seg000:000F                        ; DATA XREF: seg000:0027o
seg000:000F      mov     dl, 32h ; '2'
seg000:0011      retn
seg000:0012      ; -----
seg000:0012      Print:                ; CODE XREF: start+9j

```

```

seg000:0012                                ; start+1Fj
seg000:0012    mov     bx, [di]
seg000:0014    add     di, 2
seg000:0017    or      bx, bx
seg000:0019    jz      loc_0_21
seg000:001B    call   bx
seg000:001D    int     21h
seg000:001F    jmp     short Print
seg000:0021 ; -----
seg000:0021
seg000:0021 loc_0_21:                                ; CODE XREF: start+19j
seg000:0021    mov     ah, 4Ch
seg000:0023    int     21h
seg000:0023 start                          ; AL = exit code
seg000:0023    endp
seg000:0023 ; -----
seg000:0025 off_0_25    dw offset Def_1                ; DATA XREF: start+60
seg000:0027    dw offset Def_2
seg000:0029    dw offset def_3
seg000:002B    dw offset def_4
seg000:002D    dw 0
seg000:002F ; -----
seg000:002F
seg000:002F def_3:                                ; CODE XREF: start+1Bu
seg000:002F    ; DATA XREF: seg000:0029o
seg000:002F    mov     dl, '3'
seg000:0031    retn
seg000:0032 def_4:
seg000:0032
seg000:0032    mov     dl, '4'
seg000:0034    retn

```

Подобные примеры не редкость и встречаться с ними приходится довольно таки часто. Обратим внимание на следующую строку:

```
seg000:001B    call   bx
```

Она осуществляет последовательную передачу управления функциям, читаемых в цикле из списка, а, следовательно, ссылается более чем на один адрес.

К сожалению пока IDA не умеет автоматически вычислять значение регистра BX и, следовательно, не может ни создать перекрестных ссылок, ни даже дизассемблировать вызываемые этой строкой функции. Скорее всего они будут помечены как 'unexplored'.

Поэтому эта часть работы ложится на плечи пользователя. Часто при этом дизассемблируют код, но забывают создать перекрестные ссылки. Что при этом получается? А то, что вернувшись к дизассемблируемому файлу спустя некоторое время, вы уже не будете помнить какой код вызывает эти функции и анализ придется начинать сначала.

Но что бы создавать перекрестные ссылки нужно быть осведомленным в их архитектуре, чему и посвящена следующая глава.

## АРХИТЕКТУРА ПЕРЕКРЕСТНЫХ ССЫЛОК

В предыдущей главе говорилось, что не зависимо от типа, перекрестная ссылка состоит из двух разных частей – **источника** и **приемника**. Каждый из них связан с определенным линейным адресом. Причем с любым адресом может быть связано одновременно как несколько приемников, так и несколько источников.

Другими словами с каждым линейным адресом может быть ассоциирован список источников (приемников). А, следовательно, нужно быть готовыми для работы со списками. Но для начала разберемся с тем, какие типы перекрестных ссылок существуют, ибо для работы с ними используются различные функции.

В первом приближении их всего два. Это ссылки на код и ссылки на данные. Ссылки на код встречаются всякий раз, когда какая-то инструкция нарушает нормальное выполнение кода программы и изменяет (возможно, лишь при некоторых обстоятельствах) значение регистра – указателя команд.

Говоря проще – такие ссылки образуют все команды условного и безусловного перехода и вызова подпрограмм, такие как JMP, CALL, JZ и тому подобные.

С перекрестными ссылками на данные мы сталкиваемся всякий раз, когда какая-то инструкция обращается к данным, по их смещению. Например, LEA, MOV xx, offset и так далее, в том числе и DW offset MyData.

Но есть еще и третий тип, который кардинально отличается от первых двух уже тем, что является внутренним типом перекрестных ссылок IDA и грубо говоря, пользователем знать о его существовании, а уж тем более вникать в технические детали реализации совсем необязательно.

Однако, это помогает лучше понять работу многих команд, поэтому ниже мы его рассмотрим.

Разумеется, речь идет о «**ссылке на следующую команду**» (**Ordinary flow** в терминологии IDA). Именно с помощью его IDA и отслеживает выполнение программы. Это перекрестная ссылка указывает на следующую команду при нормальном исполнении программы. Покажем это на следующем примере:

```

seg000:0012          mov     bx, [di]      ; =>
seg000:0014          add     di, 2         ;<=>
seg000:0017          or      bx, bx       ;<=>
seg000:0019          jz     loc_0_21      ;<=>
seg000:001B          call   bx           ;<=>
seg000:001D          int    21h         ;<=>
seg000:001F          jmp    short Print  ;<
seg000:0021 ; -----
seg000:0021
seg000:0021 loc_0_21:
seg000:0021          mov     ah, 4Ch      ; =>
seg000:0023          int    21h         ;<
seg000:0023 start    endp

```

Два цвета используются только лишь для обличения восприятия, – что бы было можно отличий пары приемник-источник друг от друга. Как видно, цепочка проходит сквозь все команды, пока не доходит до команды безусловного перехода. Тут она и обрывается.

Адрес перехода можно узнать по перекрестной ссылке типа «код», которая автоматически образуется здесь. Что это дает? Возможность трассировки программы, например, для определения адреса конца функции, который заканчивается, как правило RET, то есть так же инструкцией безусловной передачи управления без возврата в текущую последовательность команд, в отличие от CALL, которая после выполнения подпрограммы передает управление следующей за ней команде.

Словом, если все упростить, то нет никакого смысла выделять ссылку на следующую команду в отдельный тип. Она хорошо описывается одним лишь типом ссылки на код, поскольку текущая инструкция как бы вызывает следующую (ну во всяком случае

фактически происходит именно так – или другими словами – текущая инструкция передает, или может передавать, управление следующей). Вот в этих случаях и создается ссылка **Ordinary flow**

“Скрытой” она объявлена по двум причинам. Первая из них очевидна – какой смысл захламлять текст лишней, никому не нужной информацией? Впрочем, IDA все же выделяет перекрестные ссылки на следующую команду. Точнее выделяет их **отсутствующие**. Сплошная черта (в нашем примере в строке 0x21) как раз и говорит об том, что ссылка на следующую команду в данном месте отсутствует.

Вторая причина кроется в оптимизации. Если все остальные ссылки хранятся в bTree, которой обеспечивает не самый быстрый доступ к данным, то ссылки на следующую команду содержатся в флагах (смотри описании виртуальной памяти), что значительно ускоряет работу с ними. А поскольку IDA очень интенсивно использует их, то выигрыш в скорости весьма существенен.

Таким образом, **Ordinary flow** можно рассматривать как отдельный, самостоятельный тип ссылок, а с другой стороны, как частный случай разновидности ссылок на код.

Предоставленные в распоряжение пользователя функции большей частью скрывают эти различия, но и то часть команд приходится выполнять с огорками, о чем и сказано в их описании.

Как будет показано ниже, IDA поддерживает еще и «уточняющий» тип – скажем Jump, call или offset. Но на самом деле это всего лишь флаг, или атрибут перекрестной ссылки и играет только информационную роль, и никакого другого влияния не оказывает.

С другой стороны два типа перекрестных ссылок на код и данные можно рассматривать вместе, поскольку операции с ними производятся аналогично, не зависимо от типа, но разными наборами функций.

Ссылки на следующую инструкцию при этом лучше не модифицировать без особой на то нужды, обращаясь к ним только на чтение, хотя и запись так же доступна.

Итак, рассмотрим работу со списком, о котором мы говорили выше. Как было сказано ранее, вся архитектура IDA базируется на линейных адресах и связанных с ними объектах и элементах. Но если с каждым линейным адресом мог был связан только один комментарий каждого типа и только одно имя, то источников и приемников у каждого адреса может быть сколько угодно. Причем один и тот же адрес может быть одновременно как источником одной перекрестной ссылки, так и приемником другой.

Например:

```
seg000:000C          jnb     loc_0_17
seg000:000E          mov     ah, 3Ch ; '<'
seg000:0010          xor     cx, cx
seg000:0012          mov     dx, 206h
seg000:0015          int     21h
seg000:0015
seg000:0015
seg000:0017
seg000:0017 loc_0_17          ; CODE XREF: seg00
seg000:0017          mov     ds:word_0_1DA, ax
seg000:01DA*word_0_1DA  dw 0          ; DATA XREF: seg000:0017w
```

Поэтому необходимо говорить о двух независимых списках – приемников и источников, да еще отдельных для каждого типа ссылок – для кода и для данных.

Возникает вопрос, – а куда входят ссылки на следующую инструкцию? Ответ – они вообще не входят в упомянутый выше список, так как физически хранятся отдельно. Но некоторые функции IDA эмулируют их присутствие в списке ссылок типа «код», однако, это приводит часто к путанице и запутывает понимание пользователя. Поэтому будем все же

считать, что инструкции на следующую команду как бы сами по себе. Это значительно упрощает понимание.

Таким образом, перейдем к рассмотрению организации этого списка и работы с ним (поскольку не зависимо от хранимых данных, – работа со всеми списками идентична). Но при ближайшем рассмотрении (и залезании с позволения так сказать интимную область IDA) никакого списка нет, а есть только двоичное дерево, в узлах которого и хранятся перекрытые ссылки в виде (from, to). При необходимости IDA просматривает дерево и извлекает все элементы, адреса которых совпадают с запрошенным.

Но, увы, доступа к Btree IDA не предоставляет, но дает функции, работающие с перекрестными ссылками исключительно, как с односвязным списком. То есть это Rfirst (получение первого элемента) и Rnext (получение всех последующих элементов).

При этом работа идет не с индексами (которых попросту нет), а исключительно со значениями элементов списка. Таким образом, Rnext принимает линейный адрес и, просматривая двоичное дерево, выдает следующую за ним перекрестную ссылку указанного типа или –1, когда список исчерпан.

Таким образом, Rnext(0) с первого взгляда равносильна Rfirst, которая становится попросту не нужна. На самом деле все немного запутаннее. И понять это автор смог только после того, как связался с разработчиком IDA и обратился к нему за разъяснениями.

На самом деле Rnext **никогда** не возвращает ссылку на следующую инструкцию. Они хранятся отдельно и поэтому выпадают из поля зрения Rnext. Но вот Rfirst действует иначе. Она проверяет – существует ли ссылка на следующую инструкцию и если да, то возвращает в первую очередь ее. В противном случае – первый элемент списка.

А теперь вообразим себе следующую ситуацию. Пусть у нас имеется следующий код:

```
seg000:0000                push    ax    ; CODE XREF:
seg000:2864p
seg000:0000                ; ← приемник

seg000:2864                call    bx    ; → источник

seg000:2869 loc_0_286                ; CODE XREF:
seg000:2864p
seg000:2869                inc     si    ; ← приемник

seg000:2892 loc_0_2892:                ; CODE XREF:
seg000:2864p
seg000:2892                ; ← приемник
seg000:2892                cmp     byte ptr [si], 22h ;
""
```

Что будет если попытаться просмотреть список приемников в строке 0x2864? По логике Rfirst должна вернуть адрес ссылки на следующую команду, то есть 0x2869. Если теперь передать его Rnext то, по логике она должна будет вернуть следующий на нем приемник, то есть 0x2892, а приемник по адресу 0x0 окажется «вне поля зрения». Так ли это? На самом деле нет! Rnext сперва проверяет – является ли переданный ей адрес ссылкой на следующую команду, и если да, то начинает просмотр с начала списка!

Однако, Функции xfirst0 ведут себя иначе и не выполняют этой дополнительной проверки, в остальном же они ничем другим не отличаются от своих собратьев.

Зачем знать все эти подробности? Да просто работать программно с перекрестными ссылками придется намного чаще, чем бы этого хотелось. Дело в том, что IDA может отображать список приемников, как в виде комментариев, так и в окне списка (подробности смотри выше в главе ALMA MATER), но вот **никак** не отображает источники, считая что «они и так очевидны».

Но это на самом деле не так, в случае с командами по типу “CALL BX” – и возникает естественная потребность посмотреть, – а куда же передается управление. Конечно, IDA не отслеживает значение регистра BX автоматически и не создает в этом месте перекрестные ссылки, но вот **человек** это сделать очень даже может.

А вот оставшуюся мелочь – перейти по требуемому адресу (или посмотреть их список) интерактивно решить, видимо, невозможно. Поэтому приходится прибегать к языку скриптов и самостоятельно просматривать список значений.

Кстати, для облегчения навигации по файлу его можно добавить в комментарий к инструкции. Это, по-видимому будет наилучшим решением.

Подробнее об архитектуре перекрестных ссылок рассказано в описании функций, которые приведены ниже.

## ХРАНИЕНИЕ ПЕРЕКРЕСТНЫХ ССЫЛОК

Как хранятся перекрестные ссылки внутри IDA скрыто от пользователя. Достоверно известно лишь то, что хранятся они **достаточно эффективно**. А детали реализации недокументированны и могут меняться от версии к версии.

Однако, значение того, как физически хранятся перекрестные ссылки помогает лучше понять их структуру и работу с ними. На самом деле перекрестная ссылка представляет собой **два** объекта.

Первый из них – источник, который «одним концом» ассоциирован с линейным адресом, по которому он расположен, а другим указывает на адрес приемника. Аналогично и с приемником. Он так же состоит из двух концов.

То есть перекрестная ссылка это «двупольный» объект и этим и объясняется идентичный набор функций для работы с ее источником и приемников, а во-вторых, скорость доступа к источнику и приемнику одинакова. Это было бы не так, если бы в узле дерева хранилась структура (from, to) и тогда бы для поиска каждого приемника пришлось бы просматривать все дерево

Это не относится к ссылкам на следующую команду, которые хранятся во флагах и организованы совсем по-другому. На каждую ссылку расходуется всего один бит(!). Если он установлен, то, следовательно, ссылка на следующую команду присутствует и наоборот.

Адрес же ссылки определяется длиной инструкции, которая численно совпадает с длиной объекта. Подробнее об этом можно прочитать в главе «Объекты в IDA»

Однако, необходимо еще раз уточнить, что все эти подробности могут и не соответствовать действительности в какой-то конкретной версии IDA. Кроме того, с течением времени алгоритмы могут быть пересмотрены и изменены на другие, более эффективные.

Возможно, даже, что изменения затронут и встроенный язык IDA, что происходит, прямо скажем, регулярно. Поэтому рекомендуется пользоваться своими функциями – обертками, что уже не раз упоминалась в данной книге.

Дело в том, что переписать библиотеку своих функций намного проще, чем изменить все скрипты. К тому же библиотеку легко разместить в любом включаемом файле в IDA и тогда скрипты окажутся переносимыми, в противном же случае их придется редактировать каждый раз заново.

А вообще, если бы язык скриптов был бы на порядок популярнее, то автор бы не позволил себе такую роскошь как менять прототипы встроенных функций без сохранения обратной совместимости.

## МЕТОДЫ

Функция	Назначение
---------	------------

<code>void AddCodeXref(long From,long To,long flowtype);</code>	Создает перекрестную ссылку типа 'code'
<code>long DelCodeXref(long From,long To,int undef)</code>	Удаляет перекрестную ссылку типа 'code'
<code>long Rfirst (long From);</code>	Функция возвращает линейный адрес приемника первой перекрестной ссылки указанного источника
<code>long Rnext (long From,long current);</code>	Эта функция возвращает линейный адрес приемника очередной перекрестной ссылки в списке.
<code>long RfirstB (long To);</code>	Функция возвращает адрес следующего источника в списке перекрестных ссылок, расположенного по указанному приемнику
<code>long RnextB (long To,long current)</code>	Функция возвращает адрес следующего источника в списке перекрестных ссылок, расположенного по указанному приемнику
<code>long Rfirst0 (long From);</code>	Функция возвращает линейный адрес приемника перекрестной ссылки для заданного источника
<code>long Rnext0 (long From,long current);</code>	Эта функция по идее (а точнее следуя из сказанного в файле <code>idc.idc</code> ) должна отличаться от <code>Rnext</code> только отсутствием доступа к перекрестным ссылкам на следующую инструкцию. Однако из-за ошибок реализации функции <code>Rnext</code> она «не видит» такой тип ссылок и это делает обе функции полностью идентичными.
<code>long RfirstB0(long To);</code>	Функция возвращает линейный адрес источника перекрестной ссылки для заданного приемника.
<code>long RnextB0 (long To,long current);</code>	Эта функция по идее (а точнее следуя из сказанного в файле <code>idc.idc</code> ) должна отличаться от <code>RnextB</code> только отсутствием доступа к перекрестным ссылкам на следующую инструкцию.
<code>void add_dref(long From,long To,long drefType);</code>	Добавляет перекрестную ссылку типа 'data'
<code>void del_dref(long From,long To);</code>	Удаляет перекрестную ссылку типа 'data'
<code>long Dfirst (long From);</code>	Функция возвращает линейный адрес приемника первой перекрестной ссылки

	указанного источника
<code>long Dnext (long From,long current);</code>	Эта функция возвращает линейный адрес приемника очередной перекрестной ссылки в списке.
<code>long DfirstB (long To);</code>	Функция возвращает линейный адрес первого источника для указанного списка приемников
<code>long DnextB (long To,long current);</code>	Функция возвращает адрес следующего источника в списке перекрестных ссылок, расположенного по указанному приемнику.
<code>long XrefType(void);</code>	Эта функция возвращает тип перекрестной ссылки, которая была возвращена последним вызовом функций Rfirst, Rnext, RfirstB, RnextB, Dfirst, Dnext, DfirstB, DnextB.

### **void AddCodeXref(long From,long To,long flowtype);**

Функция создает кодовую перекрестную ссылку. IDA содержит мощный механизм, эмулирующий выполнение инструкций процессора и отслеживающий не только прямые, но даже косвенные ссылки.

Начиная с версии 3.74, она значительно превосходит в этом даже SOURCER, до этого лидирующий среди других дизассемблеров в умении восстанавливать перекрестные ссылки.

Перекрестные ссылки действительно очень облегчают анализ исследуемой программы. Допустим, встречается в тексте стока

```
Seg000:0123 DB 'Hello, World!',0Dh,0Ah,0
```

Как узнать – какой код ее выводит? Для этого, очевидно, нужно найти ссылку на смещение 0x123. Это IDA и делает автоматически. Благодаря перекрестным ссылкам можно трассировать исполнение программы, что позволяет лучше понять ее структуру.

IDA поддерживает два типа ссылок – на код и на данные. AddCodeXref, как нетрудно догадаться добавляет новую перекрестную ссылку на код, то есть инструкцию, изменяющую линейное исполнение программы.

Структура ссылок следующая:

From (Source) → To (Target)

Источник – это 32-битный линейный адрес начала инструкции, вызывающей изменение линейного исполнения кода, а приемник – это линейный адрес начала инструкции, на которую выполняется такой переход.

IDA отображает перекрестные ссылки в виде комментариев исключительно возле инструкции приемника.

```
seg000:0475          jnz     loc_0_47A
seg000:0477          mov     cx, 4
seg000:047A
```



```

seg000:047A loc_0_47A:                                ; CODE XREF:
seg000:0475j
seg000:047A                                         sub     bx, 10h

```

В приведенном выше примере показана перекрестная ссылка From == 0x475, To == 0x47A. Каким-то особым образом отмечать источник нет необходимости, поскольку он предполагается очевидным (в данном случае адрес указан в непосредственном операнде).

Подведя курсор к метке 'loc\_0\_47A' и нажав на Enter, можно перейти к приемнику. А что бы вновь вернуться к источнику, – достаточно кликнуть по перекрестной ссылке.

Разумеется, что на один и тот же приемник может ссылаться более одного источника.

Например:

```

seg000:0C4A                                         cmp     ah, 4Dh ; 'M'
seg000:0C4D                                         jnz    loc_0_C5A
seg000:0C4F                                         cmp     byte_0_F76, 9
seg000:0C54                                         ja     loc_0_C5A
seg000:0C56                                         inc     byte_0_F76
seg000:0C5A loc_0_C5A:                                ; CODE XREF:
seg000:0C4Dj
seg000:0C5A                                         ; seg000:0C54j

```

Немного не очевидно, но IDA поддерживает мульти - источники. Однако, действительно, возможно такое условное ветвление, что в зависимости от операнда приемник может варьироваться.

Например, широко распространенная команда JMP BX, используемая многими компиляторами объективно-ориентированных языков, да и в моделях Маркова, например, то же.

Тип ссылки указывается в постфиксе. В данном случае это 'j', что обозначает близкий (NEAR) условный или безусловный переход. IDA поддерживает четыре основных типа ссылок, которые перечислены в приведенной ниже таблице.

Определение		Пояснения	Уточнение	Легенда
fl_CF	1 6	Межсегментный вызов процедуры	Call Far	P
fl_CN	1 7	Внутрисегментный вызов процедуры	Call Near	p
fl_JF	1 8	Межсегментный переход	Jump Far	J
fl_JN	1 9	Внутрисегментный переход	Jump Near	j
fl_US	2 0	Определяется пользователем	User specified	u
fl_F	2 1	Следующая инструкция	Ordinary flow	^

Последний тип необходимо отметить особо. В контекстной помощи об этом нет никакого упоминания. Правда, заглянув в SDK можно узнать, что такие перекрестные ссылки предназначены для указаний на следующую инструкцию и вообще стоят особняком от всех остальных перекрестных ссылок.

Когда другие хранятся в базе Vtree, значение Ordinary flow извлекается из флагов ячеек виртуальной памяти.

Для чего это может понадобиться? Дело в том, что некоторые процессоры имеют такую запутанную архитектуру, что вычисление адреса следующей команды под час

представляется весьма нетривиальной задачей. Поэтому, не лишние возложить эту работу на плечи IDA, создав перекрестные ссылки соответствующего типа.

Однако, заметим, что тип перекрестных ссылок – понятие чисто условное и субъективное. Он был введен лишь затем, что бы в удобной читаемой форме предоставить пользователю дополнительную информацию о ссылке, облегчая ему работу по изучению программ.

IDA не следит за корректностью типов ссылок, - забота эта лежит исключительно на плечах кода, создающего ссылки. Ничто не мешает нам создать и вовсе бессмысленную ссылку, – например:

```

seg000:0C29          mov     dx, word ptr byte_0_F76
seg000:0C2D          call   sub_0_EF8
seg000:0C30          call   sub_0_F45
seg000:0C33          jb     loc_0_C69          ; CODE XREF:
seg000:0C29J

```

Очевидно, что инструкция `mov` не может изменять порядок выполнения кода, однако, IDA безболезненно позволяет создавать перекрестную ссылку с таким источником под типом «межсегментный переход»

Заметим, что в зависимости от некоторых настроек, при создании перекрестных ссылок типа «вызов процедуры» IDA может автоматически создавать процедуру на месте приемника, даже если сама ссылка ошибочная.

Поэтому для пользовательских скриптов рекомендуется использовать специально определенный тип, который гарантировано, не влечет за собой никаких последствий.

Заметим, что IDA позволяет создавать перекрестные ссылки, указывающие на середину инструкции. Они выделяются красным цветом и располагаются перед указанной инструкцией. Аналогично адрес округляется и при всех попытках перехода.

Операнд	Пояснения
From	Адрес источника перекрестной ссылки
To	Адрес приемника перекрестной ссылки
flowtype	Тип перекрестной ссылки (смотри таблицу, приведенную выше)

Обратите внимание, что функция не возвращает никакого значения, по которому можно было бы судить об успешности завершения операции. Вместо этого функция может при необходимости выводить пояснения в окно сообщений, но это не поможет в определении ошибки автономным скриптам.

### **long DelCodeXref(long From,long To,int undef);**

Функция удаляет перекрестную ссылку типа 'code'. Для этого необходимо знать ее источник (From) и приемник (To). Подробнее об этом можно прочитать в описании функции `AddCodeXref`.

Имеется возможность автоматически пометить приемник (и все последующие инструкции) как 'undefined', если на них больше не указывает ни одной ссылки. Для этого флаг 'undef' необходимо установить равным единице.

Например:

```

seg000:002B 11 02                               dw 211h
seg000:002D          ; -----
--
seg000:002D E8 AD 00          call   sub_0_DD   ; CODE
XREF: seg000:0395p

```

```

seg000:002D                                     ;
seg000:22F5p
seg000:0030 BE BA 02                             mov     si, 2BAh
seg000:0033 E8 A7 00                             call   sub_0_DD
seg000:0036 C3                                  retn

```

```

DelCodeXref(0x10395,0x1002D,1);
DelCodeXref(0x122F5,0x1002D,1);

```

```

seg000:002B 11 02                             dw 211h
seg000:002D E8                               db 0E8h
seg000:002E AD                               db 0ADh
seg000:002F 00                               db 0
seg000:0030 BE                               db 0BEh
seg000:0031 BA                               db 0BAh
seg000:0032 02                               db 2
seg000:0033 E8                               db 0E8h
seg000:0034 A7                               db 0A7h
seg000:0035 00                               db 0

```

Однако если то же проделать со следующим кодом, то даже после удаления всех перекрестных ссылок он не будет помечен, как undefined.

```

seg000:014C                                     stosb
seg000:014D                                     loop   loc_0_143
seg000:014F                                     ; CODE XREF:
seg000:014F loc_0_14F:
seg000:013Bj
seg000:014F                                     ;
seg000:0146j
seg000:014F                                     pop    es
seg000:0150                                     pop    ds

```

```

DelCodeXref(0x1013B,0x1014F,0);
DelCodeXref(0x10146,0x1014F,1);

```

```

seg000:014C                                     stosb
seg000:014D                                     loop   loc_0_143 ; =>
seg000:014F                                     ;
seg000:014F loc_0_14F:
seg000:014F                                     pop    es
seg000:0150                                     pop    ds ; <=

```

На самом же деле мы удалили не все перекрестные ссылки. IDA поддерживает и при необходимости автоматически создает так называемую ссылку на следующую команду.

Однако они не отображаются явно на экране, но, тем не менее, скрыто присутствует, увеличивая счетчик ссылок на единицу.

DelCodeXref проверяет значение счетчика, убеждается, что он больше нуля и не преобразует код в undefined.

Как и любую другую, эту ссылку можно удалить. Но для этого прежде нужно выяснить ее источник и приемник. Приемником, очевидно, будет линейный адрес начала текущей инструкции.

То есть `seg000:0x14F` или `0x1014F`, а источником линейный адрес начала предыдущей инструкции. В нашем случае это `0x1014D`.

Теперь можно вызвать функцию `DelCodeXref` и удалить эту перекрестную ссылку.

```
DelCodeXref (0x1014F, 0x1014D, 1);
```

Это сработало! Счетчик перекрестных ссылок стал равен нулю, и IDA пометила приемник и нижележащий код, как `undefined`.

Выше, при описании функции `MakeUndef` говорилось, что она удаляет все связанные инструкции. Теперь же, познакомившись, с архитектурой перекрестных ссылок можно уточнить это определение. IDA просто спускается по цепочке перекрестных ссылок и помечает `undefined` все инструкции на пути своего следования.

```
seg000:014C          stosb
seg000:014D          loop   loc_0_143
seg000:014D ; -----
seg000:014F unk_0_14F          db    7 ;
seg000:0150          db   1Fh ;
seg000:0151          db  0C7h ; |
seg000:0152          db    5 ;
seg000:0153          db  29h ; )
seg000:0154          db    0 ;
seg000:0155          db  0C7h ; |
seg000:0156          db   45h ; E
seg000:0157          db    1 ;
```

При этом функция возвратит единицу. Это сигнал того, что код успешно преобразован в `undefined`.

Операнд	Пояснения	
From	Адрес источника перекрестной ссылки	
To	Адрес приемника перекрестной ссылки	
undef	==1	Преобразовывать код в <code>undefined</code> , когда на него не останется ссылок
	==0	Не преобразовывать код в <code>undefined</code> , когда на него не останется ссылок
Return:	Пояснения	
	==1 если код успешно преобразован в <code>undefined</code>	

### **long Rfirst (long From);**

Функция возвращает линейный адрес приемника первой перекрестной ссылки указанного источника.

Подробнее о перекрестных ссылках было рассказано в описании функции `AddCodeXref` и `DelCodeXref`.

Хотя это не очевидно, источник может иметь несколько перекрестных ссылок. Например, когда используется инструкция, наподобие `JMP BX`.

Потом не нужно забывать, что практически все инструкции снабжены перекрестными ссылками на линейный адрес начала следующей инструкции.

Обратите внимание, что если по указанному линейному адресу существует перекрестная ссылка на следующую инструкцию, то функция возвратит именно ее. Не

смотря на то, что в idc.idc утверждается, что этот тип ссылок доступен Rnext (смотри описание ниже) на самом же деле, Rnext проходя список приемников перекрестных ссылок, игнорирует этот тип.

Если указан неверный источник, (то есть линейный адрес, не содержащий перекрестных ссылок) или источник перекрестной ссылки данных, то функция возвратит ошибку BADADDR

Примеры использования:

```

seg000:28C6                pop     di ; → источник
seg000:28C7                pop     si ; ← приемник

Message("0x%X \n",
Rfirst(0x128C6)
);

0x28C7

seg000:28CB                sub_0_2847      retn
seg000:28CB                sub_0_2847      endp

Message("0x%X \n",
Rfirst(0x128CB)
);

0xFFFFFFFF

seg000:2870                jmp     loc_0_2892 ; →
источник
seg000:2870 ; -----
seg000:2872                db     90h ; P
seg000:2873                db     90h ; P

seg000:2892 loc_0_2892:                ; CODE
XREF: seg000:2870j
seg000:2892                ; ←
приемник
seg000:2892                cmp     byte ptr [si], 22h ;
'''

Message("0x%X \n",
Rfirst(0x12870)
);

0x12892

```

Операнд	Пояснения
From	Линейный адрес источника перекрестной ссылки
Return	Пояснения
	Линейный адрес приемника первой перекрестной ссылки

**long Rnext (long From,long current);**

Эта функция возвращает линейный адрес приемника очередной перекрестной

ссылки в списке. При этом тип перекрестных ссылок, указывающих на следующую инструкцию (ordinary flows) игнорируется и никогда не может быть возвращен.

Для понимания того, как работает данная функция, рекомендуется прочесть описания функций AddCodeXref, DelCodeXref, Rfirst.

IDA хранит список перекрестных ссылок для каждого источника, отсортированный по адресам приемника. Первыми в нем идут те ссылки, чей линейный адрес приемника наименьший, за ними следующие.

Напоминаем, что функция игнорирует указатель на следующую инструкцию. Если же она завершилась успешно, то возвратит линейный адрес приемника перекрестной ссылки, следующей на current.

То есть current должен быть не обязательно точно равен адресу приемника текущей перекрестной ссылки в списке. Он может быть меньше его, но, разумеется, обязательно превосходить адрес приемника предыдущей ссылки.

Поясним это на примере:

```
seg000:0000                push    ax    ; CODE XREF:
seg000:2864p
seg000:0000                ; ← приемник

seg000:2864                call    bx    ; → источник

seg000:2869 loc_0_286          ; CODE XREF:
seg000:2864p
seg000:2869                inc     si    ; ← приемник

seg000:2892 loc_0_2892:          ; CODE XREF:
seg000:2864p
seg000:2892                ; ← приемник
seg000:2892                cmp     byte ptr [si], 22h ;
""
```

Пусть при изучении программы было определено, что BX может принимать следующие значения – 0x0, 0x2869, 0x2892. В этом случае по линейному адресу seg000:2864 будет расположено три перекрестные ссылки на соответствующие приемники.

Точнее, их будет даже четыре, с учетом ссылки на следующую инструкцию, но, поскольку Rnext никогда не возвращает ее, то достаточно рассмотреть только выше упомянутые три.

IDA формирует по линейному адресу 0x12864 следующий список приемников: {0x10000, 0x12869, 0x12892} Вот эти адреса и будут возвращаться при прохождении списка функцией Rnext.

Не обязательно начинать первый вызов с Rfirst (смотри описание выше). Как уже упоминалось, Rnext хранит указатель на текущую ссылку не во внутренней скрытой переменной, а принимает его как параметр. Таким образом, это дает нам возможность легко манипулировать ее значением, управляя поведением функции.

Вообще не понятно, зачем понадобилось вводить Rfirst. Ведь первую перекрестную ссылку можно найти с помощью Rnext – и это будет следующая ссылка за нулем. Очевидно, что Rnext(0x12864,0) вернет 0x10000 – первую перекрестную ссылку в списке. Следовательно, Rnext(X, 0) идентична Rfirst.

На самом деле тут нас поджидает небольшой сюрприз. Функция Rnext, проходя список, не обнаруживает в нем ссылок на следующую команду. Это не является ошибкой, а документированной особенностью IDA.

Например:

```
auto a;
a=0;
```

```

for (;;)
{
    a=Rnext(ScreenEA(), a);
    if (a==-1) break;
    Message("0x%X \n", a);
}

```

Операнд	Пояснения
Form	Линейный адрес источника списка перекрестных ссылок
Current	Текущий адрес
Return	Пояснения
	Следующий адрес в списке
	-1 если список исчерпан или отсутствует источник

### long RfirstB (long To);

Функция возвращает линейный адрес первого источника для указанного списка приемников.

Для понимания этого, рекомендуется прочесть описания функций AddCodeXref, DelCodeXref, Rfirst, Rnext.

Очевидно, что по одному и тому же линейному адресу может существовать более одного приемника перекрестных ссылок.

Например:

```

seg000:013B          jz      loc_0_14F  ; → источник
seg000:013D          mov     ds, ax
seg000:013F          mov     ax, cs
seg000:0141          mov     es, ax
seg000:0143          lodsb
seg000:0144          cmp     al, 21h ; '!'
seg000:0146          jb     loc_0_14F  ; → источник
seg000:0148          cmp     al, 7Ah ; 'z'
seg000:014A          ja     loc_0_14F  ; → источник
seg000:014C          stosb
seg000:014D          loop   loc_0_143 ; → источник
seg000:014F          loc_0_14F:
seg000:014F          ; CODE XREF:
seg000:013Bj
seg000:014F          ; seg000:0146j ...
seg000:014F          ; ← приемник
seg000:014F          pop     es

```

Источник по адресу seg000:0x14D помечен не случайно. Он, разумеется, не имеет никакого отношения к операнду loc\_0\_143, а представляет собой перекрестную ссылку на следующую команду.

Реализация этой функции повторяет особенность реализации Rfirst. Действительно, рассмотрим список источников перекрестных ссылок, который IDA сформировала по адресу seg000:0x14F – {0x1013B, 0x10146, 0x1014A, 0x1014C}.

Вполне естественно ожидать, что вызов RfirstB должен был бы вернуть первый – самый наименьший из них.

Однако же, вместо него возвращается источник ссылка на следующую инструкцию, то есть 0x1014A. Покажем это ниже:

```

    Message("0x%X \n",
RfirstB(0x1014F)
);

```

0x1014A

Как получить действительно первый элемент источника списка можно прочитать в описании функции RnextB

Операнд	Пояснения
To	Линейный адрес приемника списка перекрестных ссылок
Return	Пояснения
	Источник ссылки на следующую инструкцию или если ее нет, то первый адрес в списке.
	-1 если список исчерпан или отсутствует источник

### long RnextB (long To,long current);

Функция возвращает адрес следующего источника в списке перекрестных ссылок, расположенного по указанному приемнику.

Для более глубокого понимания принципов работы рекомендуется ознакомиться с описанием функций AddCodeXref, DelCodeXref, Rnext, RfirstB.

Очевидно, что по одному и тому же линейному адресу может существовать более одного приемника перекрестных ссылок.

Например:

```

seg000:013B      jz      loc_0_14F ; → источник
seg000:013D      mov     ds, ax
seg000:013F      mov     ax, cs
seg000:0141      mov     es, ax
seg000:0143      lodsb
seg000:0144      cmp     al, 21h ; '!'
seg000:0146      jb     loc_0_14F ; → источник
seg000:0148      cmp     al, 7Ah ; 'z'
seg000:014A      ja     loc_0_14F ; → источник
seg000:014C      stosb
seg000:014D      loop   loc_0_143 ; → источник
seg000:014F      loc_0_14F: ; CODE XREF:
seg000:013Bj
seg000:014F      ;
seg000:0146j ...
seg000:014F      ; ← приемник
seg000:014F      pop     es

```

Реализация этой функции повторяет особенность реализации Rnext. Не смотря на то, что в idc.idc утверждается будто бы эта функция «видит» тип перекрестных ссылок, указывающих на следующую команду, в действительности этого не происходит. И выделенный красным цветом адрес источника функция не вернет никогда. {0x1013B, 0x10146, 0x1014A, 0x1014C}.



Поскольку RnextB хранит текущий адрес не во внутренней переменной, а в передаваемом параметре, то существует возможность, модифицируя его, управлять работой функции.

Так, например, RnextB(ScreenEA(), 0) гарантированно вернет следующий за ним адрес, то есть 0x1013B, а пройти весь список (за исключением ссылок на следующую инструкцию) можно с помощью следующего кода:

```

auto a;
a=0;
for (;;)
{
a=RnextB(ScreenEA(), a);
if (a== -1) break;
Message("0x%X \n", a);
}

0x1013B
0x10146
0x1014C

```

Немного модернизировав код можно добиться того, что бы на экран выдавался действительно весь список источников, включая и ссылки на следующую команду.

```

auto a;
a=0;
for (;;)
{
a=RnextB(ScreenEA(), a);
if (a== -1) break;
if (a>RfirstB(ScreenEA()))
Message("0x%X \n",
RfirstB(ScreenEA())
);
Message("0x%X \n", a);
}

0x1013B
0x10146
0x1014A
0x1014C

```

Функция возвращает ошибку BADADDR, если список исчерпан, (то есть текущий адрес наибольший в списке) или не существует.

Операнд	Пояснения
To	Линейный адрес приемника списка перекрестных ссылок
Current	Текущий адрес
Return	Пояснения
	Следующий адрес в списке
	-1 если список исчерпан или отсутствует источник

## long Rfirst0 (long From);

Функция возвращает линейный адрес приемника перекрестной ссылки для заданного источника.

Практически идентична Rfirst, за тем исключением, что не имеет доступа к ссылкам на следующую инструкцию, поэтому возвращает действительно первый элемент списка линейный адресов приемников.

Поэтому ее рекомендуется использовать в паре с функцией Rnext, впрочем, Rnext(xxx, 0) возвращает идентичный результат и хотя работает ничуть не быстрее, но немного экономит на компактности кода.

Для понимания этого рекомендуется ознакомиться с описанием функций AddCodeXref, DelCodeXref, Rfirst, Rnext

Если указан неверный источник, (то есть линейный адрес, не содержащий перекрестных ссылок) или источник перекрестной ссылки данных, то функция возвратит ошибку BADADDR

Примеры использования:

```
seg000:28C6                pop     di ; → источник
seg000:28C7                pop     si ; ← приемник

Message("0x%X \n",
Rfirst0(0x128C6)
);

0xFFFFFFFF

seg000:28CB                sub_0_2847    retn
seg000:28CB                endp

Message("0x%X \n",
Rfirst0(0x128CB)
);

0xFFFFFFFF

seg000:2870                jmp     loc_0_2892 ; →
источник
seg000:2870                nop

seg000:2892 loc_0_2892:                ; CODE
XREF: seg000:2870j
seg000:2892                ; ←
приемник
seg000:2892                cmp     byte ptr [si], 22h ;
""

Message("0x%X \n",
Rfirst0(0x12870)
);

0x12892
```

Операнд	Пояснения
From	Линейный адрес источника перекрестной ссылки

Return	Пояснения
	Линейный адрес приемника первой перекрестной ссылки

### **long Rnext0 (long From,long current);**

Эта функция по идее (а точнее следуя из сказанного в файле idc.idc) должна отличаться от Rnext только отсутствием доступа к перекрестным ссылкам на следующую инструкцию.

Однако из-за особенностей реализации функции Rnext она «не видит» такой тип ссылок и это делает обе функции полностью идентичными.

Поэтому никакого описания здесь не приводится, поскольку пришлось бы полностью повторить все сказанное об Rnext.

Чаще всего ссылка на следующую команду не требуется. В этих случаях и следует применять вызов Rnext0.

В противном случае придется воспользоваться листингом, приведенным ниже.

```

auto a;
a=0;
for (;;)
{
a=RnextB(ScreenEA(),a);
if (a==-1) break;
if (a>RfirstB(ScreenEA()))
Message("0x%X \n",
RfirstB(ScreenEA())
);
Message("0x%X \n",a);
}

```

Операнд	Пояснения
Form	Линейный адрес источника списка перекрестных ссылок
Current	Текущий адрес
Return	Пояснения
	Следующий адрес в списке
	-1 если список исчерпан или отсутствует источник

### **long RfirstB0(long To);**

Функция возвращает линейный адрес источника перекрестной ссылки для заданного приемника.

Практически идентична RfirstB, за тем исключением, что не имеет доступа к ссылкам на следующую инструкцию, поэтому возвращает действительно первый элемент списка линейный адресов источников.

Поэтому ее рекомендуется использовать в паре с функцией RnextB, впрочем, RnextB(ххх, 0) возвращает идентичный результат и хотя работает ничуть не быстрее, но немного экономит на компактности кода.

Для понимания этого рекомендуется ознакомиться с описанием функций AddCodeXref, DelCodeXref, RfirstB, RnextB

Если указан неверный источник, (то есть линейный адрес, не содержащий перекрестных ссылок) или источник перекрестной ссылки данных, то функция возвратит ошибку BADADDR

Операнд	Пояснения
To	Линейный адрес приемника списка перекрестных ссылок
Return	Пояснения
	Источник ссылки на следующую инструкцию или если ее нет, то первый адрес в списке.
	-1 если список исчерпан или отсутствует источник

### **long RnextB0 (long To,long current);**

Эта функция по идее (а точнее следуя из сказанного в файле idc.idc) должна отличаться от RnextB только отсутствием доступа к перекрестным ссылкам на следующую инструкцию. Однако из-за особенностей реализации функции Rnext она «не видит» такой тип ссылок и это делает обе функции полностью идентичными.

Поэтому никакого описания здесь не приводится, поскольку пришлось бы полностью повторить все сказанное об Rnext.

Чаще всего ссылка на следующую команду не требуется. В этих случаях и следует применять вызов Rnext0. В противном случае придется воспользоваться листингом, приведенным выше.

Операнд	Пояснения
To	Линейный адрес приемника списка перекрестных ссылок
Current	Текущий адрес
Return	Пояснения
	Следующий адрес в списке
	-1 если список исчерпан или отсутствует источник

### **void add\_dref(long From,long To,long drefType);**

Подробнее об архитектуре перекрестных ссылках было сказано в описании функции AddCodeXref.

Для удобства IDA поддерживает две группы перекрестных ссылок – на данные и на код. Каждая группа со своим набором функций и возможностей.

Типы, поддерживаемых перекрестных ссылок на данные следующие:

Определение		Пояснения	Легенда
dr_O	1	Смещение (Offset)	o
dr_W	2	Запись (Write)	w
dr_R	3	Чтение (Read)	r
dr_T	4	Пользовательский тип	t

С первого взгляда кажется, что можно создать перекрестную ссылку на данные с помощью вызова AddCodeXref, только лишь указав соответствующий тип ссылки.

Например:

```
AddCodeXref (0x10148, 0x1014C, 2);
```

```

seg000:014C loc_0_14C: ; CODE XREF:
seg000:0148w

```

На самом же деле постфикс (в данном случае 'w') играет только информационную роль и ничуть не влияет на тип ссылки, которая так и осталась кодовой, что видно по предваряющему ее ключевому слову.

Сравните это со следующим примером:

```

add_dref(0x10148,0x1014C,2);

seg000:014A          ja      loc_0_14F

                seg000:014C          ; DATA
XREF: seg000:0148w
seg000:014C          stosb

```

Обратите внимание, что IDA никак не контролирует корректность ссылки, полностью перекладывая эту работу на код, вызывающий эту функцию.

Тип ссылки играет чисто информационную роль и служит для ускорения анализа дизассемблируемой программы. Никаких других влияний на работу IDA он не оказывает.

Однако стоит все же придерживаться единой схемы наименования перекрестных ссылок, что бы ни приводить пользователя в замешательство.

#### dr\_0 - Смещение (Offset)

Под смещением понимается любое (прямое или косвенное) обращение к адресу ячейки данных.

Например:

```

Seg000:0301 push   offset loc_0_30A      ; → источник
seg000:030A loc_0_30A:                ; DATA XREF:
seg000:0301o
seg000:030A                ; ← приемник

seg000:3000 DW offset byte_0_293A   ; → источник
Seg000:293A byte_0_293A DB ?        ; DATA XREF:
seg000:3000o
Seg000:293A                ; ← приемник

```

Но и в том числе и такие инструкции, где смещение не указано явно, а только подразумевается.

Например, LEA.

#### Dr W Запись (Write)

Любая инструкция, производящая прямую запись в ячейку.

```

seg000:2928          mov     cs:byte_0_2939,1 ; → источник
seg000:2939*          DB      ?                ; DATA XREF:
seg000:2928w
seg000:2939                ; ← приемник

seg000:0E5F          dec     word_0_F72        ; → источник

```

```

seg000:0F72*           DW      ?           ; DATA XREF
seg000:0E5Fw
seg000:0F72           ; ← приемник

```

Однако, для таких инструкций, как, например, MOVS IDA автоматически не создает перекрестных ссылок. Но это могут делать продвинутые пользовательские скрипты.

## Dr R Чтение (Read)

Любая инструкция, производящая прямое чтение ячейки. Например:

```

seg000:0D08           mov     ax, word_0_F72 ; → источник
seg000:0F72*word_0_F72 dw 0           ; DATA XREF:
seg000:0D08r
seg000:0F72           ; ← приемник

```

При этом инструкции, выполняющие цикл операций чтение – вычисление – запись, IDA всегда относит к типу dr\_w, а не dr\_r

Обратите внимание, что функция add\_dref не возвращает результата успешности операции, поэтому для того, что бы определить действительно ли была создана перекрестная ссылка, или нет – приходится прибегать к полному прохождению списка в попытках найти в нем «свой» адрес.

Операнд	Пояснения
From	Адрес источника перекрестной ссылки
To	Адрес приемника перекрестной ссылки
Dreftype	Тип перекрестной ссылки (смотри таблицу, приведенную выше)

## void del\_dref(long From,long To);

Функция позволяет удалять перекрестную ссылку на данные. Для этого необходимо знать линейные адреса ее источника и приемника.

Например:

```

seg000:2331           mov     word_0_2934, ax ; → источник
seg000:2934*word_0_2934 dw 0           ; DATA XREF:
seg000:2331w
seg000:2934           ; ← приемник

Del_dref (0x2331, 9x2934) ;

seg000:2331           mov     word_0_2934, ax ;
seg000:2934*word_0_2934 dw 0           ;
seg000:2934           ;

```

К сожалению, нет никакой возможности узнать о результате успешности операции, поскольку функция возвращает тип void.

Часто путают источник и приемник местами, что приводит к ошибкам. Необходимо запомнить, что IDA всегда создает комментарий к перекрестной ссылке возле ее приемника, а не источника.

Поэтому, что бы удалить указанную перекрестную ссылку необходимо воспользоваться следующим кодом:

```
Del_dref(0x2331,9x2934);
```

Разумеется, что эта функция не пригодна для удаления перекрестных ссылок на код.

Операнд	Пояснения
From	Адрес источника перекрестной ссылки
To	Адрес приемника перекрестной ссылки

### long Dfirst (long From);

Функция возвращает линейный адрес приемника первой перекрестной ссылки указанного источника.

Подробнее о перекрестных ссылках было рассказано в описании функция AddCodeXref, add\_dref.

Хотя это не очевидно, источник может иметь несколько перекрестных ссылок. Например, когда используется инструкция, наподобие `mov ax,[BX]`.

Если указан неверный источник, (то есть линейный адрес, не содержащий перекрестных ссылок) или источник перекрестной ссылки данных, то функция возвратит ошибку BADADDR

Пример использования:

```
seg000:2331          mov     word_0_2934, ax ; → источник
seg000:2934*word_0_2934  dw 0          ; DATA XREF:
seg000:2331w
seg000:2934          ; ← приемник

Message("0x%X \n",
Dfirst(0x12331)
);

0x12934
```

Операнд	Пояснения
From	Линейный адрес источника перекрестной ссылки
Return	Пояснения
	Линейный адрес приемника первой перекрестной ссылки

### long Dnext (long From,long current);

Эта функция возвращает линейный адрес приемника очередной перекрестной ссылки в списке.

Для понимания того, как работает данная функция, рекомендуется прочесть описания функций AddCodeXref, add\_dref, Dfirst.

IDA хранит список перекрестных ссылок для каждого источника, отсортированный по адресам приемника. Первыми в нем идут те ссылки, чей линейный адрес приемника наименьший, за ними следующие.

Если функция завершится успешно, то возвратит линейный адрес приемника перекрестной ссылки, следующей на current.

То есть current должен быть не обязательно точно равен адресу приемника текущей перекрестной ссылки в списке. Он может быть меньше его, но, разумеется, обязательно превосходить адрес приемника предыдущей ссылки.

Поясним это на примере:

```

seg000:2331          mov     word_0_2934, ax ; → источник
seg000:26C1          cmp     ax, word_0_2934 ; → источник
seg000:277B          cmp     dx, word_0_2934 ; → источник

seg000:2934* word_0_2934  dw 0          ; DATA XREF:
seg000:2331w
seg000:2934*
seg000:26C1r
seg000:2934*
seg000:277Br
seg000:2934          ; ← приемник

```

IDA сформирует по линейному адресу 0x12934 следующий список приемников: {0x12331, 0x126C1, 0x1277B} Вот эти адреса и будут возвращаться при прохождении списка функцией Dnext.

Не обязательно начинать первый вызов с Dfirst (смотри описание выше). Как уже упоминалось, Dnext хранит указатель на текущую ссылку не во внутренней скрытой переменной, а принимает его как параметр.

Таким образом, это дает нам возможность легко манипулировать ее значением, управляя поведением функции.

Вообще не понятно, зачем понадобилось вводить Dfirst. Ведь первую перекрестную ссылку можно найти с помощью Dnext – и это будет следующая ссылка за нулем.

Очевидно, что Dnext(0x12934,0) вернет 0x12331 – первую перекрестную ссылку в списке. Следовательно, Dnext(X, 0) идентична Dfirst.

Вывести на экран адреса всех источников перекрестных ссылок поможет следующий код:

```

auto a;
a=0;
for (;;)
{
a=Dnext(ScreenEA(), a);
if (a==-1) break;
Message("0x%X \n", a);
}

```

Операнд	Пояснения
Form	Линейный адрес источника списка перекрестных ссылок
Current	Текущий адрес
Return	Пояснения
	Следующий адрес в списке



	-1 если список исчерпан или отсутствует источник
--	--

### long DfirstB (long To);

Функция возвращает линейный адрес первого источника для указанного списка приемников.

Для понимания этого, рекомендуется прочесть описания функций AddCodeXref, add\_dref, Dfirst.

Очевидно, что по одному и тому же линейному адресу может существовать более одного приемника перекрестных ссылок.

Например:

```

seg000:2331                mov     word_0_2934, ax ; → источник
seg000:26C1                cmp     ax, word_0_2934 ; → источник
seg000:277B                cmp     dx, word_0_2934 ; → источник
seg000:2934* word_0_2934   dw 0                ; DATA XREF:
seg000:2331w
seg000:2934*                ;
seg000:26C1r
seg000:2934*                ;
seg000:277Br

```

Рассмотрим список источников перекрестных ссылок, который IDA сформировала по адресу seg000:0x2934 – {0x12331, 0x126C1, 0x1277B}.

Вызов DfirstB возвратит первый из них – с наименьшим линейным адресом.

```

Message("0x%X \n",
BfirstB(0x12934)
);

```

0x12331

Операнд	Пояснения
To	Линейный адрес приемника списка перекрестных ссылок
Return	Пояснения
	Источник ссылки на следующую инструкцию или если ее нет, то первый адрес в списке.
	-1 если список исчерпан или отсутствует источник

### long DnextB (long To,long current);

Функция возвращает адрес следующего источника в списке перекрестных ссылок, расположенного по указанному приемнику.

Для более глубокого понимания принципов работы рекомендуется ознакомиться с описанием функций AddCodeXref, add\_dref, Dnext

Очевидно, что по одному и тому же линейному адресу может существовать более одного приемника перекрестных ссылок.

Например:

```

seg000:2331          mov     word_0_2934, ax ; → источник
seg000:26C1          cmp     ax, word_0_2934 ; → источник
seg000:277B          cmp     dx, word_0_2934 ; → источник
seg000:2934* word_0_2934 dw 0 ; DATA XREF:
seg000:2331w
seg000:2934*
seg000:26C1r
seg000:2934*
seg000:277Br

```

Поскольку DnextB хранит текущий адрес не во внутренней переменной, а в передаваемом параметре, то существует возможность, модифицируя его, управлять работой функции.

Так, например, DnextB(ScreenEA(), 0) гарантированно вернет следующий за ним адрес, а пройти весь список (за исключением ссылок на следующую инструкцию) можно с помощью следующего кода:

```

auto a;
a=0;
for (;;)
{
a=DnextB(ScreenEA(), a);
if (a==-1) break;
Message("0x%X \n", a);
}

0x12331
0x126C1
0x1277B

```

Функция возвращает ошибку BADADDR, если список исчерпан, (то есть текущий адрес наибольший в списке) или не существует.

Операнд	Пояснения
To	Линейный адрес приемника списка перекрестных ссылок
Current	Текущий адрес
Return	Пояснения
	Следующий адрес в списке
	-1 если список исчерпан или отсутствует источник

## long XrefType(void);

Эта функция возвращает тип перекрестной ссылки, которая была возвращена последним вызовом функций Rfirst, Rnext, RfirstB, RnextB, Dfirst, Dnext, DfirstB, DnextB.

Обратите внимание, что функция не принимает никаких параметров, а взаимодействует исключительно с внутренними переменными IDA.

При этом она имеет одну грубую ошибку (точнее недостаток, который вряд ли будет скоро исправлен) реализации.

Возвращаемое значение принадлежит либо множеству определений fl\_x или dr\_x. Однако как отмечалось выше, типы перекрестных ссылок понятие число условное и та же функция AddCodeXref принимает в качестве параметра определения из множества dr\_x, и даже успешно создает такие перекрестные ссылки, однако, являющиеся все равно перекрестными ссылками на **код**.

Поэтому невозможно гарантированно определить тип перекрестной ссылки по возвращаемому функцией XrefType значению.

Например:

```
seg000:014C   loc_0_14C:                                     ; CODE XREF:
seg000:0148w

Rfirst(0x10148);
Message("0x%X \n",
XrefType()
);

0x15
```

Функция вернула тип dr\_W, но это еще не дает возможности утверждать, что эта перекрестная ссылка указывает на данные.

	Пояснения
Return	Тип перекрестной ссылки возвращенной последним вызовом, манипулирующей с ней функции.

## ТОЧКИ ВХОДА

### АРХИТЕКТУРА ТОЧЕК ВХОДА

Поддержка точек входа (Entry Point) самый мало проработанный элемент архитектуры IDA. Причиной тому абсолютная ненужность их для пользователя. В большинстве случаев даже не требуется знать, что такое понятие есть и поддерживается IDA

С одной стороны, вполне логично, что каждый файл имеет некоторую точку, с которой начинается его выполнение, причем эта точка может не совпадать с началом файла. Например, exe файл может начинать выполнение с любой своей точки (между прочим, возможно даже выходящей за границы файла, - но это относится к недокументированным особенностям MS-DOS совместимых операционных систем и поэтому не будет больше заострять на этом внимания)

Адрес регистра (регистров) – указателя команд в момент передачи управления загруженному файлу и называется точкой входа.

Таким образом, любой дизассемблер как минимум должен быть осведомлен, как вычислить этот адрес. Чаще всего он присутствует в заголовке файла (значит,

дизассемблер должен понимать его формат), реже предполагается по умолчанию – так для com файлов он всегда расположен по адресу 0x100, но для бинарных файлов (дампов RAM, например), точка входа не может и вовсе не иметь смысла. Поскольку управление может быть передано на множество мест, в зависимости от обстоятельств.

Поэтому, с первого взгляда, говорить о поддержке точек входа можно только на уровне ядра дизассемблера, скрытом от пользователя, то есть собственно говорить не о чем и не за чем – манипулировать точками входа лучше предоставить ядру.

В общих чертах так оно и есть – набор функций, взаимодействующий с точками входа, очень ограничен и, откровенно говоря, не полон. Так, например, не предусмотрено функции удаления точек входа, в том числе и созданных пользователем.

Однако, создавать свои точки входа в большинстве случаев нет нужды, а вот получить адреса существующих требуется очень часто – должны же скрипты знать с какого адреса начинается выполнение программы?

Для этого предусмотрена функция `long GetEntryPoint(long ordinal)`, чем потребности рядового пользователя с лихвой удовлетворяются.

Но разного рода извращениям и маньякам этого очень мало. Например, при анализе ПЗУ сталкиваешься с тем, что код может начинать выполняться с десятков разных мест (например, обработчиков прерываний) и хорошим решением будет создать собственные точки входа (IDA, разумеется, бессильна их определить) и потом взаимодействовать с ними как интерактивно, так и программно (из скриптов).

Впрочем, многие просто создают в нужных местах функции, а точками входа пренебрегают. В чем-то такая позиция верна, поскольку точки входа не дают никаких преимуществ за исключением того, что явно указывают, что с этих адресов может начинаться выполнение программы.

Еще IDA предвывает их имена директивой `public`, делая их общедоступными. Но то же можно сделать вручную.

Подытоживая сказанное выше можно сказать, что вникать в технические подробности организации точек входа необязательно даже опытным пользователям, а тем более самостоятельно манипулировать ими.

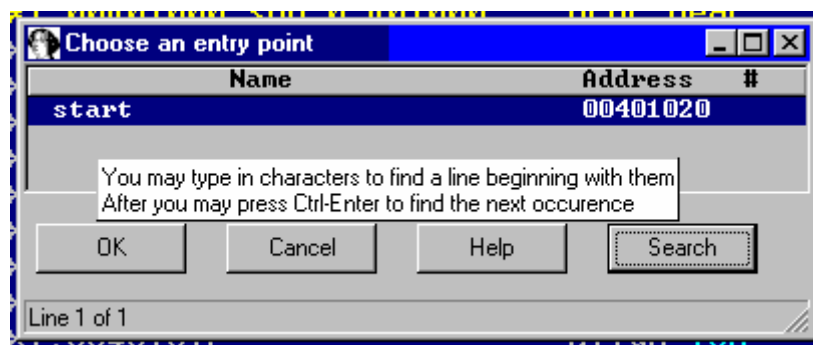
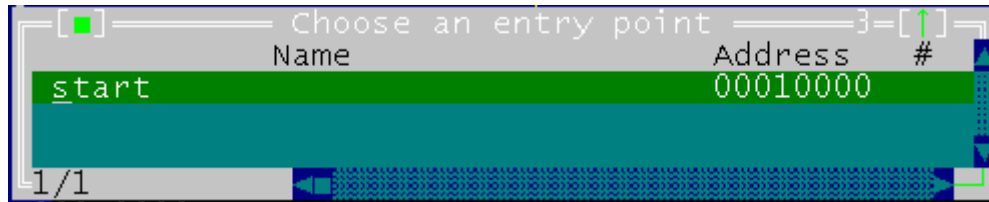
## МЕТОДЫ

Функция	Описание
<code>long GetEntryPointQty(void);</code>	Возвращает число точек входа
<code>success AddEntryPoint(long ordinal, long ea, char name, long makecode)</code>	Добавляет новую точку входа
<code>long GetEntryOrdinal(long index);</code>	Возвращает ординал точки входа по ее индексу
<code>long GetEntryPoint(long ordinal)</code>	Возвращает адрес точки входа по ординалу
<code>success RenameEntryPoint(long ordinal, char name);</code>	Переименовывает точку входа

**`long GetEntryPointQty(void);`**

Функция возвращает число точек входа (Entry Points). Обычно IDA создает только одну точку входа, адрес которой извлекается из заголовков исполняемого файла. Но иногда возникает потребность в создании более, чем в одной точке входа.

Например, PE файл, имеющий DOS-заглушку. Если мы захотим дизассемблировать последнюю, то необходимо добавить новую точку входа «вручную», поскольку IDA предпочитает в большинстве случаев обходиться всего лишь одним Entry Point



Пример использования:

```
Message("0x%X \n", GetEntryPointQty());
```

```
0x1
```

Операнд	Пояснения
Return	Число точек входа

**success AddEntryPoint(long ordinal,long ea,char name,long makecode);**

Добавляет новую точку входа. Будьте внимательны при вызове этой функции, ведь удалить созданную точку входа уже не удастся!

Для доступа к точке входа необходимо знать ее ординал, который задается пользователем при вызове функции. Если он равен нулю, то IDA установит его равным линейному адресу точки входа (и строго говоря при этом ординал не создается). С одним и тем же ординалом может существовать только одна точка входа.

При генерации точки входа IDA вставляет директиву ассемблера PUBLIC.

```
seg000:0000          public start
seg000:0000 start     proc near
seg000:0000          push    ax
seg000:0001          mov     cx, 1
```

```

seg000:0004          shl     cx, 1
seg000:0006          add     sp, cx
seg000:0008          push   ax
seg000:0009          add     sp, cx
seg000:000B          retn
seg000:000B start    endp ; sp = -4
seg000:000B          seg000
seg000:000B          ends

```

```
AddEnrtyPoint(1,0x10006,"NewEntryPoint",0);
```

```

seg000:0000          public start
seg000:0000 start    proc near
seg000:0000          push   ax
seg000:0001          mov    cx, 1
seg000:0004          shl   cx, 1
seg000:0006          public NewEntryPoint
seg000:0006 NewEntryPoint:
seg000:0006          add     sp, cx
seg000:0008          push   ax
seg000:0009          add     sp, cx
seg000:000B          retn
seg000:000B start    endp ; sp = -4
seg000:000B          seg000
seg000:000B          ends

```

Если попытаться создать более одной точки с идентичными именами, то IDA добавит к последнему знак прочерка и номер имени, начиная с нуля.

```
AddEnrtyPoint(2,0x10009,"NewEntryPoint",0);
```

```

seg000:0000          public start
seg000:0000 start    proc near
seg000:0000          push   ax
seg000:0001          mov    cx, 1
seg000:0004          shl   cx, 1
seg000:0006          public NewEntryPoint
seg000:0006 NewEntryPoint:
seg000:0006          add     sp, cx
seg000:0008          push   ax
seg000:0009          public NewEntryPoint_0
seg000:0009 NewEntryPoint_0:
seg000:0009          add     sp, cx
seg000:000B          retn
seg000:000B start    endp ; sp = -4
seg000:000B          seg000
seg000:000B          ends

```

Если попытаться создать точку входа с уже существующим ординалом, то она не будет создана, а функция вернет ошибку.

```

Message("0x%X \n",
        AddEnrtyPoint(2,0x10009,"MyEntryPoint",0)
        );

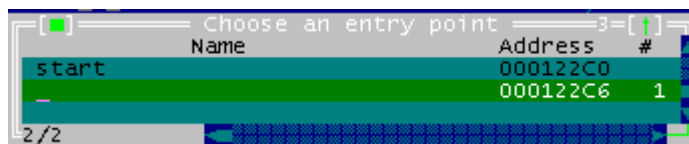
```

0x0

Допускается создание точки входа без имени. При этом она не будет отображена на экране, но появится в списке точек входа.

```
AddEntryPoint(1, 0x122C6, "", 1);
```

```
seg000:22C0                public start
seg000:22C0 start          proc near
seg000:22C0                call     sub_0_22DD
seg000:22C3                call     sub_0_2325
seg000:22C6                call     sub_0_235B
seg000:22C9                call     sub_0_2374
seg000:22CC                call     sub_0_23B6
seg000:22CF                call     sub_0_23F8
```



Если по указанному адресу уже существует метка (или функция), то она будет переименована.

```
seg000:002A sub_0_2A      proc near
seg000:002A                mov     si, 211h
seg000:002D                call   sub_0_DD
seg000:0030                mov     si, 2BAh
seg000:0033                call   sub_0_DD
seg000:0036                retn
seg000:0036 sub_0_2A      endp
```

```
AddEntryPoint(8, 0x1002A, "EntryPoint", 0);
```

```
seg000:002A                public EntryPoint
seg000:002A EntryPoint      proc near
seg000:002A                mov     si, 211h
seg000:002D                call   sub_0_DD
seg000:0030                mov     si, 2BAh
seg000:0033                call   sub_0_DD
seg000:0036                retn
seg000:0036 EntryPoint      endp
```

Если же по указанному адресу уже существует точка входа, то она не будет затерта новой, и по одному адресу будут расположены две точки входа. При этом имя предыдущей точки входа переместиться в комментарий.

```
AddEntryPoint(9, 0x1002A, "NewEntryPoint", 0);
```



```

seg000:002A
seg000:002A      public NewEntryPoint
seg000:002A NewEntryPoint  proc near                ; EntryPoint
seg000:002A      mov     si, 211h
seg000:002D      call   sub_0_DD
seg000:0030      mov     si, 2BAh
seg000:0033      call   sub_0_DD
seg000:0036      retn
seg000:0036 NewEntryPoint  endp

```

Если флаг makecode будет установлен в единицу то IDA при необходимости формирует функцию и дизассемблирует инструкции.

```

seg000:002A      db  0BEh
seg000:002B      db  11h
seg000:002C      db   2
seg000:002D      db  0E8h
seg000:002E      db  0ADh
seg000:002F      db   0
seg000:0030      db  0BEh
seg000:0031      db  0BAh
seg000:0032      db   2
seg000:0033      db  0E8h
seg000:0034      db  0A7h
seg000:0035      db   0
seg000:0036      db  0C3h

```

```
AddEntryPoint(1, 0x1002A, "MyEntryPoint", 1);
```

```

seg000:002A
seg000:002A      public MyEntryPoint
seg000:002A MyEntryPoint  proc near
seg000:002A      mov     si, 211h
seg000:002D      call   sub_0_DD
seg000:0030      mov     si, 2BAh
seg000:0033      call   sub_0_DD
seg000:0036      retn
seg000:0036 MyEntryPoint  endp

```

Операнд	Пояснения	
ordinal	Ординал функции	
Ea	Линейный адрес конца команды	
Name	Имя точки входа	
makecode	==makecode	Пояснения
	==1	Преобразовывать undefine в инструкции
	==0	Не преобразовывать undefine в инструкции
Return	Завершение	Пояснения
	1	Успешно



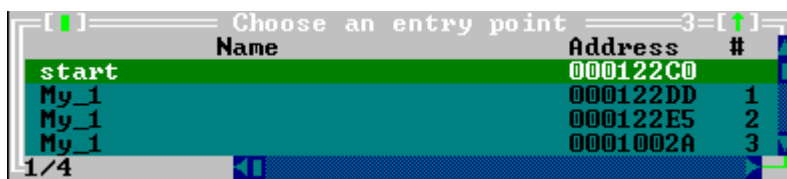
	0	Ошибка
--	---	--------

### long GetEntryOrdinal(long index);

Возвращает ординал точки входа по порядковому номеру из списка. Index может принимать значения от нуля до GetEntryPointQty()-1. Все точки входа (если их больше одной) хранятся в несортированном списке, расположенные в порядке их создания.

Если запросить несуществующий индекс, то функция вернет ноль, а не ошибку BADADDR, что само по себе достаточно странно, потому что по нулевому линейному адресу теоретически возможно создать точку входа, хотя это случается крайне редко, поскольку для большинства файлов адрес загрузки по умолчанию лежит значительно выше и равен 0x10000

Ординал точки входа будет необходим в дальнейшем для функций GetEntryPoint и RenameEntryPoint.



Следующий пример выдаст на экран ординалы всех существующих точек входа.

```

auto a,i;
i=0;
while((a=GetEntryOrdinal(i++)))
Message("0x%X \n",a);

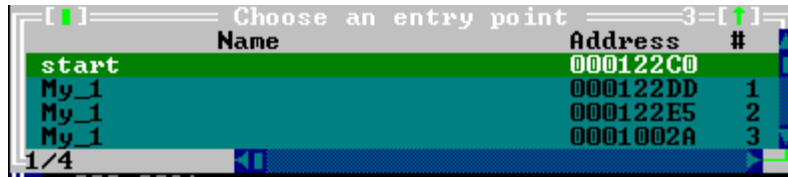
0x122C0
0x1
0x2
0x3

```

Операнд	Пояснения	
index	Индекс точки входа в списке (от нуля до GetEntryPointQty()-1)	
Return	Завершение	Пояснения
	!=0	Ординал точки входа
	0	Ошибка

### long GetEntryPoint(long ordinal)

Возвращает адрес точки входа по ординалу. Если указанный ординал не существует, возвращается ошибка BADADDR.



Следующий пример выдаст на экран адреса всех существующих точек входа.

```

auto a,i;
i=0;
while((a=GetEntryOrdinal(i++))
Message("0x%X %x \n",a, GetEntryPoint(a));

```

```

0x122C0      122c0
0x1          122dd
0x2          122e5
0x3          1002a

```

Операнд	Пояснения	
ordinal	Ординал точки входа	
Return	Завершение	Пояснения
	!=BADADDR	Адрес точки входа
	==BADADDR	Ошибка

### success RenameEntryPoint(long ordinal,char name);

Позволяет изменить имя точки входа по ординалу. При этом предыдущее имя переносится в комментарий. Если операция завершиться неуспешно, то функция вернет неравное нулю число. Такое может, случиться, например, при попытке передать в качестве нового имени пустую строку.

Пример использования:

```

seg000:22C0 start      proc near
seg000:22C0             call    My_1
seg000:22C3             call    sub_0_2325
seg000:22C6             call    sub_0_235B
seg000:22C9             call    sub_0_2374
seg000:22CC             call    sub_0_23B6
seg000:22CF             call    sub_0_23F8
seg000:22CF start      endp

```

RenameEntryPoint(0x122C0,"main");

```

seg000:22C0 main      proc near      ; start
seg000:22C0             call    My_1
seg000:22C3             call    sub_0_2325
seg000:22C6             call    sub_0_235B
seg000:22C9             call    sub_0_2374
seg000:22CC             call    sub_0_23B6
seg000:22CF             call    sub_0_23F8
seg000:22CF main      endp

```

Операнд	Пояснения	
ordinal	Ординал точки входа	
name	Новое имя функции	
Return	Завершение	Пояснения
	!=0	Успешно
	==0	Ошибка

## СТРУКТУРЫ

### ALMA MATER

Строго говоря, в языке процессора – машинном коде - нет ни типов данных, ни тем более структур, - все это привилегии языков высокого уровня. Процессор же оперирует с регистрами и ячейками памяти. Это самый низкий уровень в абстракции данных и приходится вручную разбираться с такими техническими деталями, как интерпретация знаковых битов или разрядностей ячеек.

Но ассемблер это *не* машинный код. Это первый высокоуровневый язык, придуманный человечеством, делающий огромный шаг вперед в абстракции данных. Современные ассемблеры уже трудно назвать языками низкого уровня, ибо они поддерживают макросы, средства автоматизации проектирования, сложные конструкции, типотизацию данных и даже элементы объективно ориентированного программирования!

Поддержка структур на этом фоне уже не выглядит чем-то удивительным и активно используется многими программистами. Это позволяет забыть о смещениях и оперировать одними удобочитаемыми метками.

Рассмотрим, простой пример, - фрагмент кода, который проходит по цепочке MCB блоков памяти MS-DOS.

```

CALL    .FirstMCB          ; Найти          первый MCB

gmm_while:                ; Сканируем цепочку MCB
    MOV    ES, AX          ; ES = first MCB
    CMP    Byte ptr ES:[0], 'M'; Это продолжение цепочки?
    JNZ    gmm_z           ; --> Конец/Обрыв цепочки
gmm_next:                 ; // Следующий элемент цепочки
    ADD    AX, ES:[3]      ; Размер блока
    INC    AX              ; заголовок MC
    JMP    Short gmm_while ; --> Цикл

```

Красным цветом выделены константы, которые без знания структуры MCB блока делают этот код бессмысленным. Но разве возможно удержать в голове архитектуру всех компонентов современных операционных систем с точностью до смещений?

Разумеется, нет. Да этого и не требуется, - достаточно лишь заменить их соответствующими символьными именами, которые несложно и запомнить.

В нашем примере можно поступить так:

```

MCB          struc ; (sizeof=0x10)
IsLastBlock  db ?
ParentPSPaddr dw ?
Size         dw ?
Name         db 11 dup(?)
MCB          ends

```

То есть мы определили новую структуру "MCB", и теперь для доступа к ее членам совсем не обязательно знать их смещения, от начала структуры. Это сделает за нас ассемблер!

Тогда исходный текст программы будет выглядеть так:

```

CALL    .FirstMCB          ; Найти          первый MCB

gmm_while:                ; Сканируем цепочку MCB
    MOV    ES, AX          ; ES = first MCB
    CMP    Byte ptr ES:[MCB.IsLastBlock], 'M';
                                ; Это продолжение цепочки?
                                ; --> Конец/Обрыв цепочки
    JNZ    gmm_z           ; --> Конец/Обрыв цепочки
gmm_next:                 ; // Следующий элемент цепочки
    ADD    AX, ES:[MCB.Size] ; Размер блока
    INC    AX              ; заголовок MC
    JMP    Short gmm_while ; --> Цикл

```

Не правда ли он стал понятнее? Разумеется, то же можно сказать и о дизассемблированном листинге, - чтобы не держать все смещения в памяти и ежесекундно не заглядывать в справочник, лучше определить их как члены структуры, дав им понятные символьные имена.

После IDA может показаться странным, что далеко не все дизассемблеры поддерживают структуры, а уж тем более, собственноручно определенные пользователем. В этом отношении IDA неоспоримый лидер.

Она обладает развитой поддержкой структур, которые использует для множества целей. Именно так, например, происходит обращение к локальным переменным и аргументам функций.

Да, это все члены скрытой от пользователя структуры, но программно (то есть на уровне скриптов) ни чем не отличающийся от остальных.

```

.text:00403A80 _memset      proc near
.text:00403A80
.text:00403A80
.text:00403A80 arg_0       = dword ptr 4
.text:00403A80 arg_4       = byte ptr 8
.text:00403A80 arg_8       = dword ptr 0Ch

```

Действительно, если только немного изменить синтаксис объявления локальных переменных, то он ничем не будет отличаться от структуры:

```

Memset_arg struc
Save_reg      DD ?
Arg_0         DD ?
Arg_4         DD ?
Arg_8         DD ?
Memset_arg ends

```

Таким образом, структуры перестают быть всего лишь синтаксической конструкцией целевого ассемблера, а становятся ключевым элементом архитектуры IDA, использующиеся ее ядром для облегчения доступа ко многим сгруппированным по какому-то признаку данным.

Внешне (то есть интерактивно) для работы со структурами достаточно всего лишь пары команд меню, поэтому создается ложное впечатление, что в поддержке структур ничего сложного нет.

Однако, на самом деле требуется около двух десяточков высокоуровневых функций, что бы обеспечить реализацию всех необходимых операций. Но прежде чем углубляться в описание каждой из них полезно получить представление об архитектуре структур в целом.

## Архитектура структур в IDA

Итак, что есть структура с точки зрения IDA? Это, прежде всего элемент bTree, точно как сегмент или функция.

Но в отличие от перечисленных выше, структура не связана ни с каким линейным адресом. Это самостоятельный объект, существующий вне адресного пространства дизассемблируемого файла.

В таком случае возникает вопрос, - а как же к ней может осуществляться доступ?

Приходится выбирать другую уникальную характеристику, которая бы отличала одну структуру от другой.

Можно было бы использовать имя, или любую производную от него величину, но разработчик IDA выбрал другой путь. Он связал каждую структуру с 32-разрядным целым числом, то есть **идентификатором** (сокращенно ID), который возвращался при создании структуры.

Грубо говоря, можно считать идентификатор аналогом дескриптора файла, с которым приходится сталкиваться в современных операционных системах. Различия между ними и в самом деле несущественны, хотя все же существуют – так, например, после закрытия файла, его дескриптор освобождается и может быть повторно присвоен вновь открытому файлу, а идентификаторы уникальны и никогда не присваиваются дважды, – даже если связанный с ними объект был разрушен.

Однако, идентификаторы неудобны тем, что их приходится не только хранить, но и распределять между несколькими процессами. Ведь чаще всего один скрипт (IDA, пользователь) создает структуру, с которой приходится работать совсем другому скрипту.

Точно такая проблема стояла и перед разработчикам операционной системы Zergo Way (более известной широким кругам как Windows NT). И вот и в этом случае выход был один – использовать помимо идентификаторов, **поименный** доступ к объектам.

Символьные имена в самом деле гораздо удобнее малоосмысленных 32 битных числовых значений. Однако, поддерживать два набора функций, для имен и для идентификаторов по меньшей мере неразумно.

Поэтому в IDA была введена всего лишь одна функция, которая позволяла по имени структуры установить ее идентификатор (GetStrucIdByName). И обратная ей, GetStrucName, которая по идентификатору возвращала имя.

Это позволило писать понятый код наподобие следующего:

```
DelStruc (  
  GetStrucIdByName ("struc_10")  
);
```

Небольшое замедление выполнения с лихвой окупалось его удобочитаемостью, и поэтому он стал очень популярным (именно так построены все примеры скриптов, приведенные ниже)

Однако, одно лишь это не решало всех проблем. Все равно имя структуры требовалось как-то передавать скрипту, что было не всегда осуществимо.

Поэтому был необходим механизм, обеспечивающий доступ ко всем существующим структурам. Теоретически это можно осуществить с помощью идентификаторов. Так, если проскандировать все числа от нуля до 0xFFFFFFFF, то можно обнаружить все структуры, которые присутствуют в базе и получить к ним доступ.

Но как же это будет медленно! Однако, не стоит быстро отказываться от умной идеи. Ведь можно загнать все структуры в один список, проиндексированный числами от

нуля до номера последней созданной структуры, – тогда все операции с ним не потребуют никаких накладных расходов.

И в самом деле, IDA поддерживает именно такой список. Так, например, что бы узнать идентификаторы всех существующих структур достаточно выполнить следующий бесхитрый код:

```
auto a;
a=0;
while(1)
{
Message("0x%X 0x%X \n",
        a, GetStrucId(a)
        );
a=GetNextStrucIdx(a);
if (a==-1) break;
}
```

Ключевой его фигурой является функция GetStrucid, которая возвращает идентификатор по индексу структуры.

Однако, индексы не жестко связаны с идентификаторами и использовать их для доступа к структурам можно только сразу же после получения. А точнее только на протяжении того времени, в течении которого гарантировано ни одна структура не была добавлена или удалена.

Фактически индексы были введены, что бы было можно быстро получить список структур. И ни для чего большего их использовать не рекомендуется – разве что на свой страх и риск.

При этом будьте внимательны, иначе можно совершить ошибку наподобие следующей:

```
auto a;
for(a=0;a<GetStrucQty();a++) DelStruc(GetStrucId(a));
```

С первого взгляда в этих двух строчках нет никакой ошибки и скрипт будет работать как часы, но попробуйте его запустить и произойдет нечто невразумительное.

В чем же дело? Вся причина в том, что индексы обновляются при каждом удалении структуры. То есть, удалив структуру с индексом ноль, мы не можем переходить к индексу один, так как индексы были реорганизованы, и теперь нулевому индексу соответствует другая структура, а список был сокращен на единицу.

Правильный код, как бы это ни парадоксально на первый взгляд должен выглядеть так:

```
auto a;
for(a=0;a<GetStrucQty();a++)
DelStruc(GetStrucId(0));
```

Поэтому, если вы не хотите искать подобных приключений, не используйте индексы ни для чего другого, кроме как просмотра существующих структур.

Теперь рассмотрим, как осуществляется доступ к элементам структуры. Но для начала рассмотрим все характеристики члена структуры. Как известно руководство к языкам высокого уровня – это **имя**, **тип** и **смещение** относительно начала структуры.

Однако, в отличие от языков высокого уровня ассемблер MASM использует глобальное пространство имен, а это означает, что имя каждого члена структуры уникально и не может быть дважды повторено в системе.

Это огромный недостаток, который сводит на нет все преимущества структур. Так, например, если структура MCB (смотри выше) имеет члена с именем size, то невозможно дать тоже имя никакому члену другой структуры.

Впрочем, в TASM-е это ограничение устранено. Но, к сожалению, IDA не поддерживает такого режима работы. Поэтому имя члена могло бы служить идеальным средством доступа к нему, однако, в IDA использован другой подход, который при ближайшем рассмотрении оказывается не только более удобным, но и универсальным.

Доступ к элементам структуры осуществляется по их смещением, а точнее заданием любого, принадлежащего им смещения.

Это позволяет рассматривать структуру, как непрерывный «лоскут» адресного пространства, с «объектами» - членами. Именно так, например, организован доступ к локальным переменным функций.

С точки зрения IDA каждый член структуры характеризуется не только его типом (грубо говоря, размером ячейки), но и может иметь связанные объекты, такие как имя или комментарий.

Более того, член структуры может являться не только ячейкой памяти, но и вложенной структурой!

## Методы

Функция	Описание
Long GetStrucQty(void)	Возвращает количество структур, созданных вызовом AddStrucEx
Long GetFirstStrucIdx(void);	Возвращает индекс первой структуры в списке
long GetLastStrucIdx(void);	Возвращает индекс последней структуры в списке
long GetNextStrucIdx(long index);	Возвращает следующий индекс в списке структур
long GetPrevStrucIdx(long index)	Возвращает предыдущий индекс в списке структур
long GetStrucId(long index)	Возвращает ID структуры по индексу.
long GetStrucIdByName(char name);	Возвращает идентификатор структуры по ее имени
char GetStrucName(long id)	Возвращает имя структуры по ее идентификатору
char GetStrucComment(long id, long repeatable);	Возвращает комментарии к структуре
long GetStrucSize(long id)	Возвращает размер структуры в байтах, который равен сумме размера всех ее членов
long GetMemberQty(long id);	Возвращает число членов структуры
long GetStrucNextOff(long id, long	Возвращает смещение начала очередного

offset);	элемента в структуре
long GetStrucPrevOff(long id,long offset)	Возвращает смещение начала предыдущего элемента структуры
long GetFirstMember(long id);	Возвращает смещение начала первого члена структуры
long GetLastMember(long id);	Возвращает смещение начала (не конца!) последнего члена структуры
char GetMemberName(long id,long member_offset);	Возвращает имя члена структуры
char GetMemberComment(long id,long member_offset,long repeatable);	Возвращает комментарий, связанный с членом структуры
long GetMemberSize(long id,long member_offset);	Возвращает размер члена структуры в байтах
long AddStrucEx(long index,char name,long is_union)	Создает новую структуру
long IsUnion(long id);	Возвращает единицу если тип структуры – объединение
success DelStruc(long id);	удаляет существующую структуру по ее идентификатору
long SetStrucIdx(long id,long index);	Изменяет индекс структуры
long SetStrucName(long id,char name)	Изменяет имя структуры
long SetStrucComment(long id,char comment,long repeatable)	Задает комментарий к структуре
long DelStrucMember(long id,long member_offset);	Удаляет члена структуры
long SetMemberName(long id,long member_offset,char name)	Изменяет имя члена структуры
long SetMemberType(long id,long member_offset,long flag,long typeid,long nitems)	Изменяет тип члена структуры
long SetMemberComment(long id,long member_offset,char comment,long repeatable)	Задает комментарий члена структуры



## long GetStrucQty(void);

Функция возвращает количество структур, созданных вызовом AddStrucEx. Все они отображаются IDA в списке структур, который доступен из меню ~ View \ Structures.

Структуры, обеспечивающие доступ к элементам стековых фреймов в это число не входят.

Если не создано ни одной структуры, то функция возвращает ноль.

Пример использования:

```
0000 struc_1      struc
0000 field_0      db ?
0001 field_1      db ?
0002 struc_1      ends
0002
0000 ; -----
0000
0000 struc_2      struc
0000 field_0      dw ?
0002 struc_2      ends
0002
0000 ; -----
0000
0000 struc_3      struc
0000 field_0      db ?
0001 struc_3      ends
```

```
Message ("0x%X \n",
        GetStrucQty ()
        );
```

3

	==return	Пояснения
Return	!=0	Число структур, созданных вызовами AddStrucEx
	==0	Нет ни одной структуры

## long GetFirstStrucIdx(void);

Функция возвращает индекс первой структуры в списке. Если существует хотя бы одна структура, то функция всегда возвращает ноль.

Например:

```
0000 struc_1      struc
0000 field_0      db ?
0001 field_1      db ?
0002 struc_1      ends
0002
0000 ; -----
0000
0000 struc_2      struc
0000 field_0      dw ?
0002 struc_2      ends
```

```

0002
0000 ; -----
0000
0000 struc_3      struc
0000 field_0      db ?
0001 struc_3      ends

```

```

Message("0x%X \n",
        GetFirstStrucIdx()
        );

```

0x0

Список автоматически перестраивается при операциях удаления или добавления структур, поэтому индексы не остаются постоянными. Использовать их для доступа к структуре не рекомендуется.

Например, если удалить struc\_1, а потом повторить вызов GetFirstStrucIdx, то она вновь вернет ноль, однако, теперь это индекс struc\_2, а не struc\_1.

```

0000 struc_2      struc
0000 field_0      dw ?
0002 struc_2      ends
0002
0000 ; -----
0000
0000 struc_3      struc
0000 field_0      db ?
0001 struc_3      ends

```

```

Message("0x%X \n",
        GetFirstStrucIdx()
        );

```

0x0

	==return	Пояснения
Return	==0	Индекс первой структуры в списке (всегда ноль)
	==BADADDR	Нет ни одной структуры

### long GetLastStrucIdx(void);

Функция возвращает индекс последней структуры в списке. Он всегда равен GetStrucQty() – 1. В том случае если не определено ни одной структуры, то функция возвратит ошибку BADADDR.

```

0000 struc_1      struc
0000 field_0      db ?
0001 field_1      db ?
0002 struc_1      ends
0002
0000 ; -----
0000
0000 struc_2      struc
0000 field_0      dw ?

```

```

0002 struc_2          ends
0002
0000 ; -----
0000
0000 struc_3          struc
0000 field_0          db ?
0001 struc_3          ends

```

```

Message("0x%X \n",
        GetLastStrucIdx()
        );

```

0x2

	==return	Пояснения
Return	!=BADADDR	Индекс последней структуры в списке
	==BADADDR	Нет ни одной структуры

### long GetNextStrucIdx(long index);

Функция возвращает следующий индекс в списке структур. Индекс выражается целым числом от нуля до GetStrucQty() – 1. Индексы следуют последовательно вплотную друг за другом без «пустот» Поэтому псевдокод этой функции очень прост.

```

CODE:1001D3E0          push    ebx
CODE:1001D3E1          mov     ebx, eax
CODE:1001D3E3          inc     ebx
CODE:1001D3E4          call   @get_struc_qty$qqrv
CODE:1001D3E9          cmp     ebx, eax
CODE:1001D3EB          jb     short loc_0_1001D3F2
CODE:1001D3ED          or     eax, 0FFFFFFFh
CODE:1001D3F0          pop     ebx
CODE:1001D3F1          retn

```

Или то же на языке Си:

```

long GetNextStructIdx(long index)
{
if (GetStrucQty()<a) return -1;
return ++index;
}

```

По этой причине два следующих скрипта абсолютно идентичны:

```

0000 struc_1          struc
0000 field_0          db ?
0001 field_1          db ?
0002 field_2          db ?
0003 struc_1          ends
0003
0000 ; -----
0000
0000 struc_2          struc

```

```

0000 field_0          dw ?
0002 struc_2        ends
0002
0000 ; -----
0000
0000 struc_3          struc
0000 field_0          dd ?
0004 struc_3        ends
0004

auto a;
for(a=0;a<GetStrucQty();a++)
Message("0x%X 0x%X \n",
        a,GetStrucId(a)
        );

0x0 0xFF0000F0
0x1 0xFF0000FE
0x2 0xFF000100

auto a;
a=0;
while(1)
{
Message("0x%X 0x%X \n",
        a,GetStrucId(a)
        );
a=GetNextStrucIdx(a);
if (a== -1) break;
}

0x0 0xFF0000F0
0x1 0xFF0000FE
0x2 0xFF000100

```

Какой из этих двух способов использовать дело вкуса каждого. Однако, читабельность первого примера значительно лучше, а вероятность допустить ошибку – меньше.

Операнд	Пояснения	
index	Индекс структуры в списке (от нуля до GetStrucQty()-1)	
Return	==return	Пояснения
	!=BADADDR	Индекс следующей структуры в списке
	==BADADDR	Ошибка

### long GetPrevStrucIdx(long index);

Функция возвращает предыдущий индекс в списке структур. Индекс выражается целым числом от нуля до GetStrucQty() – 1. Индексы следуют последовательно вплотную друг за другом без «пустот» Поэтому псевдокод этой функции очень прост.

```

long GetPrevStrucIdx(long index)
{

```

```

if (index<-1) return;
return -index;
}

```

По этой причине два следующих скрипта абсолютно идентичны:

```

0000 struc_1      struc
0000 field_0      db ?
0001 field_1      db ?
0002 field_2      db ?
0003 struc_1      ends
0003
0000 ; -----
0000
0000 struc_2      struc
0000 field_0      dw ?
0002 struc_2      ends
0002
0000 ; -----
0000
0000 struc_3      struc
0000 field_0      dd ?
0004 struc_3      ends
0004

```

```

auto a;
for(a=GetStrucQty();a>0;--a)
Message("0x%X 0x%X \n",
        a,GetStrucId(a)
        );

```

```

0x2 0xFF000100
0x1 0xFF0000FE
0x0 0xFF0000F0

```

```

auto a;
a=GetStrucQty()-1;
while(1)
{
Message("0x%X 0x%X \n",
        a,GetStrucId(a)
        );
a=GetPrevStrucIdx(a);
if (a==-1) break;
}

```

```

0x2 0xFF000100
0x1 0xFF0000FE
0x0 0xFF0000F0

```

Какой из этих двух способов использовать дело вкуса каждого. Однако, читабельность первого примера значительно лучше, а вероятность допустить ошибку – меньше.

Операнд	Пояснения
---------	-----------

index	Индекс структуры в списке (от нуля до GetStrucQty()-1)	
Return	==return	Пояснения
	!=BADADDR	Индекс предыдущей структуры в списке
	==BADADDR	Ошибка

### long GetStrucId(long index);

Функция возвращает ID структуры по индексу. Как уже отмечалось выше, индекс не может точно идентифицировать связанную с ним структуру, поскольку при любых операциях связанных с дополнением или удалением структур, список перестраивается, и тот же индекс уже может указывать совсем на другую структуру.

В отличие от этого, идентификатор (ID) структуры представляет собой уникальное 32-битное значение, всегда указывающие на одну и ту же структуру. Более того, даже если структура, связанная с конкретным идентификатором, была удалена, гарантируется, что тот же идентификатор не будет выдан ни одной созданной после этого структуре. Это гарантирует непротиворечивость ситуации и позволяет совместно использовать один и тот же идентификатор различным скриптам.

Пример использования:

```

0000 struc_1      struc
0000 field_0      db ?
0001 field_1      db ?
0002 field_2      db ?
0003 struc_1      ends
0003
0000 ; -----
0000
0000 struc_2      struc
0000 field_0      dw ?
0002 struc_2      ends
0002
0000 ; -----
0000
0000 struc_3      struc
0000 field_0      dd ?
0004 struc_3      ends
0004

auto a;
for(a=0;a<GetStrucQty();a++)
Message("0x%X 0x%X \n",
        a,GetStrucId(a)
        );

0x0 0xFF0000F0
0x1 0xFF0000FE
0x2 0xFF000100

```

Идентификатор, как и дескриптор, с точки зрения пользователя являются абстрактным «магическим» числом, интерпретировать которое допускается только операционной системе (в качестве которой выступает в данном случае IDA).

Операнд	Пояснения	
index	Индекс структуры в списке (от нуля до GetStrucQty()-1)	
Return	==return	Пояснения
	!=BADADDR	Идентификатор (ID) структуры
	==BADADDR	Ошибка

### long GetStrucIdx(long id);

Функция позволяет получить индекс структуры в списке по ее идентификатору (ID). Обычно такой операции не требуется, поскольку практически все функции принимают на входе именно идентификатор, а не индекс.

Операнд	Пояснения	
id	Идентификатор структуры	
Return	==return	Пояснения
	!=BADADDR	Индекс
	==BADADDR	Ошибка

### long GetStrucIdByName(char name);

Функция возвращает идентификатор структуры по ее имени. Имя структуры уникально (двух и более структур с одним и тем же именем существовать не может), поэтому неоднозначности не возникает.

Пример использования:

```

auto a,b;
a=AddStrucEx(-1,"MyNewStruc1",0);
b=GetStrucIdByName("MyNewStruc1");
Message("0x%X 0x%X \n",a,b);

0000 MyNewStruc      struc ; (sizeof=0)
0000 MyNewStruc      ends

0xFF00020A 0xFF00020A

```

Обратите внимание, что функция чувствительна к регистру, (большинство ассемблеров его игнорируют). Поэтому имена "MyStruc" и "mystruc" не считаются идентичными, что и доказывает следующий пример:

```

auto a,b;
a=AddStrucEx(-1,"MyNewStruc",0);
b=GetStrucIdByName("mynewstruc");
Message("0x%X 0x%X \n",a,b);

0000 MyNewStruc      struc ; (sizeof=0)
0000 MyNewStruc      ends

0xFF00020A 0xFFFFFFFF

```

Операнд	Пояснения	
name	Имя структуры	
Return	==return	Пояснения
	!=BADADDR	Идентификатор
	==BADADDR	Ошибка

### char GetStrucName(long id);

Функция возвращает имя структуры по ее идентификатору. Очень часто используется совместно с GetStrucId.

Например:

```

0000 MyGoodStuc      struc ; (sizeof=0x2)
0000 field_0         dw ?
0002 MyGoodStuc      ends
0002
0000 ; -----
0000
0000 MyStruc          struc ; (sizeof=0x5)
0000 field_0         dw ?
0002 field_2         dw ?
0004 field_4         db ?
0005 MyStruc          ends
0005
0000 ; -----
0000
0000 My                struc ; (sizeof=0)
0000 My                ends
0000
0000 ; -----
0000
0000 MyNewStruc       struc ; (sizeof=0)
0000 MyNewStruc       ends
0000

```

```

auto a;
for (a=0;a<GetStrucQty();a++)
    Message("%s \n",
    GetStrucName(GetStrucId(a))
    );

```

MyGoodStuc

MyStruc

My1

MyNewStruc

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	==return	Пояснения
	!=""	Имя структуры
	=="	Ошибка



## char GetStrucComment(long id,long repeatable);

Функция возвращает комментарии к структуре. В текущих версиях, включая IDA 4.0, комментарии к структурам поддерживаются лишь частично. Так, например, отсутствует возможность интерактивного комментирования функций (приходится пользоваться вызовом SetStrucComment), повторяемые комментарии поддерживаются лишь частично, что подтверждается следующим примером:

```
SetStrucComment(
    GetStrucIdByName("_msExcInfo"),
    " MyComment",1);

0000 ; MyComment
0000 _msExcInfo      struc ; (sizeof=0x8)      ; XREF: .rdata:004077E6
0000                ; .rdata:00407780r ...
0000 Id              dd ?                      ; sss
0004 Proc           dd ?                      ; offset (FFFFFFFF)
0008 _msExcInfo     ends

.rdata:004077E6      dd 1879048192            ; Id
.rdata:004077E6      dd 0                    ; Pro

    Message("%s \n",
    GetStrucComment(
        GetStrucIdByName("_msExcInfo"),
        1);

MyComment
```

Обратите внимание, что IDA не отобразила повторяемый комментарий в строке rdata:004077E6, хотя это и следовало бы.

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Repeatable	Флаг	Пояснения
	0	Неповторяемый комментарий
	1	Повторяемый комментарий
Return	Завершение	Пояснения
	!=""	Комментарий
	""	Ошибка

## long GetStrucSize(long id);

Функция возвращает размер структуры в байтах, который равен сумме размера всех ее членов. Он отображается в качестве комментария в окне просмотра структур. Допускается существование структур без единого элемента, размер которых равен нулю.

```

0000 _msExcInfo      struc ; (sizeof=0x8)
0000
0000 Id             dd ?
0004 Proc          dd ?
0008 _msExcInfo    ends

Message("0x%X \n",
        GetStrucSize(GetStrucIdByName("_msExcInfo")))
);
0x8

0000 struc_3       struc ; (sizeof=0)
0000 struc_3       ends

Message("0x%X \n",
        GetStrucSize(GetStrucIdByName("struc_3")))
);
0x0

```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	==return	Пояснения
	!=BADADDR	Размер структуры
	==BADADDR	Ошибка

### long GetMemberQty(long id);

Функция возвращает число членов структуры. Допускается существование структур без единого элемента, число членов которых равно нулю.

```

0000 _msExcept       struc ;
0000 Magic          dd ?
0004 Count         dd ?
0008 InfoPtr      dd ?
000C CountDtr     dd ?
0010 DtrPtr       dd ?
0014 _unk         dd 3 dup(?)
0020 Info         _msExcInfo 0 dup(?)
0020 _msExcept     ends

Message("0x%X \n",
        GetMemberQty(GetStrucIdByName("_msExcept")))
);
0x7

```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	

	Завершение	Пояснения
Return	!=BADADDR	Число членов структуры
	==BADADDR	Ошибка

### long GetStrucNextOff(long id,long offset);

Функция возвращает смещение начала очередного элемента в структуре. Первый элемент всегда имеет нулевое смещение (что очевидно), а последний смещение численно равно размеру структуры минус единица.

Это происходит потому, что каждую структуру замыкает «виртуальный» элемент, который не видим для всех остальных функций (в том числе и GetMemberQty). Он был введен из соображений удобства программирования, и во всех остальных случаях может не браться в расчет.

Если неверно задан идентификатор или структура не содержит ни одного члена, то обоих случаях возвращается ошибка BADADDR

Например:

```

0000 _msExcept      struc ; (sizeof=0x22)
0000
0000 Magic          dd ?
0004 Count         dd ?
0008 InfoPtr       dd ?
000C CountDtr      dd ?
0010 DtrPtr        dd ?
0014 _unk          dd 3 dup(?)
0020 Info          dw ?
0022 _msExcept     ends

```

```

auto a;
a=0;
for (;;)
{
Message("0x%X \n",a);
a=GetStrucNextOff
(GetStrucIdByName("_msExcept"),a);
if (a== -1) break;
}

```

```

0x0
0x4
0x8
0xC
0x10
0x14
0x20
0x22

```

```

0000 struc_9      struc ; (sizeof=0)
0000 struc_9     ends

```

```

Message("0x%X \n",
GetStrucNextOff(
GetStrucIdByName("struc_9")
));

```

0xFFFFFFFF

Числа, отображаемые IDA слева элементов структуры, и есть искомые смещения элементов. При этом необязательно, что бы каждому смещению соответствовал именованный элемент. Поскольку для доступа к членам структуры используются не имена, а смещения элементов, то IDA поддерживает и безыменные поля, которые могут оказаться полезными в ряде случаев.

```
0000 struc_3      struc ; (sizeof=0xd)
0000 field_0      dw ?
0002              db ? ; undefined
0003 field_3      dd ?
0007 field_7      db ?
0008              db ? ; undefined
0009              db ? ; undefined
000A              db ? ; undefined
000B field_B      db ?
000C field_C      db ?
000D struc_3      ends
```

Членом структуры могут быть, в том числе, и массивы однотипных (гомогенных) элементов.

```
0000 struc_7      struc ; (sizeof=0x15)
0000 field_0      dd ?
0004 field_4      db 16 dup(?)
0014 field_14     db ?
0015 struc_7      ends
```

В этом случае очередным смещением будет смещение следующего элемента структуры.

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	Завершение	Пояснения
	!=BADADDR	Смещение начала очередного члена структуры
	==BADADDR	Ошибка

### long GetStrucPrevOff(long id,long offset);

Функция возвращает смещение начала предыдущего элемента структуры. В остальном полностью идентична GetStrucNextOff

Смещение **конца** (не начала!) последнего элемента можно получить вызовом GetStrucPrevOff(id,-1);

Если неверно задан идентификатор или структура не содержит ни одного члена, то обоих случаях возвращается ошибка BADADDR

Например:

```
0000 _msExcept     struc ; (sizeof=0x22)
```

```

0000 Magic          dd ?
0004 Count         dd ?
0008 InfoPtr      dd ?
000C CountDtr     dd ?
0010 DtrPtr       dd ?
0014 _unk         dd 3 dup(?)
0020 Info         dw ?
0022 _msExcept    ends

auto a;
a=-1;
for (;;)
{
a=GetStrucPrevOff
(GetStrucIdByName("_msExcept"), a);
if (a==-1) break;
Message("0x%X \n", a);
}

0x22
0x20
0x14
0x10
0xC
0x8
0x4
0x0

0000 struc_9      struc ; (sizeof=0)
0000 struc_9      ends

Message("0x%X \n",
GetStrucPrevOff(
GetStrucIdByName("struc_9"))
);

0xFFFFFFFF

```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	Завершение	Пояснения
	!=BADADDR	Смещение начала предыдущего члена структуры
	==BADADDR	Ошибка

### long GetFirstMember(long id);

Функция возвращает смещение начала первого члена структуры. Это значение всегда равно нулю, за тем исключением, когда неверно задан идентификатор или структура не содержит ни одного члена. В обоих случаях возвращается ошибка BADADDR  
Например

```

0000 _msExcInfo    struc ; (sizeof=0x8)
0000 Id           dd ?

```

```

0004 Proc          dd ?
0008 _msExcInfo   ends

Message("0x%X \n",
GetFirstMember(
GetStrucIdByName("_msExcept"))
);

0x0

0000 struc_9      struc ; (sizeof=0)
0000 struc_9      ends

Message("0x%X \n",
GetFirstMember(
GetStrucIdByName("struc_9"))
);

0xFFFFFFFF

```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	Завершение	Пояснения
	!=BADADDR	Смещение начала первого члена структуры
	==BADADDR	Ошибка

### long GetLastMember(long id);

Функция возвращает смещение начала (не конца!) последнего члена структуры. Обратите внимание, что этот результат не совпадает со значением, возвращаемым GetStrucNextOff для последнего элемента!

Например:

```

0000 _msExcept     struc ; (sizeof=0x22)
0000
0000 Magic        dd ?
0004 Count       dd ?
0008 InfoPtr     dd ?
000C CountDtr    dd ?
0010 DtrPtr      dd ?
0014 _unk        dd 3 dup(?)
0020 Info        dw ?
0022 _msExcept   ends

auto a;
a=0;
for (;;)
{
Message("0x%X \n", a);
a=GetStrucNextOff
(GetStrucIdByName("_msExcept"), a);
if (a==-1) break;

```

```

}

0x0
0x4
0x8
0xC
0x10
0x14
0x20
0x22

Message ("0x%X \n",
GetLastMember (
GetStrucIdByName ("_msExcept")
);

0x20

```

Если неверно задан идентификатор или структура не содержит ни одного члена, то  
обоих случаях возвращается ошибка BADADDR

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	Завершение	Пояснения
	!=BADADDR	Смещение начала последнего члена структуры
	==BADADDR	Ошибка

### char GetMemberName(long id,long member\_offset);

Функция возвращает имя члена структуры. Для этого необходимо задать  
идентификатор (ID) структуры и смещение интересующего нас элемента от ее начала  
(member\_offset) Подробнее об этом можно почитать в описании функции GetStrucNextOff.

Пример использования:

```

0000 MyStruc          struc ; (sizeof=0x7)
0000 field_0          db ?
0001 field_1          dw ?
0003 field_3          dd ?
0007 MyStruc          ends

auto a;
for (a=0;;)
{
Message ("0x%X %s \n",
a,
GetMemberName (
GetStrucIdByName ("MyStruc"), a
);
a=GetStrucNextOff (
GetStrucIdByName ("MyStruc"), a
);
}

```

```

if (a== -1) break;
}

```

```

0x0 field_0
0x1 field_1
0x3 field_3
0x7

```

Очевидно, что код работает неправильно, и пытается вернуть на один элемент больше, чем содержит структура. Причина такого поведения заключается в том, что функция `GetStrucNextOff` возвращает смещение «виртуального» элемента, замыкающего структуру. И хотя IDA отображает его имя, как показано ниже, на самом деле виртуальный элемент не имеет никакого имени и не видим для всех остальных функций, кроме `GetStrucNextOff`, `GetPrevNextOff`

```

0007 MyStruc          ends

```

Поэтому необходимо использовать другую проверку, например очередное смещение, возвращенное `GetStrucNextOff` со смещением последнего элемента, которое можно узнать вызовом `GetLastMember`.

В результате код должен выглядеть так:

```

auto a;
for (a=0;;)
{
Message("0x%X %s \n",
        a,
        GetMemberName(
            GetStrucIdByName("MyStruc"), a)
        );

a=GetStrucNextOff(GetStrucIdByName("MyStruc"), a);

if (a>GetLastMember(GetStrucIdByName("MyStruc"))) break;
}

```

```

0x0 field_0
0x1 field_1
0x3 field_3

```

Не обязательно указывать точное смещение начала элемента. Необходимо лишь, что бы указанная величина лежала в границах интересующего нас члена структуры. IDA автоматически округлит ее до смещения начала элемента.

Этот может продемонстрировать следующий скрипт:

```

0000 MyStruc          struc
0000 field_0          dw ?
0002 field_1          dw ?
0004 field_2          dw ?
0006 MyStruc          ends

auto a;
for (a=0;;)
{

```



```

Message("0x%X %s \n",
    a,
    GetMemberName(
        GetStrucIdByName("MyStruc"),
        a+1)
    );
a=GetStrucNextOff(
    GetStrucIdByName("MyStruc"), a
    );
if (a== -1) break;
}

```

```

0x0 field_0
0x1 field_1
0x3 field_2

```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
member_offset	Смещение, лежащее в границах интересующего нас элемента	
Return	Завершение	Пояснения
	!=""	Имя члена структуры
	=="	Ошибка

**char GetMemberComment(long id, long member\_offset, long repeatable);**

Функция возвращает комментарий, связанный с членом структуры. IDA поддерживает два типа комментариев – 'regular' и 'repeatable'. Последний отличается тем, что дублируется по месту обращения к элементу структуры. Однако в случае со структурами и их членами, IDA игнорирует это требование, в чем убеждает следующий пример:

```

0000 MyStruc          struc ; (sizeof=0x4)      ; XREF: seg000:0F72r
0000 field_0         dw ?                      ; My Repeatable comment
0002 field_1         dw ?                      ;
0004 MyStruc          ends

seg000:0F72*stru_0_F72  dw 0          ; field_0 ; DATA XREF: sub_0_F56r
seg000:0F72*          dw 0          ; sub_0_2456+1Cw ...
seg000:0F72*          dw 0          ; field_1

seg000:0F56          mov     es, stru_0_F72.field_0
seg000:0F5A          assume es:nothing
seg000:0F5A          xor     bx, bx

```

Остается только надеяться, что в будущем рано или поздно такая поддержка появится.

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
member_offset	Смещение, лежащее в границах интересующего нас элемента	

	Флаг	Пояснения
Repeatable	0	Неповторяемый комментарий
	1	Повторяемый комментарий
Return	Завершение	Пояснения
	!"	Комментарий
	"	Ошибка

### long GetMemberSize(long id,long member\_offset);

Функция возвращает размер члена структуры в байтах. Для этого необходимо задать любое, принадлежащее ему смещение. Это дает возможность самостоятельно проходить список элементов, без использования GetStrucNextOff, которая отличается несколько необычным поведением (подробности можно узнать в описании этой функции)

Например:

```
0000 MyStruc          struc ; (sizeof=0x19)
0000 field_0          db ?
0001 field_1          dw ?
0003 field_3          dd ?
0007 field_7          dq ?
000F field_F          dt ?
0019 MyStruc          ends
```

```
auto a;
for (a=0;;)
{
Message ("0x%X 0x%X\n",
a,
GetMemberSize(
GetStrucIdByName("MyStruc"),
a));

a=a+GetMemberSize(
GetStrucIdByName("MyStruc"),
a);

if (a>GetLastMember(
GetStrucIdByName("MyStruc"))
) break;
}

0x0 0x1
0x1 0x2
0x3 0x4
0x7 0x8
0xF 0xA
```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
member_offset	Смещение, лежащее в границах интересующего нас элемента	
Return	Завершение	Пояснения

	!=BADADDR	Размер члена структуры в байтах
	==BADADDR	Ошибка

```

0000 MyStruc      struc ; (sizeof=0x12)
0000 field_0      db ?
0001 field_1      dw ?
0003 field_3      dd ?
0007 field_7      dq ?
000F field_F      ChldStruc ?
0012 MyStruc      ends
0012
0000
0000 ChldStruc    struc ; (sizeof=0x3)
0000 field_0      db ?
0001 field_1      dw ?
0003 ChldStruc    ends

```

```

auto a;
for (a=0;;)
{
Message ("0x%X 0x%X\n",
        a,
        GetMemberStrId(
        GetStrucIdByName("MyStruc"),
        a));

a=a+GetMemberSize(
    GetStrucIdByName("MyStruc"),
    a);

if (a>GetLastMember(
    GetStrucIdByName("MyStruc"))
) break;
}

```

```

0x0  0xFFFFFFFF
0x1  0xFFFFFFFF
0x3  0xFFFFFFFF
0x7  0xFFFFFFFF
0xF  0xFF0000FB

```

### **long GetMemberStrId(long id,long member\_offset);**

Функция возвращает ID элемента структуры, если он является структурой или BADADDR в противном случае.

То есть, IDA поддерживает вложенные структуры, и дает возможность получить идентификатор, для нисходящего разбора.

Вложение при этом может быть как угодно глубоким и вложенные структуры могут содержать ссылки друг на друга.

Например:

```

0000 struc_1      struc ; (sizeof=0x11)
0000 field_0      dw ?
0002 field_2      dw ?
0004 field_4      dw ?
0006 field_6      dw ?
0008 field_8      struc_2 0 dup(?)
0011 struc_1      ends
0011
0000 ; -----
0000
0000 struc_2      struc ; (sizeof=0x11)
0000 field_0      struc_1 ?
0011 struc_2      ends
0011

```

Можно создавать и вовсе бессмысленные комбинации, за исключением, пожалуй, структур, ссылающихся на самих себя.

На самом деле максимальная глубина вложенности равна единице! То есть IDA всего-навсего поддерживает членов типа «структура» и умеет возвращать их ID. Все остальное ложиться на плечи программиста, пишущего скрипт.

И, как нетрудно убедиться, что все эти вольности, допускаемые IDA при обращении со структурами приводят к огромным трудностям в написании действительно, корректно работающего скрипта.

Так, в приведенном выше примере, при попытке вывести полный перечень членов структуры, включая вложенные, получится бесконечный рекурсивный спуск и скрипт «зависнет»

Однако, по-видимому, все же не стоит усложнять код, а просто лишь быть внимательным в отношении вложенных структур и не допускать подобных ситуаций.

Пример использования:

```

0000 struc_2      struc ; (sizeof=0x11)
0000 field_0      struc_1 ?
0011 struc_2      ends

```

```

Message("%x \n",
GetMemberStrId(
GetStrucIdByName("struc_2"), 0x0)
);

```

ff0000f8

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
member_offset	Смещение, лежащее в границах интересующего нас элемента	
Return	Завершение	Пояснения
	!=BADADDR	Идентификатор структуры
	==BADADDR	Элемент не является структурой

## **long AddStrucEx(long index,char name,long is\_union);**

Функция позволяет создавать новую структуру. Для этого необходимо указать ее имя (которое впоследствии может быть изменено) и тип, который в дальнейшем уже не может быть изменен.

Поддерживаются следующие типы структур:

Флаг is_union	Значение
1	Структура типа «Объединение» UNION
0	Структура по умолчанию

С точки зрения IDA оба типа абсолютно идентичны и работа с один из них ничем не отличается от другого.

Однако, различия проявляются при ассемблировании листинга. При необходимости ассемблер может располагать члены структуры по удобным для него адресам, вставляя незначимые байты, например, для выравнивания (что ускоряет работу кода).

Разумеется, что во многих случаях это недопустимо и просто развалит всю программу. Для предотвращения этого и была введена поддержка типа «объединение», которая отличается от обычной структуры лишь тем, что ассемблер всегда ее оставляет в неприкосновенности и гарантирует, что смещения членов относительно друг друга всегда останутся неизменными.

```
0000 struc_3          struc
0000 field_0          dw ?
0002 field_2          db ?
0003 field_3          dw ?
0005 struc_3          ends

0000 struc_4          union
0000 field_0          dw ?
0002 field_2          db ?
0003 field_3          dw ?
0005 struc_3          ends
```

Какой тип выбрать при создании структуры должен решать сам пользователь. Чаще всего взаимное расположение элементов в структуре не критично, поскольку ассемблер автоматически вычисляет смещение каждого члена.

Но если речь идет о структуре, выражающей, скажем, заголовок файла, то в этом случае любые отклонения от жестко заданных смещений приведут к неправильной работе программы, а следовательно, для работы с ними необходимо использовать тип «объединение»

При этом структуры оба типа разделяют общее пространство имен. Другими словами невозможно создать структуру, совпадающую по имени с уже существующим объединением.

В остальном же на имена наложены точно такие ограничения, как и на метки. Необходимо помнить, что IDA различает строчечные и заглавные буквы, поэтому имена 'MyStruc' и "MYSTRUC" для нее два разных имени и могут быть присвоены двум структурами.

```
0000 MyStruc          struc ; (sizeof=0x0)
0000 MyStruc          ends
0000
0000 ; -----
0000
```

```

0000 MYSTRUC      struc ; (sizeof=0x0)
0000 MYSTRUC      ends

```

Но большинство ассемблеров не различает регистра и выдаст ошибку! Поэтому необходимо не допускать таких ситуаций.

Индекс структуры обычно устанавливают равным BADADDR – при этом IDA добавляет новую структуру в конец списка, автоматически расширяя последний.

Например:

```

0000 struc_10     struc ; (sizeof=0x0)
0000 struc_10     ends
0000
0000 ; -----
0000
0000 struc_11     struc ; (sizeof=0x0)
0000 struc_11     ends

```

**AddStrucEx(-1, "MYSTRUC", 0);**

```

0000 struc_10     struc ;
0000 struc_10     ends
0000
0000 ; -----
0000
0000 struc_11     struc ;
0000 struc_11     ends
0000
0000 ; -----
0000
0000 MYSTRUC      struc ;
0000 MYSTRUC      ends
0000

```

Но имеется так же возможность задать произвольный индекс *из уже существующих*. При этом старая структура будет **уничтожена**! Поэтому, обычно к этому способу не прибегают.

Например:

```

0000 struc_10     struc ; (sizeof=0x0)
0000 struc_10     ends
0000
0000 ; -----
0000
0000 struc_11     struc ; (sizeof=0x0)
0000 struc_11     ends

```

**AddStrucEx(0, "MY\_STRUC", 0);**

```

0000 MY_STRUC     struc ;
0000 MY_STRUC     ends
0000
0000 ; -----
0000
0000 struc_10     struc ;

```

```
0000 struc_10      ends
0000
```

При этом невозможно создать со структуру с индексом более чем на единицу превышающим индекс последнего существующего.

Например:

```
0000 struc_10      struc ; (sizeof=0x0)
0000 struc_10      ends
0000
0000 ; -----
0000
0000 struc_11      struc ; (sizeof=0x0)
0000 struc_11      ends
```

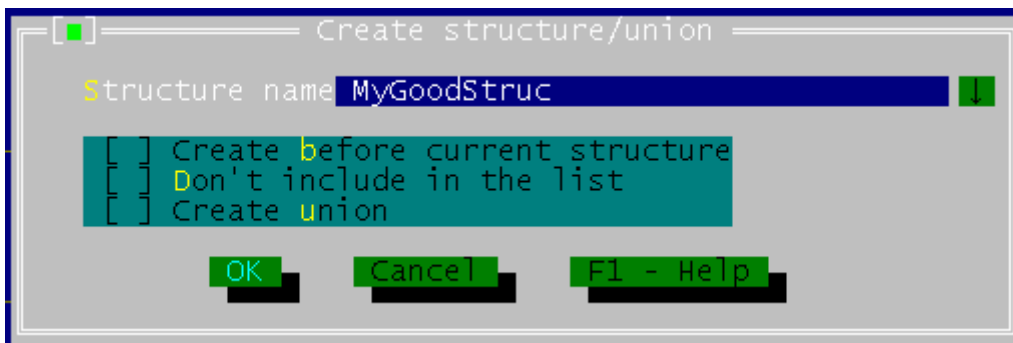
```
Message ("0x%X \n",
AddStrucEx (0x22, "MY_STRUC", 0)
);
```

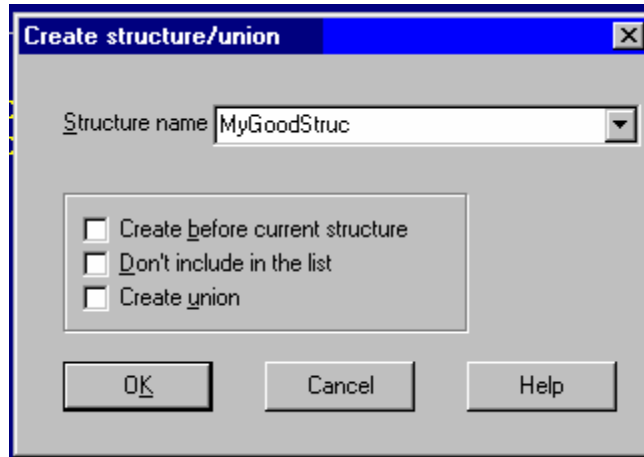
```
0xFFFFFFFF
```

Операнд	Пояснения	
index	==index	Действие
	[0,MaxIdx]	Индекс структуры (старая структура при этом будет затерта)
	MaxIdx+1	Индекс новой структуры
	BADADDR	Индекс новой структуры
name	Имя структуры	
is_union	==is_union	Значение
	1	Структура типа «Объединение» UNION
	0	Структура по умолчанию
Return	Завершение	Пояснения
	!=BADADDR	Идентификатор структуры
	==BADADDR	Элемент не является структурой

Интерактивно создать структуру можно вызвав командой меню ~ View \ Structures список всех структур и нажав <INS>

Появится следующее диалоговое окно:





### **long IsUnion(long id);**

Функция определяет тип структуры. Если она представляет собой объединение, то возвращается единица, и ноль в противном случае.

Пример использования:

```
0000 union_13      union ; (sizeof=0x0)
0000 union_13      ends
0000
```

```
Message("%x \n",
IsUnion(
  GetStrucIdByName("union_13")
)
);
```

1

```
0000 MY_STRUC      struc
0000 MY_STRUC      ends
```

```
Message("%x \n",
IsUnion(
  GetStrucIdByName("MY_STRUC")
)
);
```

0

**Ноль!** Обратите внимание, что функция в результате ошибки возвращает не BADADDR, а

Например:

```
0000 MY_STRUC      struc
0000 MY_STRUC      ends
```



```

Message ("%x \n",
IsUnion (
GetStrucIdByName ("MYSTRUCT")
)
);

0

```

Не ясно, действительно ли структура MYSTRUCT не объединение, или же ее вообще не существует.

Поэтому достоверным значением, возвращенным функцией, следует считать только единицу.

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	Завершение	Пояснения
	==1	Структура типа UNION
	==0	Структура не типа UNION или ошибка

### **success DelStruc(long id);**

Функция удаляет существующую структуру по ее идентификатору. В большинстве случаев используется совместно с GetStrucIdxByName, поскольку ID структуры скрыт от пользователя.

Например:

```

0000 MY_STRUC      struc
0000 MY_STRUC      ends
0000
0000 ; -----
0000
0000 struc_10      struc
0000 struc_10      ends
0000
0000 ; -----
0000
0000 union_13       union
0000 union_13       ends

DelStruc (
GetStrucIdByName ("struc_10")
);

0000 MY_STRUC      struc ;
0000 MY_STRUC      ends
0000
0000 ; -----
0000
0000 union_13       union ;
0000 union_13       ends
0000

```

Обратите внимание, что при этом заново перестраиваются таблицы индексов структур, поэтому полученные ранее значения уже не действительны. Их необходимо обновить заново.

Так, например:

```
0000 MY_STRUC      struc
0000 MY_STRUC      ends
0000
0000 ; -----
0000
0000 struc_10      struc
0000 struc_10      ends
0000
0000 ; -----
0000
0000 union_13       union
0000 union_13       ends

auto id1,id2;
id1= GetStrucId(0);
id2= GetStrucId(1);

DelStruc(id1);

DelStruc(id2);

0000 struc_10      struc
0000 struc_10      ends
```

Вместо того, что бы удалить первые две структуры IDA удалила первую и третью. Но эта ошибка не IDA, а разработчика скрипта!

Действительно, когда была удалена первая структура, то по индексу 1 стала теперь расположена не вторая, а третья структура!

Однако, если попытаться сделать так:

```
auto id1,id2;
id1= GetStrucId(0);

DelStruc(id1);

id2= GetStrucId(1);

DelStruc(id2);
```

То получится то же самое! Индексы были обновлены, однако, этого оказалось мало! В действительности код должен выглядеть так:

```
auto id1,id2;
id1= GetStrucId(0);

DelStruc(id1);

id2= GetStrucId(0);
```

```
DelStruc(id2);
```

Из этого примера следует, что бы не усложнять себе жизнь не стоит пользоваться индексами структур, особенно при операциях удаления.

Вместо этого лучше получить идентификаторы структуры по их имени, вызовом функции GetStrucIdByName

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

Для того, что бы интерактивно удалить функцию достаточно вызвать их список командой меню ~ View \ Structures, а затем, встав на любой элемент структуры предназначенной для удаления, нажать DEL

### long SetStrucIdx(long id,long index);

Эта функция позволяет изменить индекс структуры заданной ее идентификатором. Может использоваться для упорядочивания структур в списке.

Индекс может принимать значения от нуля до максимального индекса структуры.

При этом две структуры обмениваются местами, и затирания не происходит.

Например:

```
0000 MY_STRUC      struc ; (sizeof=0x0)
0000 MY_STRUC      ends
0000
0000 ; -----
0000
0000 union_13       union ; (sizeof=0x0)
0000 union_13       ends
0000
0000 ; -----
0000
0000 struc_11       struc ; (sizeof=0x0)
0000 struc_11       ends
0000
0000 ; -----
0000
0000 MYSTRUC        struc ; (sizeof=0x0)
0000 MYSTRUC        ends

SetStrucIdx (
GetStrucIdByName ("MY_STRUC"),
2);

0000 union_13       union ; (sizeof=0x0)
0000 union_13       ends
0000
0000 ; -----
```

```

0000
0000 struc_11      struc ; (sizeof=0x0)
0000 struc_11      ends
0000
0000 ; -----
0000
0000 MY_STRUC      struc ; (sizeof=0x0)
0000 MY_STRUC      ends
0000
0000 ; -----
0000
0000 MYSTRUC      struc ; (sizeof=0x0)
0000 MYSTRUC      ends

```

Если заданный индекс больше максимально допустимого, то считается, что был указан последний существующий индекс.

Например:

```

0000 union_13      union ; (sizeof=0x0)
0000 union_13      ends
0000
0000 ; -----
0000
0000 struc_11      struc ; (sizeof=0x0)
0000 struc_11      ends
0000
0000 ; -----
0000
0000 MY_STRUC      struc ; (sizeof=0x0)
0000 MY_STRUC      ends
0000
0000 ; -----
0000
0000 MYSTRUC      struc ; (sizeof=0x0)
0000 MYSTRUC      ends
0000

```

```

SetStrucIdx(
  GetStrucIdByName("MY_STRUC"),
  44);

```

```

0000 union_13      union ; (sizeof=0x0)
0000 union_13      ends
0000
0000 ; -----
0000
0000 struc_11      struc ; (sizeof=0x0)
0000 struc_11      ends
0000
0000 ; -----
0000
0000 MYSTRUC      struc ; (sizeof=0x0)
0000 MYSTRUC      ends
0000
0000 ; -----
0000
0000 MY_STRUC      struc ; (sizeof=0x0)

```

```
0000 MY_STRUC          ends
```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

### long SetStrucName(long id,char name);

Функция позволяет изменить имя структуры, заданной по ее идентификатору. Например:

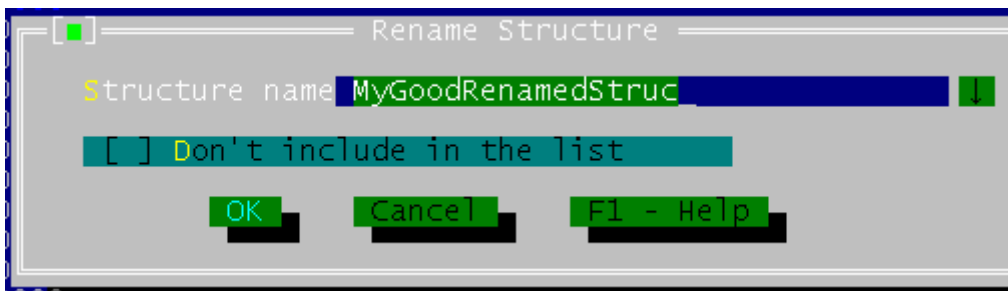
```
0000 union_13         union ;
0000 union_13         ends

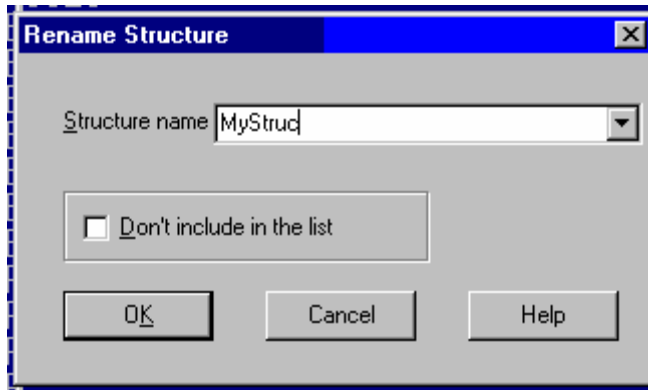
SetStrucName (
GetStrucIdByName ("union_13"),
"MyGoodRenamedStruc");

0000 MyGoodRenamedStruc union ; (sizeof=0x0)
0000 MyGoodRenamedStruc ends
```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
name	Имя структуры	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

Интерактивно переименовать функцию можно, вызвав список всех структур командой меню ~ View \ Structures, затем переместить курсор в начало выбранной структуры и нажать <N>





### **long SetStrucComment(long id,char comment,long repeatable);**

Функция возвращает комментарии к структуре. В текущих версиях, включая IDA 4.0, комментарии к структурам поддерживаются лишь частично. Так, например, отсутствует возможность интерактивного комментирования функций (приходится пользоваться вызовом SetStrucComment), повторяемые комментарии поддерживаются лишь частично, что подтверждается следующим примером:

```

SetStrucComment (
    GetStrucIdByName ("_msExcInfo"),
    " MyComment", 1);

0000 ; MyComment
0000 _msExcInfo      struc ; (sizeof=0x8)      ; XREF: .rdata:004077E6
0000                ; .rdata:00407780r ...
0000 Id              dd ?                      ; sss
0004 Proc           dd ?                      ; offset (FFFFFFFF)
0008 _msExcInfo     ends

.rdata:004077E6      dd 1879048192             ; Id
.rdata:004077E6      dd 0                    ; Pro

```

Обратите внимание, что IDA не отобразила повторяемый комментарий в строке rdata:004077E6, хотя это и следовало бы.

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
Comment	Комментарий	
Repeatable	Флаг	Пояснения
	0	Неповторяемый комментарий
	1	Повторяемый комментарий
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

**long AddStrucMember(long id,char name,long offset,long flag, long typeid,long nbytes);**

Добавляет нового члена к ранее созданной структуре, заданной ее идентификатором ID.

Членом структуры может быть ASCII строка, ячейка или другая структура, заданная ее идентификатором.

Тип добавляемого члена структуры определяется флагом flag следующим образом:

Определение	Значение	Пояснения
FF BYTE	0x00000000L	Байт
FF WORD	0x10000000L	Слово
FF DWRD	0x20000000L	Двойное слово
FF QWRD	0x30000000L	Четвертное слово
FF TBYT	0x40000000L	Восьмерное слово
FF ASCI	0x50000000L	ASCII строка
FF STRU	0x60000000L	Структура
FF RESERVED	0x70000000L	Зарезервировано
FF FLOAT	0x80000000L	Float
FF DOUBLE	0x90000000L	Double
FF PACKREAL	0xA0000000L	Packed decimal real
FF ALIGN	0xB0000000L	Директива выравнивания

В зависимости от состояния флага, значение аргумента typeid может трактоваться по-разному.

Состояние flag	Значение typeid
FF_STRU	ID структуры
FF_ASCII	Тип ASCII строки (см. таблицу ниже)
Другое	BADADDR

Обратите внимание, что если новый член структуры не представляет собой ни вложенную структуру, ни ASCII строку, то аргумент typeid должен быть равен BADADDR

Определение	Значение	Пояснения
ASCSTR_C	ASCSTR_TERMCHR	C-style ASCII string
ASCSTR_TERMCHR	0	Character-terminated ASCII string
ASCSTR_PASCAL	1	Pascal-style ASCII string (length byte)
ASCSTR_LEN2	2	Pascal-style, length has 2 bytes
ASCSTR_UNICODE	3	Unicode string
ASCSTR_LEN4	4	Pascal-style, length has 4 bytes

Таким образом, тип нового члена структуры определяется сразу двумя аргументами.

Аргумент `offset` указывает смещение элемента в структуре. Как уже было рассказано в описании предыдущих функций, доступ к членам структуры осуществляется по их смещению.

Что бы добавить новый член к структуре достаточно в качестве смещения указать `BADADDR` и тогда IDA вычислит его автоматически.

Например:

```
0000 MYSTRUC          struc
0000 field_0          db ?
0001 MYSTRUC          ends

AddStrucMember (
GetStrucIdByName ("MYSTRUC"),
"MyMember",
-1,
FF_WORD,
-1,
2);

0000 MYSTRUC          struc ; (sizeof=0x3)
0000 field_0          db ?
0001 MyMember        dw ?
0003 MYSTRUC          ends
```

Однако, то же значение можно вычислить и самостоятельно:

```
0000 MYSTRUC          struc
0000 field_0          db ?
0001 MYSTRUC          ends

AddStrucMember (
GetStrucIdByName ("MYSTRUC"),
"MyMember",
GetLastMember (
GetStrucIdByName ("MYSTRUC")
),
FF_WORD,
-1,
2);

0000 MYSTRUC          struc ; (sizeof=0x3)
0000 field_0          db ?
0001 MyMember        dw ?
0003 MYSTRUC          ends
```

Но так или иначе, при попытке указать смещение, принадлежащие уже существующему члену функция вернет ошибку.

```
0000 MYSTRUC          struc ; (sizeof=0x3)
0000 field_0          db ?
0001 MyMember        dw ?
0003 MYSTRUC          ends

Message ("0x%X \n",
AddStrucMember (
GetStrucIdByName ("MYSTRUC"),
```



```

    "MyMember2",
    3,
    FF_WORD,
    -1,
    2)
);
0000 MYSTRUC          struc ; (sizeof=0x3)
0000 field_0         db ?
0001 MyMember        dw ?
0003 MYSTRUC          ends

0xFFFFFFFF

```

Возникает вопрос – «а для чего тогда был введен аргумент смещение?» На самом деле он может быть равен не только смещению последнего элемента. Дело в том, что при удалении членов структуры, IDA не изменяет смещения остальных. Она просто преобразует удаляемые члены в неопределенные байты, на месте которых могут быть созданы новые.

Например:

```

0000 MYSTRUC          struc ; (sizeof=0x6)
0000 field_0         db ?
0001 MyMember4       db ?
0002                 db ? ; undefined
0003                 db ? ; undefined
0004                 db ? ; undefined
0005 field_5         db ?
0006 MYSTRUC          ends

```

```

AddStrucMember (
  GetStrucIdByName ("MYSTRUC"),
  "MyMember4",
  3,
  FF_WORD,
  -1,
  2)

```

```

0000 MYSTRUC          struc ; (sizeof=0x6)
0000 field_0         db ?
0001 MyMember4       db ?
0002                 db ? ; undefined
0003 MyMember2       dw ?
0005 field_5         db ?
0006 MYSTRUC          ends
0006

```

Последний аргумент nbytes задает размер нового члена структуры в байтах. С первого взгляда это бессмысленно, поскольку и так ясно из типа члена (смотри определения flag), за исключением, правда может быть ASCII строки, но и та в большинстве случаев определяется завершающим символом или типом, указанным в typeid.

На самом деле этот аргумент необходимо указывать всегда. Он должен быть кратен размеру члена структуры и если это отношение больше единицы, то IDA автоматически создает массив однотипных элементов.

Например:

```
0000 MY_STRUC      struc ; (sizeof=0x8)
0008 MY_STRUC      ends
```

```
AddStrucMember(
GetStrucIdByName("MY_STRUC"),
"MyMember_1",
-1,
FF_WORD,
-1,
8);
```

```
0000 MY_STRUC      struc ; (sizeof=0x8)
0000 MyMember_1    dw 4 dup(?)
0008 MY_STRUC      ends
```

Если попробовать указать не кратный размер, то вызов функции завершится ошибкой:

```
0000 MY_STRUC      struc ; (sizeof=0x8)
0000 MyMember_1    dw 4 dup(?)
0008 MY_STRUC      ends
```

```
Message("0x%X \n",
AddStrucMember(
GetStrucIdByName("MY_STRUC"),
"MyMember_2",
-1,
FF_WORD,
-1,
9)
);
```

```
0000 MY_STRUC      struc ; (sizeof=0x8)
0000 MyMember_1    dw 4 dup(?)
0008 MY_STRUC      ends
```

0xFFFFFFFF

Эта функция практически единственная, способная возвращать расширенный код ошибки, не только указывающий на неуспешное завершение вызова, но еще и определяющий его источник.

Все возможные коды возврата приведены в таблице ниже:

Определение		Значение
STRUC_ERROR_MEMBER_NAME	-1	Заданное имя уже существует
STRUC_ERROR_MEMBER_OFFSET	-2	Смещение принадлежит другому члену
STRUC_ERROR_MEMBER_SIZE	-3	Неверный аргумент nbyte
STRUC_ERROR_MEMBER_TINFO	-4	Неверный аргумент typeid
STRUC_ERROR_MEMBER_STRUCT	-5	Неверный id структуры
STRUC_ERROR_MEMBER_UNIVAR	-6	Объединение не может иметь членов, переменного размера
STRUC_ERROR_MEMBER_VARLAST	-7	Члены переменного размера

		ДОЛЖНЫ НАХОДИТСЯ В КОНЦЕ
--	--	--------------------------

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
name	Имя структуры	
Offset	==offset	Значение
	!=BADADDR	Смещение нового члена структуры
	==BADADDR	Добавить новый член в конец
flag	Тип нового члена	
typeid	Идентификатор структуры или тип ASCII-строки	
nbytes	Размер нового члена в байтах	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка (см коды завершения выше)

### long DelStrucMember(long id,long member\_offset);

Функция удаляет члена структуры, заданной идентификатором. Доступ к члену обеспечивается указанием любого принадлежащего ему смещения.

Однако, на самом деле IDA не удаляет члена структуры, а только преобразует его в последовательность неопределенных байтов. Поэтому, строго говоря, удалить ни какой член структуры (кроме последнего) **нельзя**, во всяком случае, так, что бы изменить смещения всех остальных (а это требуется и довольно часто!)

Например:

```
0000 MY_STRUC      struc ; (sizeof=0xb)
0000 MyMember_1   dw 4 dup(?)
0008 field_8      db ?
0009 field_9      dw ?
000B MY_STRUC     ends
```

```
DelStrucMember (
GetStrucIdByName ("MY_STRUC"),
0);
```

```
0000 MY_STRUC      struc ; (sizeof=0xb)
0000              db ? ; undefined
0001              db ? ; undefined
0002              db ? ; undefined
0003              db ? ; undefined
0004              db ? ; undefined
0005              db ? ; undefined
0006              db ? ; undefined
0007              db ? ; undefined
0008 field_8      db ?
0009 field_9      dw ?
000B MY_STRUC     ends
```

При попытке же удалить последний член структуры, он действительно удаляется без остатка.

Пример:

```
0000 MY_STRUC      struc ; (sizeof=0xb)
```

```

0000 MyMember_1      dw 4 dup(?)
0008 field_8        db ?
0009 field_9        dw ?
000B MY_STRUC       ends

DelStrucMember(
GetStrucIdByName("MY_STRUC"),
9);

0000 MY_STRUC       struc ; (sizeof=0xb)
0000 MyMember_1     dw 4 dup(?)
0008 field_8        db ?
000B MY_STRUC       ends

```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
offset	Смещение удаляемого члена структуры	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

Интерактивно то же самое можно сделать, если перевести курсор на нужный элемент структуры и нажать клавишу <U>

**long SetMemberName(long id,long member\_offset,char name);**

Функция позволяет изменять имя члена функции. Функция задается идентификатором, а член любым, принадлежащим ему смещением.

Например:

```

0000 MY_STRUC       struc ; (sizeof=0x9)
0000                db ? ; undefined
0001                db ? ; undefined
0002                db ? ; undefined
0003                db ? ; undefined
0004                db ? ; undefined
0005                db ? ; undefined
0006                db ? ; undefined
0007                db ? ; undefined
0008 field_8        db ?
0009 MY_STRUC       ends

```

```

SetMemberName(
GetStrucIdByName("MY_STRUC"),
8,"MyGoodMember");

```

```

0000 MY_STRUC       struc ; (sizeof=0x9)
0000                db ? ; undefined
0001                db ? ; undefined
0002                db ? ; undefined
0003                db ? ; undefined

```

```

0004          db ? ; undefined
0005          db ? ; undefined
0006          db ? ; undefined
0007          db ? ; undefined
0008 MyGoodMember db ?
0009 MY_STRUC   ends

```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
name	Новое имя члена структуры	
offset	Смещение удаляемого члена структуры	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

Интерактивно изменить имя можно, переместив на него курсор и нажав клавишу <N>

### **long SetMemberType(long id,long member\_offset,long flag,long typeid,long nitems);**

Функция позволяет изменять тип члена структуры. Он определяется флагом flag следующим образом:

Определение	Значение	Пояснения
FF BYTE	0x00000000L	Байт
FF WORD	0x10000000L	Слово
FF DWRD	0x20000000L	Двойное слово
FF QWRD	0x30000000L	Четвертное слово
FF TBYT	0x40000000L	Восьмерное слово
FF ASCI	0x50000000L	ASCII строка
FF STRU	0x60000000L	Структура
FF RESERVED	0x70000000L	Зарезервировано
FF FLOAT	0x80000000L	Float
FF DOUBLE	0x90000000L	Double
FF PACKREAL	0xA0000000L	Packed decimal real
FF ALIGN	0xB0000000L	Директива выравнивания

В зависимости от состояния флага, значение аргумента typeid может трактоваться по-разному.

Состояние flag	Значение typeid
FF_STRU	ID структуры
FF_ASCII	Тип ASCII строки (см. таблицу ниже)
Другое	BADADDR

Обратите внимание, что если новый член структуры не представляет собой ни вложенную структуру, ни ASCII строку, то аргумент typeid должен быть равен BADADDR

Определение	Значение	Пояснения

ASCSTR_C	ASCSTR_TERMCHR	C-style ASCII string
ASCSTR_TERMCHR	0	Character-terminated ASCII string
ASCSTR_PASCAL	1	Pascal-style ASCII string (length byte)
ASCSTR_LEN2	2	Pascal-style, length has 2 bytes
ASCSTR_UNICODE	3	Unicode string
ASCSTR_LEN4	4	Pascal-style, length has 4 bytes

При этом необходимо, чтобы для нового члена хватало места. Если его размер превосходит предыдущий, а следующие за ним смещения принадлежат остальным членам, то тип члена не будет изменен

Пример:

```

0000 MY_STRUC      struc ; (sizeof=0x9)
0000 field_0      dw ?
0002 field_2      dw ?
0004 field_4      dw ?
0006 field_6      dw ?
0008 MyGoodMember db ?
0009 MY_STRUC      ends
0009

```

```

Message("0x%X \n",
SetMemberType(
GetStrucIdByName("MY_STRUC"),
2,
FF_DWRD,
-1,
4));

```

```

0000 MY_STRUC      struc ; (sizeof=0x9)
0000 field_0      dw ?
0002 field_2      dw ?
0004 field_4      dw ?
0006 field_6      dw ?
0008 MyGoodMember db ?
0009 MY_STRUC      ends
0009

```

0x0

Напротив, если новый член занимает места меньше чем старый, то преобразование происходит без проблем, а «лишние» байты помечаются, как неопределенные.

Например:

```

0000 MY_STRUC      struc ; (sizeof=0x9)
0000 field_0      dw ?
0002 field_2      dw ?
0004 field_4      dw ?
0006 field_6      dw ?
0008 MyGoodMember db ?

```

```

0009 MY_STRUC          ends
0009

Message("0x%X \n",
SetMemberType(
GetStrucIdByName("MY_STRUC"),
2,
FF_BYTE,
-1,
1))

0000 MY_STRUC          struc ; (sizeof=0x9)
0000 field_0           dw ?
0002 field_2           db ?
0003                   db ? ; undefined
0004 field_4           dw ?
0006 field_6           dw ?
0008 MyGoodMember     db ?
0009 MY_STRUC          ends

```

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
name	Имя структуры	
Offset	==offset	Значение
	!=BADADDR	Смещение нового члена структуры
	==BADADDR	Добавить новый член в конец
flag	Новый тип члена	
typeid	Идентификатор структуры или тип ASCII-строки	
Nbytes	Размер нового члена в байтах	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка (см коды завершения выше)

### long SetMemberComment(long id,long member\_offset,char comment,long repeatable);

Функция устанавливает комментарий, связанный с членом структуры. IDA поддерживает два типа комментариев – 'regular' и 'repeatable'. Последний отличается тем, что дублируется по месту обращения к элементу структуры. Однако в случае со структурами и их членами, IDA игнорирует это требование, в чем убеждает следующий пример:

```

0000 MyStruc           struc ; (sizeof=0x4)      ; XREF: seg000:0F72r
0000 field_0           dw ?                      ; My Repeatable comment
0002 field_1           dw ?                      ;
0004 MyStruc           ends

seg000:0F72*stru_0_F72  dw 0          ; field_0 ; DATA XREF: sub_0_F56r
seg000:0F72*           dw 0          ; sub_0_2456+1Cw ...
seg000:0F72*           dw 0          ; field_1

seg000:0F56           mov     es, stru_0_F72.field_0
seg000:0F5A           assume es:nothing
seg000:0F5A           xor    bx, bx

```

Остается только надеяться, что в будущем рано или поздно такая поддержка появится.

Операнд	Пояснения	
id	Идентификатор (ID) структуры	
comment	Комментарий члена	
member_offset	Смещение, лежащее в границах интересующего нас элемента	
Repeatable	Флаг	Пояснения
	0	Неповторяемый комментарий
	1	Повторяемый комментарий
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка (см коды завершения выше)

## ПЕРЕЧИСЛЕНИЯ

### ALMA MATER

Организация перечислений очень близка к организации структур, поэтому рекомендуется ознакомиться с главой «структуры», - что бы не повторяться многие моменты при описании перечислений будут опущены, если они ничем не отличаются от описанных выше.

Прежде всего, – что же такое перечисления? Грубо говоря, – это константы, то есть предопределенные символьные значения, которые в ходе ассемблирования заменяются действительными значениями.

Использование непосредственных значений – дурной тон программирования. Как, например, на счет следующего кода:

```
PUSH 10
PUSH 02
CALL GotoXY
PUSH offset ProgramName
CALL WriteLn
```

Как нетрудно догадаться, числа 10 и 2 представляют собой экранные координаты, в которых будет выведено имя программы. Впрочем, если вы не автор этого фрагмента кода, то догадаться может быть вовсе не так просто, да и кроме того, что делать если придется переписывать программу для работы с другим экранным разрешением?

Просматривать весь код на предмет поиска всех, относящихся к экранным координатам констант?

Вот для этого в ассемблерах и появилась директива EQU, которая позволяла определить «говорящие» константы, которые не только повышали информативность листинга и заменяли комментарии. Но позволяла легко модифицировать их, – ведь теперь непосредственное значение указывалось только в одной точке.

Разумеется, IDA поддерживает константы. Но делает не так, как это можно ожидать. Если все ассемблеры поддерживают исключительно глобальные списки констант, что часто вызывает путаницу, то IDA умеет «разбивать» их на отдельные кучки – каждая под своей «крышей»

Внешне список констант напоминает структуру. Взгляните, в самом деле это очень похоже:

```
; enum enum_1
enum_1_0 = 3
enum_1_4 = 5
```



```

; -----
; enum enum_3
enum_3_14E    = 1
enum_3_0      = 2Ch
enum_3_2D     = 14Dh

```

Однако, в отличие от структуры элементы перечисления не имеют ни типа, ни размера. Точнее тип определяется только на стадии ассемблирования.

Так, например, если enum\_1\_0 равен трем, это еще не означает, что он имеет тип байт. Вполне вероятно, что он окажется словом или даже и словом и байтом одновременно, например:

```

MOV  AL, enum_1_0
CMP  AX, enum_1_0

```

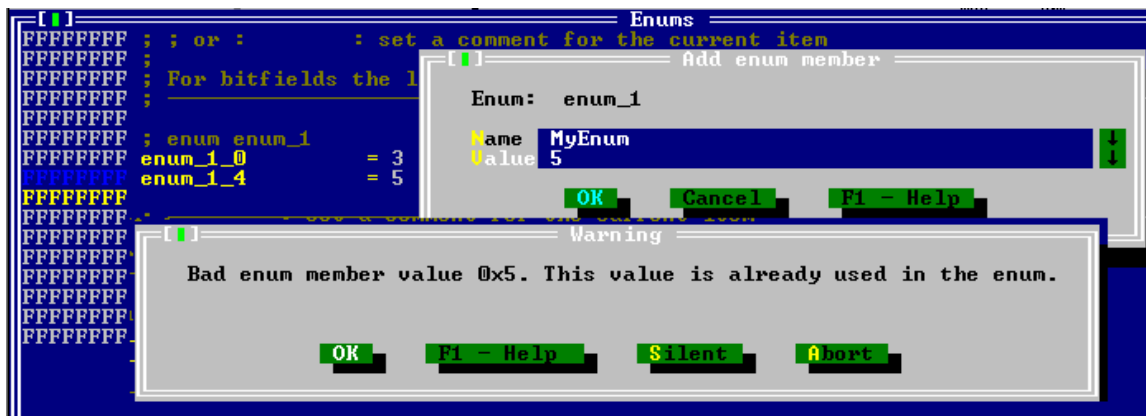
Этот код, не смотря на всю его чудакость, все же будет успешно ассемблирован!

Но если нет типов, и не возможно вычислить размер членов, то как же тогда осуществить к ним доступ?

Теоретически было можно условиться, что каждый член занимает 32 байта (двойное слово) и организовать к ним доступ точно так, как и в структурах. И это бы неплохо работало!

Но разработчик IDA пошел по другому пути - он связал каждый член с идентификатором! Разумеется, существует функция, возвращающая идентификатор по имени функции и наоборот.

В свете этого становится еще более непонятым, какой смысл имеет «группировка» перечислений. Имена членов – глобальные, идентификаторы – тем более. Что же дает принадлежность элемента к той или иной группе?



В каждой группе может существовать **не более одной** константы с одним и тем же **значением**. С первого взгляда не понятно ни как можно «жить» с этим, ни какие мотивы побудили принять разработчика такое нелепое ограничение.

Однако, на самом деле это следствия выбранной архитектуры. И весьма удачной, стоит только взглянуть на нее изнутри, чем мы сейчас и займемся.

## АРХИТЕКТУРА ПЕРЕЧИСЛЕНИЙ

Прежде чем углубляться в технические дебри реализации и архитектуры перечислений, зададимся простым вопросом, - что же по сути представляют собой члены перечислений?

Разумеется, это операнды, или еще точнее иная форма представления непосредственных операндов. В главе, посвященной объективной модели IDA уже отмечалось, что один и тот же операнд может быть по-разному отображен на экране дизассемблера. Он может быть не только непосредственным значением, но и смещением, например.

Однако, перечисления – это не просто иная форма отображения операнда на экране – с точки зрения IDA это элемент bTree, который может ссылаться на линейный адрес, объекта... впрочем, не стоит повторяться, об этом уже писалось выше.

Но если каждый сегмент (имя, комментарий, функция) связан только с одним линейным адресом, то **одно и то же** перечисление может повторяться в десятке разных мест! И поэтому старые методы для него не подходят!

Поэтому был использован тот же механизм, который был создан для поддержки структур. Каждый объект ссылался на **тег** структуры, а операнд указывал на требуемый элемент внутри ее.

Точно то же происходит и с перечислениями. Есть список перечислений, на который ссылается объект. Элементы списка просматриваются до тех пор, пока не найдется элемент совпадающий по значению с операндом, объекта.

Обратите внимание еще раз на тот факт, что и структура и перечисление связываются не с операндом, а с обладающим им объектом, а точнее линейным адресом его начала.

Представление операнда в виде члена структуры или перечисления происходит на втором этапе, – и жесткой связки тут нет, простой поиск на совпадение значений.

Но если в структуре смещение каждого члена уникально, то есть никакие два члена не могут быть расположены по одному и тому же смещению, то в перечислениях два разных элемента **могут** иметь одно и то же значение.

Вот, собственно и ответ на вопрос о необходимости поддержки более чем одного списка перечислений, а заодно и тактика группировки элементов. То есть главным критерием должно быть не родственность каких-то признаков, а гарантия непопадающих значений.

При этом разумно стремиться к уменьшению числа списков, поскольку, как уже говорилось выше, для представления операнда в виде перечисления достаточно сослаться на список, и IDA самостоятельно подберет нужный элемент!

В идеале, если у нас всего один список (что бывает достаточно часто) необходимо перевести курсор на нужную строку и нажать <T>, как IDA все сделает автоматически.

Программная работа, в отличие от интерактивной, несколько сложнее. Кроме того, теги списков (это не теги, конечно, но много названия просто нет, - поэтому будет считать, что это как бы теги) вообще практически не фигурируют.

Действительно, все члены связаны с уникальными глобальными идентификаторами, да и имена каждого из них не менее уникальны.

## МЕТОДЫ

Функция	Назначение
long GetEnumQty(void)	Возвращает число типов перечислений
long GetnEnum(long idx)	Возвращает идентификатор перечисления по ее индексу
long GetEnumIdx(long enum_id);	Возвращает индекс перечисления по его

	идентификатору
long GetEnum(char name)	Возвращает идентификатор перечисления по его имени
char GetEnumName(long enum_id)	Возвращает имя перечисления по его идентификатору
char GetEnumCmt(long enum_id,long repeatable)	Возвращает комментарий перечисления
long GetEnumSize(long enum_id)	Возвращает число членов перечисления
long GetEnumFlag(long enum_id)	Возвращает флаги, определяющие представление элементов перечисления
long GetConstByName(char name)	Возвращает идентификатор константы по ее имени
long GetConstValue(long const_id)	Возвращает значение константы по ее идентификатору
char GetConstName(long const_id)	Возвращает имя константы по ее идентификатору
char GetConstName(long const_id)	Возвращает комментарий константы по ее идентификатору
long AddEnum(long idx,char name,long flag)	Добавляет новое перечисление
void DelEnum(long enum_id)	Удаляет перечисление
success SetEnumIdx(long enum_id,long idx)	Задает индекс перечисления в списке

### long GetEnumQty(void);

Функция возвращает число типов перечислений. Все они могут быть отображены вызовом списка командой меню ~ View \ Enumeration's

```

FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF enum_2_0      = 16h
FFFFFFFF

```

```

Message("0x%X \n",
GetEnumQty()
);

```

0x2

	==return	Пояснения
Return	!=0	Число перечислений
	==0	Нет ни одного перечисления

### long GetnEnum(long idx);

Функция возвращает ID перечисления по индексу. Как уже отмечалось выше, индекс не может точно идентифицировать связанное с ним перечисление, поскольку при любых операциях связанных с дополнением или удалением перечислений, список перестраивается, и тот же индекс уже может указывать совсем на другое перечисление.

В отличие от этого, идентификатор (ID) перечисления представляет собой уникальное 32-битное значение, всегда указывающие на одно и ту же перечисление. Более того, даже если перечисление, связанное с конкретным идентификатором, было удалено, гарантируется, что тот же идентификатор не будет выдан ни одному созданному после этого перечислению.

Это гарантирует непротиворечивость ситуации и позволяет совместно использовать один и тот же идентификатор различным скриптам.

Пример использования:

```

FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF enum_2_0      = 16h

auto a;
for(a=0;a<GetEnumQty();a++)
Message("0x%X 0x%X \n",
        a,GetEnumId(a)
);

```

```

0x0 0xFF0000F0
0x1 0xFF0000FE

```

Идентификатор, как и дескриптор, с точки зрения пользователя являются абстрактным «магическим» числом, интерпретировать которое допускается только операционной системе (в качестве которой выступает в данном случае IDA).

Операнд	Пояснения	
index	Индекс перечисления в списке (от нуля до GetEnumQty()-1)	
Return	==return	Пояснения

	!=BADADDR	Идентификатор (ID) перечисления
	==BADADDR	Ошибка

### long GetEnumIdx(long enum\_id);

Функция возвращает индекс перечисления по ее идентификатору. Необходимо помнить, что индексы не связаны жестко с перечислениями и при каждой операции удаления или добавления новых перечислений тем же индексам уже могут соответствовать новые перечисления.

Пример использования:

```

FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF enum_2_0      = 16h

Message("0x%X \n",
GetEnumIdx(
GetEnum("enum_1")
)
);

0x0

Message("0x%X \n",
GetEnumIdx(
GetEnum("enum_2")
)
);

0x1

```

Операнд	Пояснения	
ID	Идентификатор перечисления	
Return	==return	Пояснения
	!=BADADDR	Индекс перечисления
	==BADADDR	Ошибка

### long GetEnum(char name);

Функция возвращает идентификатор перечисления по его имени. Если перечисления с указанным именем не существует, то функция возвращает ошибку – BADADDR.

Пример использования:

```

FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2

```

```

FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF enum_2_0      = 16h

Message("0x%X \n",
GetEnum("enum_1")
);

0xFF000131

Message("0x%X \n",
GetEnum("enum_2")
);

0xFF000132

Message("0x%X \n",
GetEnum("enum_3")
);

0xFFFFFFFF

```

Операнд	Пояснения	
name	Имя перечисления	
Return	==return	Пояснения
	!=BADADDR	Идентификатор перечисления
	==BADADDR	Ошибка

### char GetEnumName(long enum\_id);

Функция возвращает имя перечисления по его идентификатору. Если указанному идентификатору не соответствует ни одно перечисление функция возвращает пустую строку.

Например:

```

FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF enum_2_0      = 16h

Message("%s \n",
GetEnumName(
GetnEnum(1)
)
);

enum_2

```

Операнд	Пояснения	
Enum_id	ID перечисления	
Return	==return	Пояснения
	!="	Имя перечисления
	=="	Ошибка

### char GetEnumCmt(long enum\_id,long repeatable)

Возвращает комментарий перечисления, заданного идентификатором. Комментарии бывают двух типов – постоянные и повторяемые. Постоянные отображаются только впереди перечисления, а повторяемые при обращении к каждому из его членов.

```

FFFFFFFF ; My Enum regulag commnet
FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2

seg000:0046                rol     bx, enum_1_0

Message("%s \n",
GetEnumCmt(
GetEnum("enum_1"),
0);

My Enum regulag commnet

FFFFFFFF ; My Enum repeatable commnet
FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2

seg000:0046                rol     bx, enum_1_0    ; My Enum
Repeatable commnet

Message("%s \n",
GetEnumCmt(
GetEnum("enum_1"),
1);

My Enum Repeatable commnet

```

Операнд	Пояснения	
id	Идентификатор (ID) перечисления	
Repeatable	Флаг	Пояснения
	0	Неповторяемый комментарий
	1	Повторяемый комментарий
Return	Завершение	Пояснения
	!="	Комментарий
	""	Ошибка

## long GetEnumSize(long enum\_id);

Функция возвращает число членов перечисления, заданного идентификатором. Обратите внимание, именно число элементов, а не занимаемый ими размер, который вообще вычислить невозможно, поскольку члены перечисления не имеют типа.

Пример:

```
FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
```

```
Message("0x%X \n",
GetEnumSize(
GetEnum("enum_1")
)
);
```

0x2

Если перечисление пусто, то функция возвращает ноль, но то же значение возвращается, если указать неверный идентификатор, поэтому возникает неоднозначная ситуация – либо перечисление отсутствует (было удалено?) либо же попросту пусто.

Пример:

```
FFFFFFFF ; enum enum_2
```

```
Message("0x%X \n",
GetEnumSize(
GetEnum("enum_2")
)
);
```

0x0

```
Message("0x%X \n",
GetEnumSize(BADADDR)
);
```

0x0

Операнд	Пояснения	
Enum_id	Идентификатор перечисления	
Return	==return	Пояснения
	!=0	Число членов перечисления
	==0	Пустое перечисление
		Ошибка



## long GetEnumFlag(long enum\_id);

Функция возвращает флаги, определяющие представление членов перечисления, заданного идентификатором.

Возможные значения перечислены ниже в таблице:

FF_0NUMH	0x00100000	шестнадцатеричное представление первого операнда
FF_0NUMD	0x00200000	десятичное представление первого операнда
FF_0CHAR	0x00300000	символьное представление первого операнда
FF_0SEG	0x00400000	первый операнд – сегмент
FF_0OFF	0x00500000	первый операнд – смещение
FF_0NUMB	0x00600000	Представление первого операнда в бинарном виде
FF_0NUMO	0x00700000	Представление первого операнда в восьмеричном виде
FF_0ENUM	0x00800000	Представление первого операнда в виде перечисления
FF_0FOP	0x00900000	Принудительный первый операнд
FF_0STRO	0x00A00000	Представление первого операнда как смещения в структуре
FF_0STK	0x00B00000	первый операнд стековая переменная
FF_1VOID	0x00000000	тип второго операнда Void
FF_1NUMH	0x00100000	Шестнадцатеричное представление второго операнда
FF_1NUMD	0x00200000	десятичное представление второго операнда
FF_1CHAR	0x00300000	символьное представление второго операнда
FF_1SEG	0x00400000	второй операнд – сегмент
FF_1OFF	0x00500000	второй операнд – смещение
FF_1NUMB	0x00600000	Представление второго операнда в бинарном виде
FF_1NUMO	0x00700000	Представление второго операнда в восьмеричном виде
FF_1ENUM	0x00800000	Представление второго операнда в виде перечисления
FF_1FOP	0x00900000	Принудительный второй операнд
FF_1STRO	0x00A00000	Представление второго операнда как смещения в структуре
FF_1STK	0x00B00000	второй операнд стековая переменная

Пример:

```
FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
```

```
Message("0x%X \n",
GetEnumFlag(
GetEnum("enum_1")
);
```

0x1100000

Операнд	Пояснения	
Enum_id	Идентификатор перечисления	
Return	==return	Пояснения
	!=0	Флаг отображения членов перечисления
	==0	Ошибка

## long GetConstByName(char name);

Функция возвращает идентификатор константы по ее имени. Все перечисления разделяют общее пространство имен, другими словами одно и то же имя не может быть повторено дважды, поэтому является уникальным.

Например:

```
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF MyEnum        = 16h
```

```
Message ("0x%X \n",
GetConstByName ("MyEnum")
);
```

```
0xFF000136
```

Идентификатор обеспечивает доступ к константе. Что бы, например, получить ее значение необходимо воспользоваться функцией long GetConstValue(long const\_id), которая описана ниже.

Операнд	Пояснения	
name	Имя константы	
Return	==return	Пояснения
	!=0	Идентификатор константы
	==0	Ошибка

## long GetConstValue(long const\_id);

Функция возвращает значение константы по ее идентификатору или ноль в результате ошибки. Поэтому часто возникает неопределенность, – то ли действительно имела место ошибка (например, был указан несуществующий идентификатор) или же просто константа имеет такое значение.

Пример использования:

```
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF MyEnum        = 16h
```

```
Message ("0x%X \n",
GetConstValue (
GetConstByName ("MyEnum")
)
);
```

```
0x16
```

Операнд	Пояснения	
Const_id	Идентификатор константы	
Return	==return	Пояснения
	!=0	Значение константы
	==0	Ошибка
		Значение константы

### char GetConstName(long const\_id);

Функция возвращает имя константы, заданной идентификатором. Если идентификатор указан неправильно, то возвращается пустая строка  
Например:

```

FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF MyEnum        = 16h
FFFFFFFF

Message("%s \n",
GetConstName (
GetConstByName ("MyEnum")
)
);

MyEnum

```

Операнд	Пояснения	
Const_id	ID константы	
Return	==return	Пояснения
	!=""	Имя константы
	=="	Ошибка

### char GetConstCmt(long const\_id,long repeatable);

Возвращает комментарий константы, заданной идентификатором. Комментарии бывают двух типов – постоянные и повторяемые. Постоянные отображаются только справа от константы, а повторяемые при каждом обращении к ней.

```

FFFFFFFF ;
FFFFFFFF ; FFFFFFFFF enum_1_0      = 1 ; My regulag commnet
FFFFFFFF enum_1_2      = 2

seg000:0046                rol     bx, enum_1_0

Message("%s \n",
GetConstCmt (

```

```

GetConstByName("enum_1_0"),
0);

My regulag commnet

FFFFFFFF
FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0      = 1 ; My Enum repeatable commnet
FFFFFFFF enum_1_2      = 2

seg000:0046          rol    bx, enum_1_0    ; My
Repeatable commnet

Message("%s \n",
GetConstCmt(
GetConstByName("enum_1_0"),
1);

My Repeatable commnet

```

Операнд	Пояснения	
id	Идентификатор (ID) константы	
Repeatable	Флаг	Пояснения
	0	Неповторяемый комментарий
	1	Повторяемый комментарий
Return	Завершение	Пояснения
	!=""	Комментарий
	""	Ошибка

### long AddEnum(long idx,char name,long flag);

Функция добавляет новое перечисление. Для этого необходимо указать его имя (которое впоследствии может быть изменено) и тип представления констант в перечислении.

Индекс задает положение перечисления в списке. Если он равен BADADDR, то новое перечисление будет добавлено в конец списка, иначе же старое перечисление будет затерто! Подробнее об этом рассказано в описании функции AddStrucEx

Флаг определяет представление констант в перечислении. Может принимать значения, перечисленные ниже в таблице:

FF_0NUMH	0x00100000	шестнадцатеричное представление первого операнда
FF_0NUMD	0x00200000	десятичное представление первого операнда
FF_0CHAR	0x00300000	символьное представление первого операнда
FF_0SEG	0x00400000	первый операнд – сегмент
FF_0OFF	0x00500000	первый операнд – смещение
FF_0NUMB	0x00600000	Представление первого операнда в бинарном виде
FF_0NUMO	0x00700000	Представление первого операнда в восьмеричном виде
FF_0ENUM	0x00800000	Представление первого операнда в виде перечисления

FF_0FOP	0x00900000	Принудительный первый операнд
FF_0STRO	0x00A00000	Представление первого операнда как смещения в структуре
FF_0STK	0x00B00000	первый операнд стековая переменная
FF_1VOID	0x00000000	тип второго операнда Void
FF_1NUMH	0x00100000	Шестнадцатеричное представление второго операнда
FF_1NUMD	0x00200000	десятичное представление второго операнда
FF_1CHAR	0x00300000	символьное представление второго операнда
FF_1SEG	0x00400000	второй операнд – сегмент
FF_1OFF	0x00500000	второй операнд – смещение
FF_1NUMB	0x00600000	Представление второго операнда в бинарном виде
FF_1NUMO	0x00700000	Представление второго операнда в восьмеричном виде
FF_1ENUM	0x00800000	Представление второго операнда в виде перечисления
FF_1FOP	0x00900000	Принудительный второй операнд
FF_1STRO	0x00A00000	Представление второго операнда как смещения в структуре
FF_1STK	0x00B00000	второй операнд стековая переменная

Пример использования:

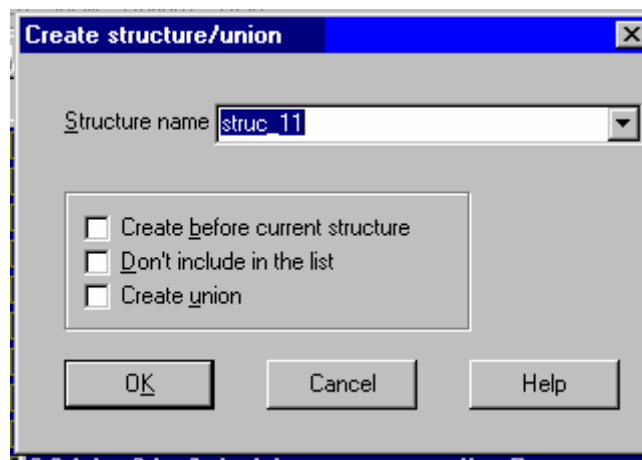
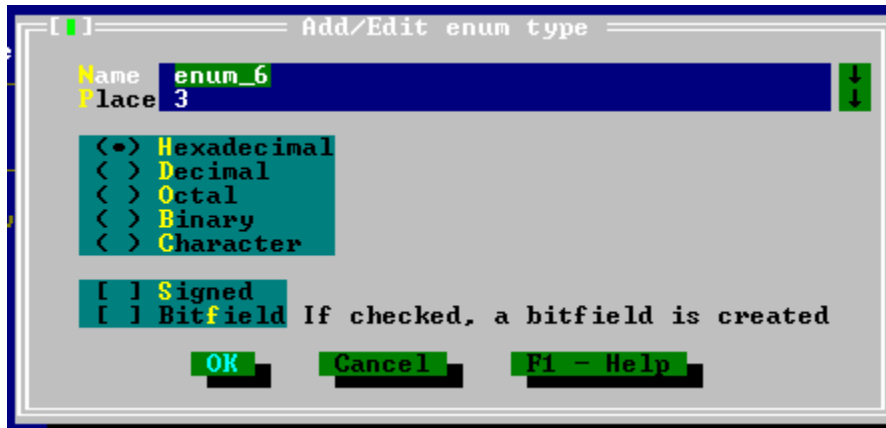
```
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
```

```
AddEnum(-1, "MyNewEnum", 0);
```

```
FFFFFFFF enum_1_0      = 1
FFFFFFFF enum_1_2      = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum MyNewEnum
```

Операнд	Пояснения	
index	==index	Действие
	[0,MaxIdx]	Индекс перечисления (старое перечисление при этом будет затерто)
	MaxIdx+1	Индекс нового перечисления
	BADADDR	Индекс нового перечисления
name	Имя перечисления	
Return	Завершение	Пояснения
	!=BADADDR	Идентификатор перечисления
	==BADADDR	Ошибка

Интерактивно структуру добавить можно, вызвав список командой меню ~ View \ Structures и нажав клавишу <INS>



**void DelEnum(long enum\_id);**

Функция удаляет перечисление, заданное идентификатором вместе со всеми его членами.

Пример:

```

FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0    = 1
FFFFFFFF enum_1_2    = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_2
FFFFFFFF MyEnum      = 16h
FFFFFFFF

DelEnum(
  GetEnum("enum_2")
);

FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0    = 1

```

```
FFFFFFFF enum_1_2    = 2
FFFFFFFF
```

Операнд	Пояснения
Enum_id	Идентификатор перечисления

Интерактивно перечисление можно удалить, установив курсор на любой его элемент и нажав клавишу <DEL>

### success SetEnumIdx(long enum\_id,long idx);

Функция позволяет изменять индекс перечисления в списке. При этом перечисления меняются местами, и затирания не происходит.

Например:

```
FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0    = 1
FFFFFFFF enum_1_2    = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum MyNewEnum
FFFFFFFF MyNewEnum_0 = 0
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_9
FFFFFFFF enum_9_0    = 0

SetEnumIdx(
GetEnum("enum_1"),1
);

FFFFFFFF ; enum MyNewEnum
FFFFFFFF MyNewEnum_0 = 0
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_1
FFFFFFFF enum_1_0    = 1
FFFFFFFF enum_1_2    = 2
FFFFFFFF
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum enum_9
FFFFFFFF enum_9_0    = 0
FFFFFFFF
```

Операнд	Пояснения
id	Идентификатор (ID) перечисления
Return	==return Пояснения

	==1	Успешное завершение
	==0	Ошибка

## FIXUP

### ALMA MATER

Более привычным синонимом fixup вероятно, окажется термин «перемещаемые элементы».

Что это такое? Как можно судить из названия, это относительные адреса, обеспечивающие портатбельность кода, то есть независимость от базового адреса загрузки.

Поскольку IDA эмулирует загрузку файла в собственное адресное пространство и даже трассирует его выполнение, то она должна поддерживать и перемещаемые элементы.

Покажем это на небольшом примере, MS-DOS exe файла.

Просмотр с помощью HIEW

```
00000200: B80100    mov ax,00001
00000203: 8ED8     mov ds,ax
00000205: B409     mov ah,009 ;"
00000207: BA0000   mov dx,00000
0000020A: CD21     int 021
```

Просмотр с помощью IDA

```
seg000:0000 start proc near
seg000:0000    mov ax, 1001h
seg000:0003    mov ds, ax
seg000:0005    assume ds:dseg
seg000:0005    mov ah, 9
seg000:0007    mov dx, 0
seg000:000A    int 21h
```

Что грузиться в регистр AX? С виду непосредственное значение. Однако, это не так. Если приглядеться, то можно увидеть, что дальше оно помещается в регистр DS и, следовательно, скорее всего указывает на сегмент данных программы.

То есть 0x1 это адрес сегмента данных выраженный в параграфах. Знакомым с архитектурой IBM PC может показаться, с чего бы это вдруг он казался расположенным глубоко в таблице векторов прерываний. Но ничего странного нет. И сегмент данных расположен вовсе не там, где можно было бы подумать.

Ведь это **относительный** адрес. Резанный вопрос относительный **чего?** В exe файлах от отсчитывается от адреса загрузки первого байта, расположенного за заголовком файла.

По умолчанию IDA загружает exe файлы по адресу 0x10000. Следовательно, считая в параграфах, 0x1000+0x1 == 0x1001, как видим, результат совпадает с тем, что отобразила IDA.

Вот это и понимается под поддержкой перемещаемых элементов. Остается только ответить на вопрос, откуда IDA узнала, как следует трактовать этот непосредственный операнд? Эвристическим анализатором? Нет, конечно. Она поступила точно так же, как и операционная система на ее месте, - просто прочитала заголовок файла.

```
00000000: 4D 5A 2C 00-02 00 02 00-20 00 00 00-FF FF 00 00 MZ, □ □  _
```



```

00000010: 00 00 00 00-00 00 00 00-3E 00 00 00-01 00 FB 71 > □ _q
00000020: 6A 72 00 00-00 00 00 00-00 00 00 00-00 00 00 00 jr
00000030: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 01 00 □
00000040: 00 00 0D 00-00 00 00 00-00 00 00 00-00 00 00 00

```

Файлы типа OLD EXE имеют очень простую структуру, реализующую поддержку перемещаемых элементов. В таблице, состоящей из двойных слов, перечисляются линейные адреса, указывающие на перемещаемые элементы, - то есть относительные адреса, каждый из которых представляет собой слово, ссылающееся на сегмент (адрес в параграфах).

Таким образом, IDA остается только выполнить простое арифметическое сложение. Впрочем, существуют и более запутанные форматы. Так, например, все win32 файлы, поддерживающие динамическое связывание, или говоря проще, возможность вызова функций чужих DLL вынуждены иметь похожие структуры, - ведь адреса вызываемых функций не известны на этапе линковки программы и могут быть вычислены только после загрузки файла в память.

Если же обратиться к другим платформам, а не замыкаться на серии Intel 80x86, то мы столкнемся с феейверком самых разных технических решений от которого может быстро зарядить в глазах.

Каким же образом IDA может поддерживать все это одним махом? Ведь перемещаемыми элементами управляют меньше десяти функций.

Действительно, если абстрагироваться от вариаций технических реализаций и сосредоточиться на природе перемещаемых элементов, то можно с удивлением обнаружить, что в ее основы могут быть сформулированы всего одной фразой.

*Вот этот операнд ссылается **туда**.* И все! Этого достаточно, что бы обеспечить нормальную функциональность и работоспособность!

Какой конкретно элемент и куда ссылается, вычисляют специальные модули, отвечающие за загрузку файла определенного формата. Ядро IDA такие проблемы не волнуют.

Поэтому фактически, поддержка перемещаемых элементов обеспечивается не IDA, а внешними модулями, к которым у пользователя программного доступа из языка скриптов **нет**.

Другими словами, перемещаемыми элементами пользователь не управляет. Да, конечно, он может посмотреть все существующие перемещаемые элементы и даже их скорректировать, но нужно ли это?

Большинство, использующих IDA, этой возможности ни когда в своей жизни не использовали (ну, может быть, разве что из любопытства). И это правильно.

Правильно, потому что IDA и сама неплохо справляется с поставленной задачей. Поддержка перемещаемых элементов это не та область, что требует внимания со стороны пользователя.

Впрочем, из этого правила все же есть исключения. Если некоторая ну очень хитрая программа имеет свой загрузчик (или чаще - интерпретируемый код), который работает не стандартно и стало быть IDA его не поддерживает.

Немного поразмыслив, сюда же можно отнести и некоторые случаи самомодифицирующегося кода. Ведь поддержка перемещаемых элементов лежит на самом низком уровне иерархии IDA - другими словами «исправляется» виртуальная память, поэтому это работает не только с операндами, но и с кодом, то есть влияет на дизассемблирование инструкций.

Основная трудность описания перемещаемых элементов в объяснении их типов. А они то же разными бывают. Причем большая часть существует только на других платформах и совершенно чужда пользователям PC с Windows и Pentium. А поэтому описывать их подробно в этой книге совершенно бессмысленно.

Для этого планируется выпустить специально приложение, рассказывающие об особенностях использования IDA на других платформах.

Поэтому ниже на типе перемещаемых элементах внимание акцентироваться не будет.

## МЕТОДЫ

Функция	Назначение
long GetNextFixupEA(long ea)	Возвращает линейный адрес следующего перемещаемого элемента
long GetPrevFixupEA(long ea)	Возвращает линейный адрес предыдущего перемещаемого элемента
long GetFixupTgtType(long ea)	Тип перемещаемого элемента
long GetFixupTgtSel(long ea)	Возвращает селектор перемещаемого элемента
long GetFixupTgtOff(long ea)	Возвращает смещение перемещаемого элемента
void SetFixup(long ea,long type,long targetsel,long targetoff,long displ)	Добавляет новый перемещаемый элемент
void DelFixup(long ea)	Функция удаляет перемещаемый элемент

### long GetNextFixupEA(long ea);

Возвращает линейный адрес следующего перемещаемого элемента. Обратите внимание, что эта функция действительно возвращает адрес перемещаемого элемента, а не адрес начала содержащей его инструкции.

Например:

```
dseg:0000 public start
dseg:0000 start          proc near
dseg:0000 B8 00 10      mov     ax, seg dseg
dseg:0003 8E D8        mov     ds, ax
```

```
Message ("0x%X \n",
        GetNextFixupEA(0)
        );
```

0x1001

Эмулятор загрузки инициализировал перемещаемый элемент необходимым значением, указывающим на адрес сегмента в виртуальной памяти. В нашем случае он равен 0x10.

Операнд	Пояснения	
ea	Линейный адрес	
Return	Завершение	Пояснения
	!=BADADDR	Успешно
	==BADADDR	Ошибка

### long GetPrevFixupEA(long ea);

Возвращает предыдущий адрес следующего перемещаемого элемента. Обратите внимание, что эта функция действительно возвращает адрес перемещаемого элемента, а не адрес начала содержащей его инструкции.

Например:

```
dseg:0000 public start
dseg:0000 start          proc near
dseg:0000 B8 00 10      mov     ax, seg dseg
dseg:0003 8E D8        mov     ds, ax
```

```
Message ("0x%X \n",
        GetNextFixupEA(-1)
    );
```

0x1001

Эмулятор загрузки инициализировал перемещаемый элемент необходимым значением, указывающим на адрес сегмента в виртуальной памяти. В нашем случае он равен 0x10.

Операнд	Пояснения	
ea	Линейный адрес	
Return	Завершение	Пояснения
	!=BADADDR	Успешно
	==BADADDR	Ошибка

### long GetFixupTgtType(long ea);

Функция возвращает тип перемещаемого элемента по его линейному адресу. Возможные значения перечислены в таблице ниже. Поскольку большинство из них на платформе Intel не имеет места, то подробное описание их назначения приведено в факультативном приложении к книге «Использование IDA на не-Intel платформах»

FIXUP_MASK	0xF	
FIXUP_BYTE	FIXUP_OFF8	Восьми битное смещение
FIXUP_OFF8	0	
FIXUP_OFF16	1	16-битное смещение
FIXUP_SEG16	2	16-битный сегмент (селектор)
FIXUP_PTR32	3	32-битный длинный указатель (16-бит база; 16-бит селектор)
FIXUP_OFF32	4	32-битное смещение
FIXUP_PTR48	5	48-битный указатель (16-бит база; 32-бит смещение).
FIXUP_HI8	6	Старшие 8 бит 16-битного смещения
FIXUP_HI16	7	Старшие 16 бит 32-битного смещения
FIXUP_LOW8	8	Младшие 8 бит 16-битного смещения
FIXUP_LOW16	9	Младшие 16бит 32-битного смещения
FIXUP_REL	0x10	fixup is relative to the linear address specified in the 3d parameter to set fixup()

FIXUP_SELFREL	0x0	elf-relative? - disallows the kernel to convert operands in the first pass- this fixup is used during output This type of fixups is not used anymore. Anyway you can use it for commenting purpose in the loader modules
FIXUP_EXTDEF	0x20	target is a location (otherwise - segment)
FIXUP_UNUSED	0x40	fixup is ignored by IDA disallows the kernel to convert operands- this fixup is not used during output
FIXUP_CREATED	0x80	fixup was not present in the input file

Пример:

```
seg000:032D          cmp     word ptr [bp+8], seg seg000
seg000:0332          jnz    loc_0_33A
```

```
Message("0x%X \n",
GetFixupTgtType(
GetNextFixupEA(0)
)
);
```

0x2

Операнд	Пояснения	
ea	Линейный адрес	
Return	Завершение	Пояснения
	!=BADADDR	Тип перемещаемого элемента
	==BADADDR	Ошибка

### long GetFixupTgtSel(long ea);

Функция возвращает селектор перемещаемого элемента, заданного линейным адресом.

Пример использования:

```
C0000000 ; Segment type: Pure code
C0000000 LCOD      segment para public 'CODE' use32
C0000000          assume cs:LCOD
C0000000          ;org 0C0000000h
C0000000          assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
C0000000 Service_Table_0 dd offset unkserve_0 ; DATA XREF: LCOD:C0000040
```

```
Message("0x%X, 0x%X \n",
GetNextFixupEA(0),
GetFixupTgtSel(GetNextFixupEA(0)
)
);
```

0xC0000000, 0x2

Операнд	Пояснения	
ea	Линейный адрес	
Return	Завершение	Пояснения
	!=BADADDR	Селектор перемещаемого элемента
	==BADADDR	Ошибка

### long GetFixupTgtOff(long ea);

Функция возвращает смещение перемещаемого элемента, заданного линейным адресом.

Пример использования:

```

C0000000 ; Segment type: Pure code
C0000000 LCOD      segment para public 'CODE' use32
C0000000          assume cs:LCOD
C0000000          ;org 0C0000000h
C0000000          assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
C0000000 Service_Table_0 dd offset unkserve_0 ; DATA XREF: LCOD:C0000040□

```

```

Message("0x%X, 0x%X \n",
GetNextFixupEA(0),
GetFixupTgtOff(GetNextFixupEA(0))
)
);

```

0xC0000000, 0xC0003514

Операнд	Пояснения	
ea	Линейный адрес	
Return	Завершение	Пояснения
	!=BADADDR	Смещение перемещаемого элемента
	==BADADDR	Ошибка

### long GetFixupTgtDispl(long ea);

Функция возвращает displacement перемещаемого элемента, заданного линейным адресом. В настоящее время на платформе Intel практически не используется и подробнее описана в факультативном приложении к книге «Использование IDA на не-Intel платформах»

Операнд	Пояснения	
ea	Линейный адрес	
Return	Завершение	Пояснения
	!=BADADDR	Displacement перемещаемого элемента
	==BADADDR	Ошибка

### void SetFixup(long ea,long type,long targetsetl,long targetoff,long displ);

Функция позволяет управлять перемещаемым элементом. Практически никогда не используется, так как IDA сама разбирается в большинстве существующих форматах файлов и правильно задает значение перемещаемых элементов.

Однако, может быть полезна при написании своего загрузчика файла. Тогда необходимо для каждого перемещаемого элемента вызвать эту функцию, указав его линейный адрес. Обратите внимание, не адрес начала манипулирующей с ним инструкции.

Тип перемещаемого элемента может принимать одно из следующих значений:

FIXUP_MASK	0xF	
FIXUP_BYTE	FIXUP_OFF8	Восьми битное смещение
FIXUP_OFF8	0	
FIXUP_OFF16	1	16-битное смещение
FIXUP_SEG16	2	16-битный сегмент (селектор)
FIXUP_PTR32	3	32-битный длинный указатель (16-бит база; 16-бит селектор)
FIXUP_OFF32	4	32-битное смещение
FIXUP_PTR48	5	48-битный указатель (16-бит база; 32-бит смещение).
FIXUP_HI8	6	Старшие 8 бит 16-битного смещения
FIXUP_HI16	7	Старшие 16 бит 32-битного смещения
FIXUP_LOW8	8	Младшие 8 бит 16-битного смещения
FIXUP_LOW16	9	Младшие 16бит 32-битного смещения
FIXUP_REL	0x10	fixup is relative to the linear address specified in the 3d parameter to set fixup()
FIXUP_SELFRE L	0x0	elf-relative? - disallows the kernel to convert operands in the first pass- this fixup is used during output This type of fixups is not used anymore. Anyway you can use it for commenting purpose in the loader modules
FIXUP_EXTDEF	0x20	target is a location (otherwise - segment)
FIXUP_UNUSED	0x40	fixup is ignored by IDA disallows the kernel to convert operands- this fixup is not used during output
FIXUP_CREATE D	0x80	fixup was not present in the input file

Следующие два аргумента задают селектор (сегмент) и смещение объекта, на который ссылается перемещаемый элемент.

Например:

```

seg000:0000 start          proc near
seg000:0000                mov     ax, seg dseg
seg000:0003                mov     ds, ax
seg000:0005                assume ds:dseg
seg000:0005                mov     ah, 9
seg000:0007                mov     dx, 0Ch
seg000:000A                int     21h                ; DOS - PRINT
STRING
seg000:000A                ; DS:DX ->
string terminated by "$"
seg000:000C                mov     ax, seg dseg
seg000:000F                mov     ds, ax
seg000:0011                mov     ah, 9

```

```

seg000:0013          mov     dx, 0Ch
seg000:0016          int     21h          ; DOS - PRINT
STRING
seg000:0016          ; DS:DX ->
string terminated by "$"
seg000:0018          mov     ah, 4Ch
seg000:001A          int     21h          ; DOS - 2+ -
QUIT WITH EXIT CODE (EXIT)
seg000:001A start    endp          ; AL = exit
code
seg000:001A
seg000:001A seg000   ends
seg000:001A
dseg:000C ; -----
-----
dseg:000C
dseg:000C ; Segment type: Pure data
dseg:000C dseg      segment para public 'DATA' use16
dseg:000C          assume cs:dseg
dseg:000C          ;org 0Ch
dseg:000C aHelloSailor db 'Hello,Sailor!',0Dh,0Ah,'$'
dseg:000C dseg      ends

```

В приведенном выше коде перемещаемый элемент был создан вызовом:

```
SetFixup(0x10001, FIXUP_SEG16, 0, 0x1001, 0);
```

Операнд	Пояснения
ea	Линейный адрес
Type	Тип перемещаемого элемента
sel	Селектор
off	Смещение
Displ	Displacement

**void DelFixup(long ea);**

Функция удаляет перемещаемый элемент.

Операнд	Пояснения
ea	Линейный адрес

## АНАЛИЗ

Имя функции	Назначение
long FindText (long ea,long flag,long y,long x,char str);	Ищет фрагмент дизассемблируемого текста
Прочие функции	
Имя функции	Назначение
Void DeleteAll	Удаляет все элементы и связанные с ними элементы.

Long AnalyseArea (long sEA,long eEA)	Дизассемблирует выбранный регион
void AutoMark (long ea,long queueType);	Управляет автоанализом
void AutoMark2 (long start,long end,long queueType)	Управляет автоанализом

### void DeleteAll ();

Невероятно “полезная” функция, удаляющая всю информацию о дизассемблируемой программе - сегменты, метки, комментарии, словом **все** флаги и связанные с ними объекты.

Как вариант частичного отката результата использования этой функции – немедленный аварийный выход из IDA без сохранения последних изменений.

Пример использования:

```

seg000:0000 ;
seg000:0000 ; File Name   : F:\IDAF\IDA\test.exe
seg000:0000 ; Format       : MS-DOS executable (EXE)
seg000:0000 ; Base Address: 1000h Range: 10000h-132EAh Loaded length:
32EAh
seg000:0000 ; Entry Point : 1000:22C0
seg000:0000
seg000:0000
seg000:0000 ; -----
seg000:0000
seg000:0000 ; Segment type: Pure code
seg000:0000 seg000      segment byte public 'CODE' use16
seg000:0000                assume cs:seg000
seg000:0000                assume es:nothing, ss:nothing, ds:nothing,
fs:nothing, gs:nothing
seg000:0000
seg000:0000; _____ S U B R O U T I N E _____
seg000:0000
seg000:0000
seg000:0000 MyFuncnt      proc near          ; CODE XREF: sub_0_22DD+1Ep
seg000:0000                push     ax                ; My Comment
seg000:0001                push     bx
seg000:0002                push     cx
seg000:0003                push     dx
seg000:0004
seg000:0004 MyLabel:
seg000:0004                mov     ax, 3D02h
seg000:0007                mov     dx, 206h
seg000:000A                int     21h

```

### DeleteAll ();

```

0:00010000                db     50h ; P
0:00010001                db     53h ; S
0:00010002                db     51h ; Q
0:00010003                db     52h ; R

```



```

0:00010004          db 0B8h ; +
0:00010005          db  2 ;
0:00010006          db  3Dh ; =
0:00010007          db 0BAh ; |
0:00010008          db  6 ;
0:00010009          db  2 ;
0:0001000A          db 0CDh ; -

```

### long AnalyseArea (long sEA,long eEA);

Полный анализ выбранной области. Весь код будет дизассемблирован независимо от того, были обнаружены на него ссылки или нет.

Сравните:

```

seg000:0100 start          db 0BBh ; +
seg000:0101              db  5 ;
seg000:0102              db  1 ;
seg000:0103              db 0FFh ;
seg000:0104              db 0E3h ; y
seg000:0105              db 0B4h ; |
seg000:0106              db  6 ;
seg000:0107              db 0B2h ; _
seg000:0108              db  7 ;
seg000:0109              db 0CDh ; -
seg000:010A              db 21h ; !
seg000:010B              db 0C3h ; +
seg000:010B seg000      ends

```

MakeCode (0x10100);

```

seg000:0100 start:
seg000:0100              mov     bx, 105h
seg000:0103              jmp     bx
seg000:0103 ; -----
seg000:0105              db 0B4h ; |
seg000:0106              db  6 ;
seg000:0107              db 0B2h ; _
seg000:0108              db  7 ;
seg000:0109              db 0CDh ; -
seg000:010A              db 21h ; !
seg000:010B              db 0C3h ; +
seg000:010B seg000      ends

```

AnalyseArea (0x10100,0x10B);

```

seg000:0100 start:
seg000:0100              mov     bx, 105h
seg000:0103              jmp     bx
seg000:0105 ; -----
seg000:0105              mov     ah, 6
seg000:0107              mov     dl, 7
seg000:0109              int     21h
seg000:010B              retn
seg000:010B seg000      ends

```

Аналогично авто анализу функция может выполняться в фоновом режиме. Поэтому, в скриптах эта функция используется совместно с Wait(). К сожалению никаких других, более развитых средств синхронизации не предусмотрено.

Задаваемые границы анализируемой области не обязательно должны существовать в природе. В крайнем случае, будет проанализирован регион от наименьшего до наибольшего существующих адресов.

AnalyseArea(0,0x30000) успешно выполнится, даже если наименьшим из существующих окажется адрес '0x10000'.

'AnalyseArea(0,BADADDR-1);' выполнится успешно и для удобства может быть определено как макрос 'AnalyseAll' и размещено в файле 'ida.idc'

При нормальном завершении функция возвращает '1' и '0' если анализ был прерван пользователем по нажатию 'Ctrl-Break'.

Операнд	пояснения	
sEA	линейный адрес начала анализируемой области	
eEA	линейный адрес конца анализируемой области	
Return	==return	пояснения
	==1	Анализ был успешно завершен
	==0	Анализ был прерван, нажатием Ctrl-Break

**void AutoMark (long ea,long queuetype);**  
**void AutoMark2 (long start,long end,long queuetype);**

Функции, позволяющие непосредственно управлять автоанализом. Пользователь может явно указать, как будет трактоваться тот или иной регион. Допустим, нам известно, что в приведенном ниже примере по адресу 0xE расположены данные, а не код.

Однако IDA не может «догадаться» до этого и дизассемблирует код не так, как мы этого ожидаем.

Разумеется, можно прибегнуть к ручному анализу, но это будет слишком утомительно. Гораздо проще использовать следующие команды:

```
seg000:0004 unk_0_4      db 0B8h
seg000:0005                db  2
seg000:0006                db 3Dh
seg000:0007                db 0BAh
seg000:0008                db  0
seg000:0009                db 32h
seg000:000A                db 0CDh
seg000:000B                db 21h
seg000:000C                db 73h
seg000:000D                db  9
seg000:000E                db 0B4h
seg000:000F                db 3Ch
```

```
AutoMark(0x1000E,AU_UNK);
MakeCode(0x10004);
```

```
seg000:0004 loc_0_4:
seg000:0004                mov     ax, 3D02h
seg000:0007                mov     dx, 3200h
seg000:000A                int     21h
seg000:000C                jnb    near ptr unk_0_1
seg000:000C ; -----
```

```

seg000:000E          db  0B4h ; !
seg000:000F          db  3Ch  ; <

```

Как видно, «MakeCode» дошла до адреса 0xE и остановилась. Это очень удобный способ ограничить диапазон ее действия (конечного адреса MakeCode не имеет).

Кроме этого с помощью AutoMark можно создавать функции, применять библиотеки сигнатур и так далее. Действие определяется выбором параметра **queuetype**. Для него определены следующие константы:

определение	действие
AU_UNK	не исследовать указанную область
AU_CODE	преобразовать указанную область в код
AU_PROC	создать функцию по указанному адресу
AU_USED	реанализ
AU_LIBF	применить сигнатуру FLIRT
AU_FINAL	свернуть все неисследованные области

Для одного и того же адреса допустимо задавать более одного указания. IDA помещает все запросы в очередь, поэтому «затирания» не происходит. После анализа запросы удаляются из очереди, поэтому при реанализе следует их задать повторно.

AutoMark отличается от AutoMark2 тем, что последняя позволяет явно указать область действия запроса, тогда как первая определяет ее автоматически.

Обращаться с этой функцией следует осторожно. Если тип запроса не будет соответствовать определенным константам, то IDA выдаст сообщение об ошибке и аварийно выйдет в операционную систему без сохранения результата работы.

Операнд	пояснения
'ea'	линейный адрес
queuetype	Тип запроса (смотри таблицу выше)

### **long FindText (long ea,long flag,long y,long x,char str);**

Эта очень полезная и мощная функция для глобального поиска подстроки во всем дизассемблируемом тексте. Сюда входят не только строковые выражения, но и символическое представление инструкций, имен, комментариев, меток и перекрестных ссылок. Словом, равносильно тому, как если бы мы вывели результат работы в дизассемблера в LST файл - отчета и искали бы в нем требуют подстроку.

Заметим, что чаще все же так и поступают потому что то FindText работает достаточно медленно и не поддерживает символы-джокеры, как, например hiew. Очень часто критерии поиска настолько сложны, что не могут быть реализованы через 'FindText'. Поэтому приходится прибегать к созданию хитрых скриптов для весьма изощренного поиска.

Однако, в ряде случаев 'FindText' все же хватает для повседневных задач и легче использовать несколько вызовов этой функции с разными параметрами, чем прибегать к нештатным средствам.

Младший бит флага задает направление поиска. Если он установлен, то поиск будет идти от младших адресов к старшим и наоборот.

Первый, считая от нуля, бит флагов указывает на чувствительность функции к регистру. Если он установлен, то заглавные и строчечные символы будут различаться.

Координаты 'x' и 'y' применимы только к многострочным комментариям (ExtLinA\ExtLinB) в остальных же случаях они игнорируются и могут быть равны нулю.

'srt' задает подстроку поиска.

```

seg000:005A 88 04      mov     [si], al
seg000:005C 46          inc     si
seg000:005D E2 E7      loop   loc_0_46
seg000:005F BE EC 01    mov     si, 1ECh
seg000:0062 E8 78 00    call   sub_0_DD

```

```

Message ("%x \n",
FindText (0x1005A, 1, 0, 0, "loop")
);

```

1005D

В случае ошибки поиска функция возвращает константу BADADDR.

Операнд	пояснения	
ea	Линейный адрес	
flag	==flag	Направление поиска
	1	Поиск «вперед»
	0	Поиск «назад»
Return	==return	Пояснения
	!=BADADDR	Линейный адрес найденного текстового вхождения
	==BADADDR	Ошибка

char Demangle(char name, long disable\_mask)

Функция «размангляет» переданное ей имя **name** в соответствии с заданными настройками **disable\_mask** (см. таблицу).

Если функция не может разманглить имя, она возвращает пустую строку. IDA Pro поддерживает спецификации Watcom, Microsoft и Borland. Перечни символов используемых для замангления имен содержатся в поле "*MangleChars*" конфигурационного файла *<ida.cfg>*

```

MangleChars = "$:?( [.] )" // watcom
               "@$%?" // microsoft
               "@$%"; // borland

```

Пример использования:

```

Message (">%s\n", Demangle ("??1streambuf@@UAE@XZ", MNG_DEFNONE));
а) вызов функции Demangle для «размангления» имени
"??1streambuf@@UAE@XZ"

```

```

>public: virtual __thiscall streambuf::~streambuf(void)
b) результат

```

флаг	#	пояснения
MNG_NOPTRTYPE	0x00000001	не показывать ни <b>far</b> , ни <b>near</b> , ни <b>huge</b> модификаторы
MNG_DEFNEAR	0x00000000	не показывать <b>near</b> модификатор
MNG_DEFFAR	0x00000002	не показывать <b>far</b> модификатор
MNG_DEFHUGE	0x00000004	не показывать <b>huge</b> модификатор
MNG_DEFNONE	0x00000006	показывать модификаторы <b>near</b> , <b>far</b> , <b>huge</b> (если установлен MNG_NOPTRTYPE. модификаторы не

		будут отображаться)
MNG_NODEFINIT	0x00000008	не показывать ничего, кроме главного имени
MNG_NOUNDERSCORE	0x00000010	не показывать символы прочерка
MNG_NOTYPE	0x00000020	не выполнять преобразование типов передаваемых параметров и базового класса
MNG_NORETTYPE	0x00000040	не показывать тип значения, возвращаемого функцией
MNG_NOBASEDT	0x00000080	не показывать базовый тип
MNG_NOCALLC	0x00000100	нигде не преобразовывать типы
MNG_NOSCCTYP	0x00000400	не показывать ключевые слова <b>public</b> , <b>private</b> , <b>protect</b>
MNG_NOTHROW	0x00000800	не показывать описатель <b>throw</b>
MNG_NOSTVIR	0x00001000	не показывать ключевые слова <b>static</b> и <b>virtual</b>
MNG_NOECSU	0x00002000	не показывать ключевые слова <b>class</b> , <b>struct</b> , <b>union</b> , <b>enum</b>
MNG_NOCSVOL	0x00004000	не показывать ключевые слова <b>const</b> и <b>volatile</b>
MNG_NOCLOSUR	0x00008000	не показывать ключевого слова <b>__closure</b>
MNG_SHORT_S	0x00010000	заменять <b>signed int</b> на <b>sint</b>
MNG_SHORT_U	0x00020000	заменять <b>unsigned int</b> на <b>uint</b>
MNG_ZPT_SPACE	0x00040000	не показывать пробелы после запятых
MNG_IGN_ANYWAY	0x00080000	игнорировать суффикс <b>_nn</b> в конце имен
MNG_IGN_JMP	0x00100000	игнорировать префикс <b>j_</b> в начале имен
MNG_MOVE_JMP	0x00200000	переносить префикс <b>j_</b> и в замангленные имена

Это две предварительно определенные сокращенные и полные формы записи. Для просмотра и модификации активируйте пункт меню (~Options\ Demangled names...)

??? #верстальщику – change table

аргумент	пояснения	
name	замангленное имя	
disable_mask	маска (смотри таблицу ???)	
return	=return	пояснения
	!=	размангленное имя
	==	ошибка

## ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ

### ALMA MATER

Изначально IDA проектировалась, как интерактивная среда, то есть тесно взаимодействующая с пользователем.

Однако, для скриптов большинство интерфейсных функций не доступно. Нельзя, например, сконструировать диалог или создать свой пункт меню. В распоряжении пользователя оказывается набор функций, обеспечивающий примитивный ввод – вывод. То есть простейшие диалоговое окна запроса параметров и вывода результатов своей деятельности на экран.

Впрочем, этого в большинстве случаев оказывается достаточно, потому что большинство скриптов предназначено для работы в автономном режиме.

Но иногда все же требуется узнать позицию курсора на экране, или наоборот, перевести его на определенное место, что бы привлечь внимание пользователя, сообщить о результатах своей работы и так далее.

Вот для этого и существует набор специальных функций, ютящихся под одной крышей, только потому, что они попали под критерий «взаимодействие с пользователем». В отличие от всех, описанных выше, они не относятся к какому-то определенному объекту и не понятно какой частью архитектуры IDA они являются.

Но... они есть, и следовательно, будут тщательно рассмотрены и описаны. Для облегчения понимания введем некоторую дополнительную классификацию, хотя она, конечно, будет весьма условна.

Итак, одна группа функций взаимодействует с курсором на экране. Что есть курсор с точки зрения IDA? Это указатель текущей строки, которая связана с некоторым объектом, точнее с линейным адресом его начала.

То есть при работе с курсором IDA не рассматривает его экранные координаты, а только линейный адрес памяти, на который этот курсор указывает.

Часто бывает так, что несколько строк расположены по одному и тому же линейному адресу.

Например:

```
.text:00401020
.text:00401020 ; _____ S U B R O U T I N E
_____
.text:00401020
.text:00401020 ; Attributes: library function bp-based frame
.text:00401020
.text:00401020 public start
.text:00401020 start proc near
.text:00401020
.text:00401020 var_20 = dword ptr -20h
.text:00401020 var_1C = dword ptr -1Ch
.text:00401020 var_18 = dword ptr -18h
.text:00401020 var_14 = dword ptr -14h
.text:00401020 var_4 = dword ptr -4
.text:00401020
.text:00401020 push ebp
```

Все эти строки совершенно идентичны с точки зрения IDA, поэтому в которой бы из них не находился курсор, при попытке определить его положение, всегда вернется адрес 0x401020, что в общем-то неудивительно.

Но вот далеко не однозначно, на какую строку переместится курсор при попытке изменить его положение. Оказывается, что на первую в которой присутствует инструкция или команда.

Впрочем, это относится к тем тонкостям реализации, сохранность которых в последующих версиях не гарантируется. Но, с другой стороны, скорее всего не будет изменяться, поскольку это решение выглядит достаточно логичным.

Другая группа функций работает с выделением, то есть отмеченным блоком на экране. Собственно это наиболее популярный способ передачи скрипту входных данных для работы, а точнее диапазона адресов.

Этим заведуют всего две функции, - SelStart и SelEnd. К сожалению, выделение программно доступно «Только на чтение» и выделить регион самостоятельно скрипт не может.

Теперь перейдем к функциям, управляющим экранным вводом – выводом. Ввод данных обеспечивает едва ли не десяток специализированных функций, создающих диалоговые окна и проверяющие корректность ввода пользователя.

Однако, в силу больших условностей и множества оговорок, лучше пользоваться только низкоуровневыми функциями, обеспечивающих ввод строкового или длинного целого значений и проверять их корректность самостоятельно.

Вывод данных в основном направляется в окно сообщений, и за редким исключением в всплывающие диалоговые окна. Все необходимые функции подробнее будут описаны ниже.

## МЕТОДЫ

Функция	Назначение
char AskStr (char defval, char prompt)	Запрашивает у пользователя строку
char AskFile (long forsave, char mask, char prompt)	Создает диалоговое окно для выбора имени файла
long AskAddr (long defval, char prompt)	Запрашивает у пользователя адрес
long AskLong (long defval, char prompt)	Запрашивает у пользователя число типа long
long AskSeg (long defval, char prompt)	Запрашивает сегмент у пользователя
char AskIdent (char defval, char prompt);	Запрашивает у пользователя ввод имени идентификатора
long AskYN (long defval, char prompt)	Создает модальный диалог Yes \ No \ Cancel
void Message (char format,...);	Выводит строку в окно сообщений
void Warning (char format,...)	Функция выводит предупреждающий диалог
void Fatal (char format,...)	Выводит фатальный диалог
long ScreenEA ();	Возвращает линейный адрес строки, на которой стоит курсор
long SelStart ();	Возвращает линейный адрес начала выделенной области
long SelEnd ();	Возвращает линейный адрес конца выделенной области
success Jump (long ea)	Изменяет позицию курсора в окне дизассемблера
void Wait ();	Функция ожидает конца автоанализа
long AddHotkey(char hotkey, char	Добавляет новую горячую клавишу

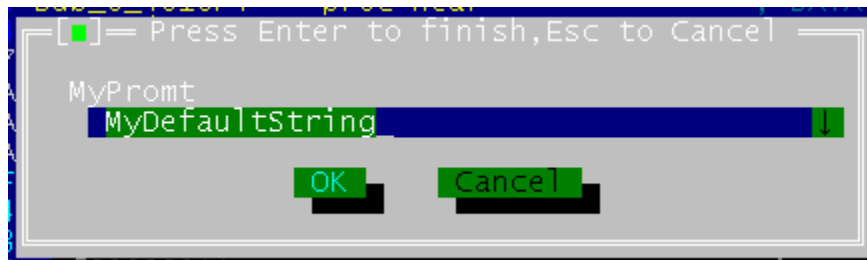
idcfunc);	
success DelHotkey(char hotkey);	Удаляет горячую клавишу

**char AskStr (char defval,char prompt);**

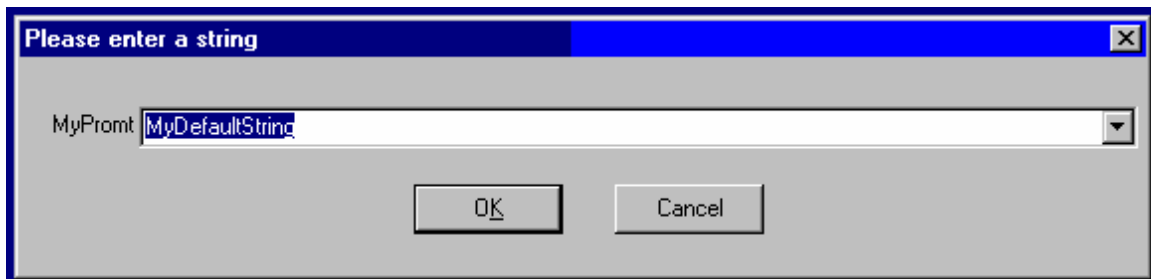
Функция создает и выводит модальный диалог ввода строки. Используется в скриптах, тесно взаимодействующими с пользователем.

В консольной версии окно будет выглядеть следующим образом:

`AskStr("MyDefaultString","MyPromt");`



А в графической версии приглашения ввода выглядит так:



операнд	Пояснения	
defval	Значение по умолчанию	
promt	Пояснение, которое будет выведено в диалоговом окне	
Return	Завершение	Пояснения
	!=""	Строка
	""	Ошибка

Если пользователь откажется от ввода и нажмет <Esc> или Cancel, то функция возвратит *пустую* строку, а не значение по умолчанию.

Пример использования:

```

auto s;
s="";
while(s!="-")
{
s=AskStr(s,"Ваше имя?");
Message
("Добрый день (утро, вечер, ночь!) %s! \n",s);
}

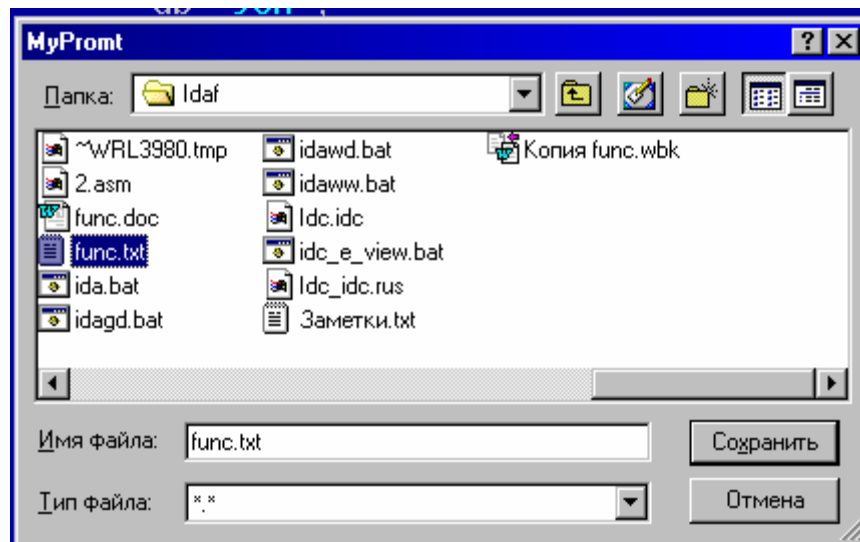
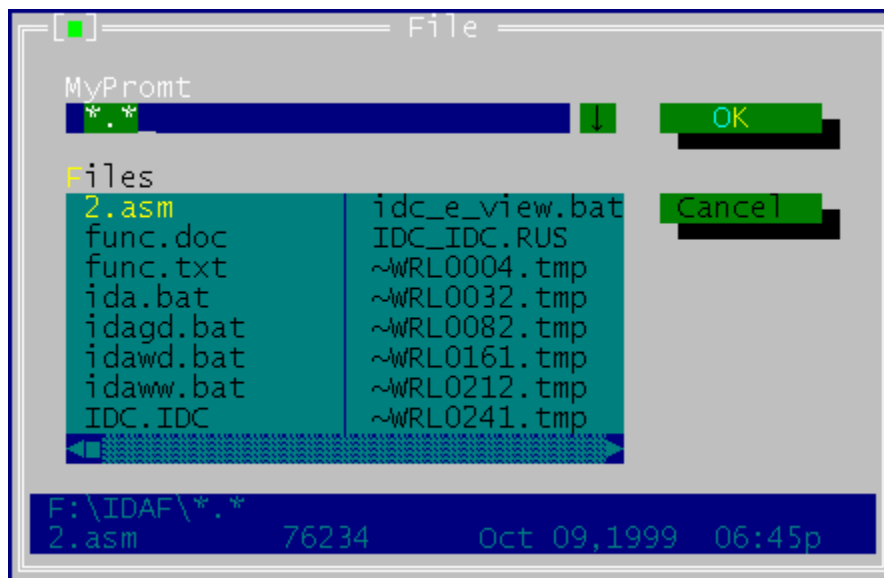
```



**char AskFile (long forsave, char mask, char prompt);**

Функция выдает диалоговое окно, предназначенное для выбора файла, оснащенное минимальными средствами навигации.

Внешний вид окна для консольной и GUI версий, естественно различен. И в последнем случае у пользователя значительно больше возможностей и свободы действий.



Флаг 'forsave', вероятно, должен был уточнять тип окна – на открытие файла или на запись. За кажущейся идентичностью (и то и другое окно с точки зрения пользователя выглядит одинаково) скрыта большая разница. Окно выбора файла для записи должно само запрашивать подтверждение, если запрошенный файл уже существует.

IDA, однако, это не делает независимо от значения флага forsave. И в любом случае не выдает никаких подтверждений, если указанный файл уже существует.

Если пользователь откажется от выбора и нажмет <Esc> или CANCEL – функция вернет пустую строку. В противном случае имя файла.

Эта функция поддерживает длинные имена Windows 95\Windows NT 4.0, и это следует учитывать при операциях с именами файлов (например, синтаксическом разборе)

операнд	пояснения	
forsave	Флаг, выбора типа диалога. Не поддерживается в настоящих версиях	
mask	Маска для выбора отображаемых в окне файлов	
prompt	Заголовок окна	
Return	Завершение	Пояснения
	!=""	Имя файла
	""	Ошибка

Пример использования (два приведенные выше окна были созданы с помощью вызова)

```
AskFile(0, "*. *", "MyPrompt");
```

**long AskAddr (long defval, char prompt);**

Функция выводит модальный диалог, запрашивающий у пользователя ввод адреса в формате segment : offset. Если сегмент указан, то функция вернет значение, вычисленное по следующей формуле.

$$\text{Value} == \text{segment} \ll 4 + \text{offset}$$

При этом функция позволяет указывать не только адреса, но и имена сегментов, вычисляя при этом адреса автоматически (обратите внимание, что при этом необходимо соблюдать регистр).

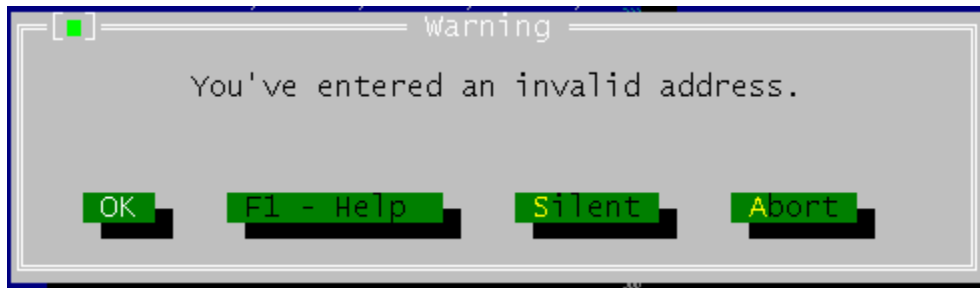
Например:

```
Message ("%x \n", AskAddr(0, "MyPrompt"));
```

```
[seg000:0]
```

```
10000
```

Но вот уже [Seg000:0] приведет к выводу диалога, предупреждающего о неверно введенном адресе, а функция возвратит ошибку BADADDR.

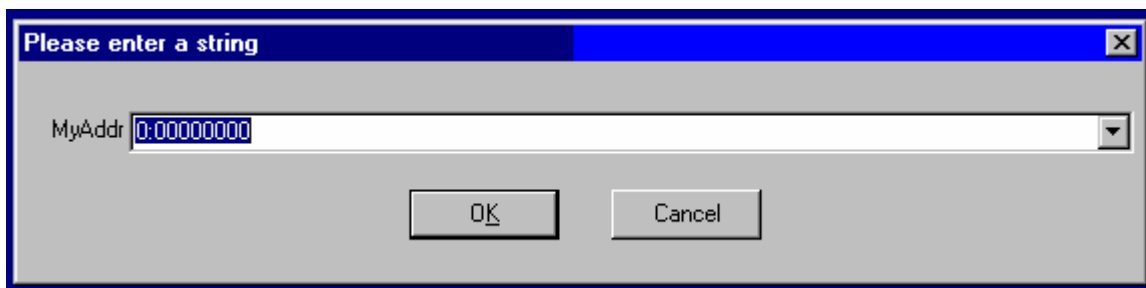
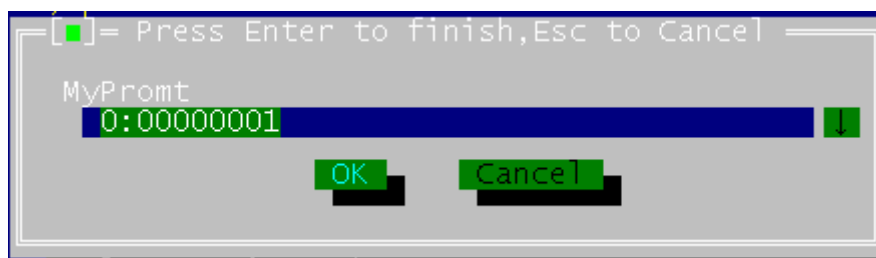


Если не указывать сегмент (а только одно смещение), то функция возьмет по умолчанию базовый сегмент файла.

Например:

```
Message ("0x%x \n", AskAddr (0, "MyAddr")); <Ctrl-Enter>
123h <Enter>
0x10123
```

Если уж необходимо, что бы функция воспринимала ввод «как он есть», то следует вместо сегмента указать '0', как показано ниже:



Очень полезна поддержка относительного адреса. Если перед вводимым числом указать знак, то возвращенное функцией значение будет вычислено по следующей формуле:

$$\text{RetVal} == \text{ScreenEA}() + \text{EntVal}$$

То есть вычисляется адрес, относительно текущей позиции курсора. При этом полученное значение может выходить за рамки доступных адресов сегмента, - никакой проверки функция не обеспечивает, - эта задача ложится на плечи пользователя, то есть программиста, разрабатывающего скрипт.

Например:

```

seg000:32E8> db 21h ;
seg000:32E9 db 0 ;
seg000:32E9 seg000 ends :
Message ("0x%X \n", AskAddr(0, "")); <Ctrl-Enter>
+4 <Enter>
0x132ED

```

Обратите внимание, что если указать '+0:4', то IDA будет трактовать такое выражение совершенно иначе! А именно, как абсолютный адрес.

При этом отрицательные значения преобразуются в беззнаковое с учетом разрядности сегмента (16 или 32 бит), а переполнение «вверх» никак не отслеживается. Это дает возможность адресовать память, как в пределах сегмента, так и за ними.

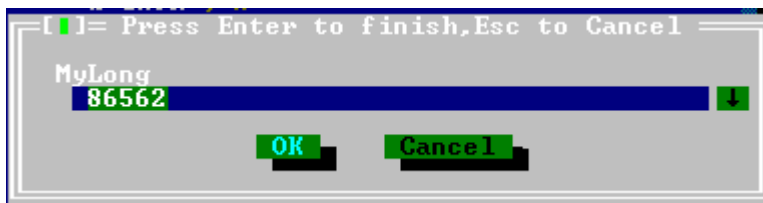
Операнд	Пояснения	
defval	Значение по умолчанию (long)	
prompt	Заголовок окна	
Return	Завершение	Пояснения
	!=BADADDR	Адрес
	""	Ошибка

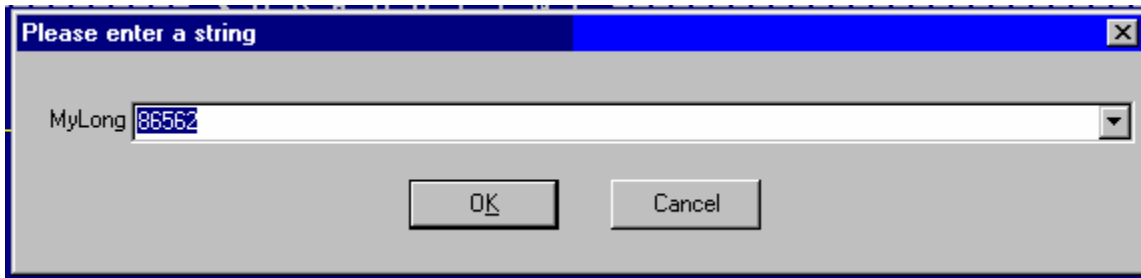
Обратите внимание, что 'defval' имеет значение long, а не char, и, следовательно, представляет собой линейный адрес, преобразование которого в сегментный ложится на плечи IDA. Логично, что было бы необходимо воспользоваться следующей формулой –  $seg = EntVal / 10$ ;  $off = EntVal - seg$ , однако, до версии 4.0 IDA не выполняет никаких операций над адресом, используя нулевой сегмент, если только адрес невозможно представить комбинацией уже существующего сегмента и смещения. То есть, '0x10002' будет автоматически преобразовано IDA в 'seg000:2'. При этом всегда проверяются принадлежность образовавшегося смещения к доступному диапазону адресов выбранного сегмента и в случае нарушении границ, никакого преобразования не происходит.

### long AskLong (long defval, char prompt);

Функция запрашивает у пользователя ввод длинного целого числа. По умолчанию используется шестнадцатеричная система исчисления. Префикс 'x' можно ставить, а можно не ставить – все равно число будет трактоваться как шестнадцатеричное.

Отмена ввода или некорректный ввод приводит к возвращению ошибки BADADDR и, возможно, предупреждающему диалоговому окну, поясняющим источник ошибки.





Пример использования:

```
AskLong (86562, "MyLong") ;
```

Операнд	Пояснения	
defval	Значение по умолчанию	
prompt	Заголовок окна	
Return	Завершение	Пояснения
	!=BADADDR	Введенное пользователем число
	""	Ошибка

#### long AskSeg (long defval,char prompt);

Функция выводит диалог, запрашивающий ввод сегмента (селектора). Допустимо вводить имена сегментов с учетом регистра. Введенные селекторы автоматически не преобразуются адреса сегментов и эту операцию приходится выполнять вручную.

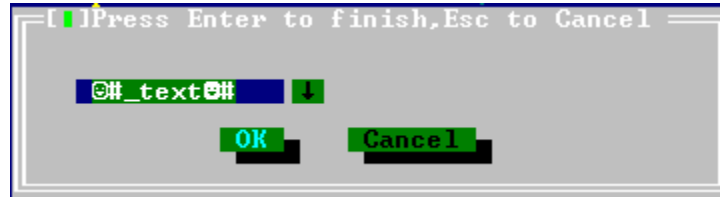
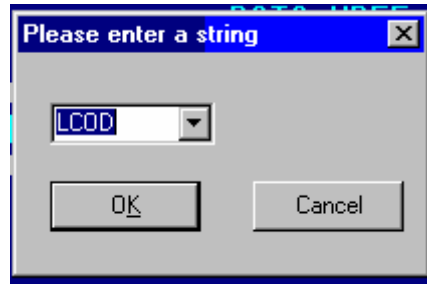
В любом случае функция возвращает значение типа word, а не long. О факте выхода за допустимые границы IDA не сообщает, просто отбрасывая старшее слово введенного значения.

Операнд	Пояснения	
defval	Значение по умолчанию (long)	
prompt	Заголовок окна	
Return	Завершение	Пояснения
	!=BADADDR	Сегмент
	""	Ошибка

Обратите внимание, что 'defval' имеет тип long, а не char. Поэтому непосредственная передача имени сегмента по умолчанию невозможна. Однако IDA автоматически подставляет его, если сегмент с заданным адресом уже существует.

К сожалению, в IDA, включая версию 4.0, присутствует ошибка, в результате чего, вместо ожидаемого символьного имени сегмента выводится нечто нечитаемое и непечатаемое.

```
AskSeg (1, "");
```



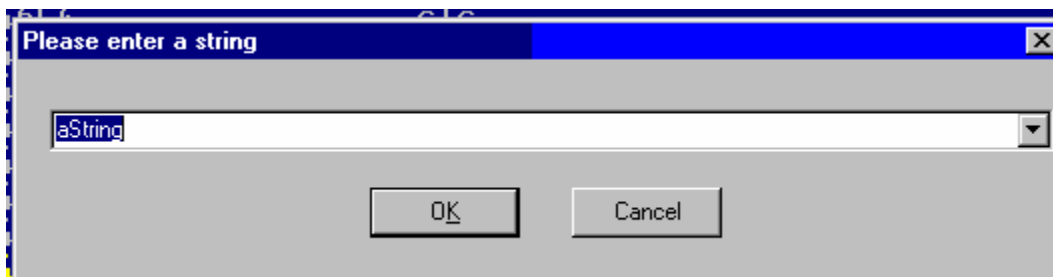
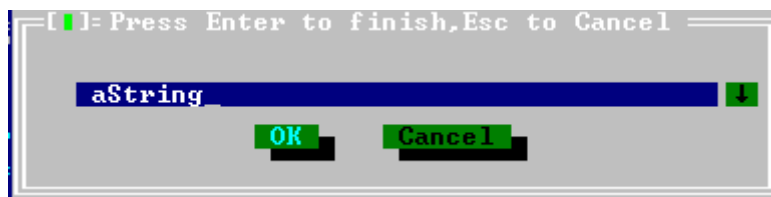
Обратите внимание, что IDA успешно распознала переданный ей селектор и определила какому сегменту он принадлежит. Обратная операция, к сожалению не поддерживается.

В случае ошибки (или отмены) ввода возвращается ошибка BADSEL (не BADADDR!). Это происходит потому, что функция маскирует старшее слово, в результате чего  $(0xFFFFFFFF \& 0xFFFF) == 0xFFFF$ , то есть BADSEL, а не BADADDR и не -1.

#### **char AskIdent (char defval, char prompt);**

Эта функция предназначена для ввода идентификатора (имени). От AskStr ее отличает лишь дополнительная проверка корректности (максимальная длина имени, первый символ строки не цифра и так далее).

В отличие от остальных функций, возвращающих в случае неверного ввода ошибку, AskIdent возвращает управление только дождавшись корректного ввода или его явной отмены.



Если строка начинается с символа '@', то функция всегда возвращает «»; двоеточие не считается недопустимым символом, даже если оно находится в середине строки.

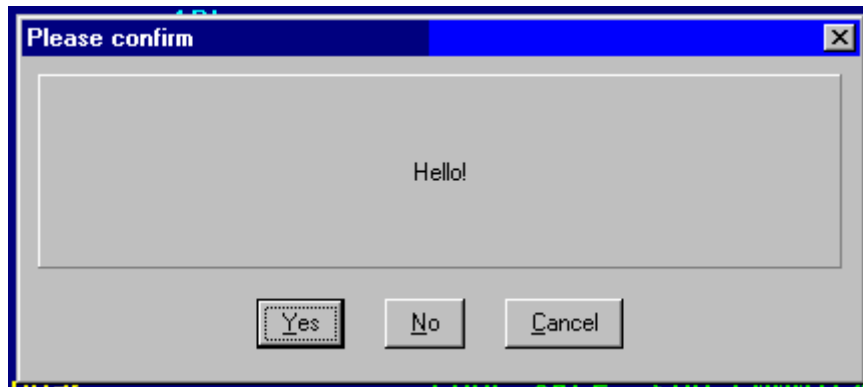
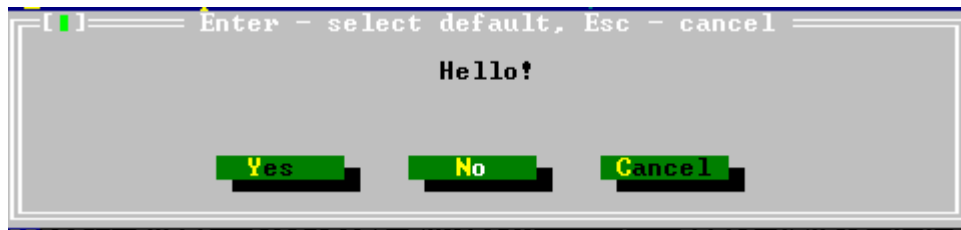
Поэтому в некоторых ответственных случаях не помешает воспользоваться функцией AskStr и все необходимые проверки выполнить самостоятельно.

Операнд	Пояснения	
defval	Значение по умолчанию	
prompt	Заголовок окна	
Return	Завершение	Пояснения
	!=""	Строка
	""	Ошибка

### long AskYN (long defval,char prompt);

Функция создает модальный диалог "Yes \ No \ Cancel".

```
AskYN(1, "Hello!");
```



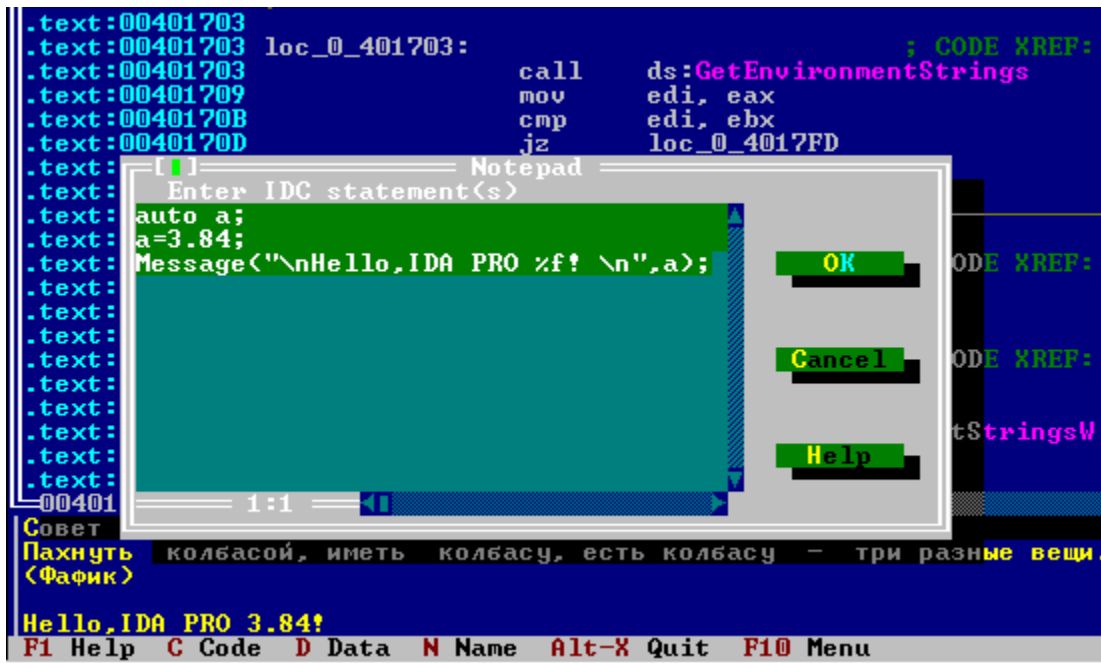
Операнд	Пояснения	
Defval	Значение по умолчанию	
	==defval	Копка по умолчанию
	0	<NO>
	1	<YES>
	-1	<CANCEL>

prompt	Текст, выводимый в окне диалога	
return	0	Пользователь нажал <NO>
	1	Пользователь нажал <YES>
	-1	Пользователь нажал <CANCEL> или Escape

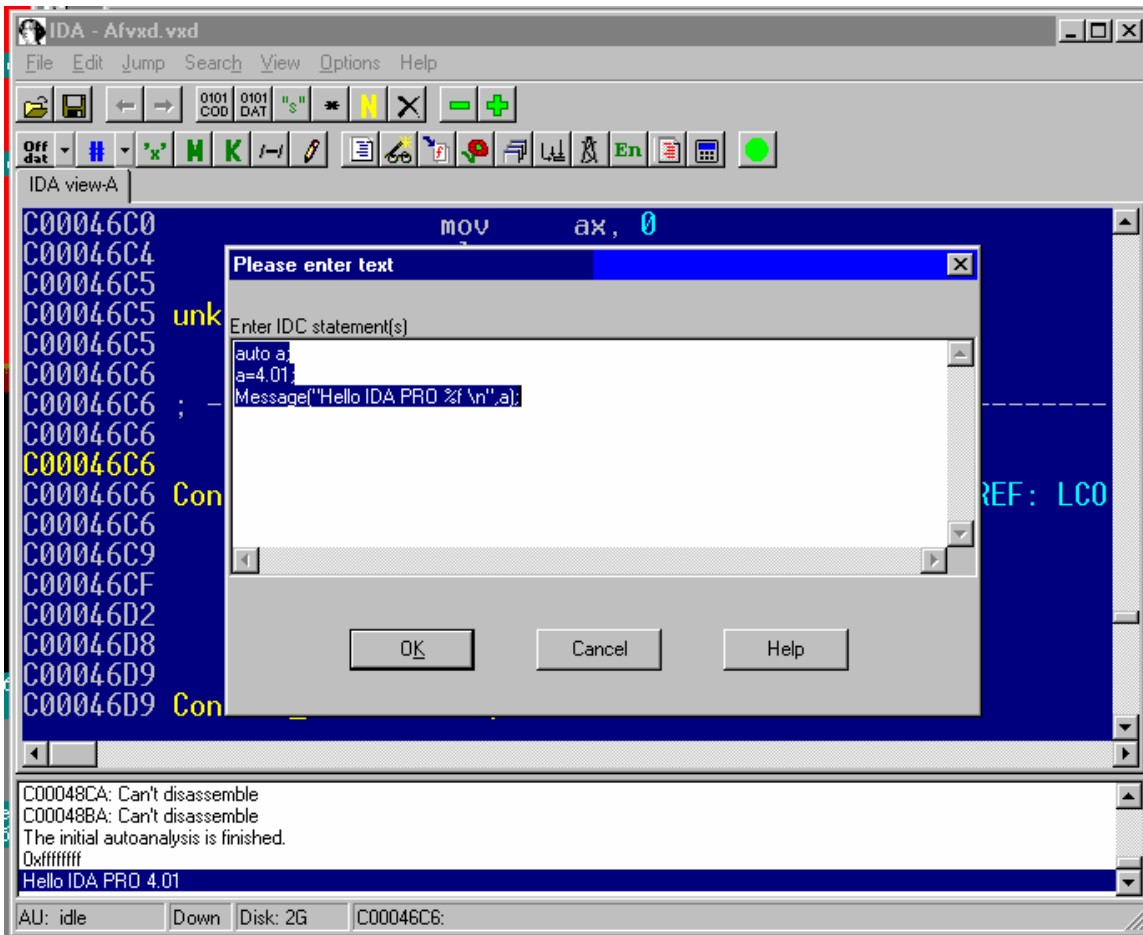
**void Message (char format,...);**

Функция выводит строку в окно сообщений (Messages windows) IDA. Это наиболее популярный способ вывода результатов работы скриптов, а так же отладочных и диагностических сообщений.

Перед выполнением примера убедитесь, что окно сообщений не закрыто остальными окнами.







Message понимает стандартные спецификаторы формата вывода Си и ближе всего близка к функции printf (смотри так же описание form).

Сф	пояснение
%d	десятичное длинное знаковое целое Пример: <code>Message ("%d", 0xF);</code> 15
%x	шестнадцатеричное длинное целое строчечными символами Пример: <code>Message ("%x", 10);</code> a
%X	шестнадцатеричное длинное целое заглавными символами Пример: <code>Message ("%X", 10);</code> A
%o	восьмеричное длинное знаковое целое Пример: <code>Message ("%o", 11);</code> 13
%u	десятичное длинное беззнаковое целое Пример: <code>Message ("%u", -1);</code>

	4294967295
%f	десятичное с плавающей точкой Пример: <pre>Message("%f", 1000000);</pre> 1.e6
%c	символьное значение Пример: <pre>Message("%c", 33);</pre> !
%s	строковое значение Пример: <pre>Message("%s", "Hello, Word! \n");</pre> Hello, Word!
%e	вывод чисел в экспоненциальной форме Пример: Пример: <pre>Message("%e", 1000000);</pre> 1.e6
%g	вывод чисел в экспоненциальной форме <b>ЗАМЕЧАНИЕ:</b> В оригинале спецификатор '%g' заставляет функцию саму решать, в какой форме выводить число - с десятичной точкой или в экспоненциальной форме, из соображений здравомыслия и удобочитаемости. IDA всегда при задании этого спецификатора представляет числа в экспоненциальной форме.
	вывод указателя (не поддерживается) <b>ЗАМЕЧАНИЕ:</b> вместо спецификатора '%p' IDA использует '%a', преобразующее линейный адрес в строковой сегментный, и автоматически подставляет имя сегмента. Так, например, 'message("%a \n", 0x10002)' выдаст 'seg000:2'. Обратите внимание, что таким способом <b>нельзя</b> узнать адрес переменной. Пример: <pre>auto a;</pre> <pre>a="Hello!\n";</pre> <pre>Message("%a \n", a);</pre> 0 <b>Возвращается ноль, а не указатель на переменную.</b>
%p	вывод десятичного целого всегда со знаком, не опуская плюс.
%+d	в оригинале - вывод шестнадцатеричного целого всегда со знаком, но ida воспринимает эту конструкцию точно так же как и 'x'.
%+x	'n' длина выводимого десятичного числа, при необходимости дополняемая слева пробелами. Например: <pre>Message("Число-%3d \n", 1);</pre> Число- 1 Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком. Например: <pre>Message("число-%3d \n", 10000);</pre> число-10000
%nd	'n' длина выводимого шестнадцатеричного числа, при необходимости дополняемая слева пробелами. Например: <pre>Message("Число-%3x \n", 1);</pre> Число- 1

	<p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Например:</p> <pre>Message ("Число-%3x \n", 0x1234); Число-1234</pre>
%nd	<p>'n' длина выводимого десятичного числа, при необходимости дополняемая слева незначащими нулями.</p> <p>Пример:</p> <pre>Message ("Число-%03d", 1); Число-001</pre> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Пример</p> <pre>Message ("Число-%03d", 1000) Число-1000</pre>
%0nx	<p>'n' длина выводимого шестнадцатеричного числа, при необходимости дополняемая слева незначащими нулями.</p> <p>Пример:</p> <pre>Message ("Число-%03x", 0x1); Число-001</pre> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Пример:</p> <pre>Message ("число-%03x", 0x1234); Число-1234</pre>
%#x	<p>Вывод префикса '0x' перед шестнадцатеричными числами</p> <p>Пример:</p> <pre>Message ("%#x", 123); 0x123</pre>
%#o	<p>Вывод префикса '0' перед восьмеричными числами</p> <p>Пример:</p> <pre>Message ("%#o", 1); 01</pre>
%n	Количество выведенных символов (не поддерживается)

### void Warning (char format,...);

Функция выводит диалоговое окно, предупреждающие об аварийной ситуации. Обратите на тип возвращаемого значения void. То есть функция не предоставляет информации, о том какая клавиша была нажата.

<OK> или <ESC> просто возвращают управление скрипту; <Abort> приводит к аварийному выходу из IDA (правда перед этим у пользователя будет запрошено подтверждение). А <SILENT> включает «тихий» режим, в котором подобные окна не отображаются.

```
Warning ("Hello!");
```



Ситуаций, в которых бы требовался аварийный выход из IDA очень немного. Между тем – эта функция вторая по популярности после Message. Очень часто она используется как простой информирующий диалог, не ожидающий от пользователя никакого выбора (например, так поступает демонстрационный плагин Strings)

Сравните приведенный выше пример с результатом демонстрации работы AskYN. Не правда ли “HELLO! – OK” вполне очевидно, тогда как “HELLO! YES? NO? CANCEL?” может вызвать легкое недоумение и растерянность.

К тому же Warning в отличие от AskYN, поддерживает стандартные спецификаторы форматированного вывода Си. (Подробнее смотри описание функции Message)

Но все же использование Warning по поводу и без повода – относится к «дурным» приемам программирования, которых следует по возможности избегать.

**void Fatal (char format,...);**

Эта функция создает модальный диалог, выводящий указанное сообщение и немедленно аварийно выходит из IDA **без подтверждений**.

Существует очень немного случаев, требующих применения столь «варварских» средств.

```
Fatal ("Hello");
```





Функция поддерживает стандартные спецификаторы Си, которые подробнее были описаны в функции Message.

**long           ScreenEA ();**

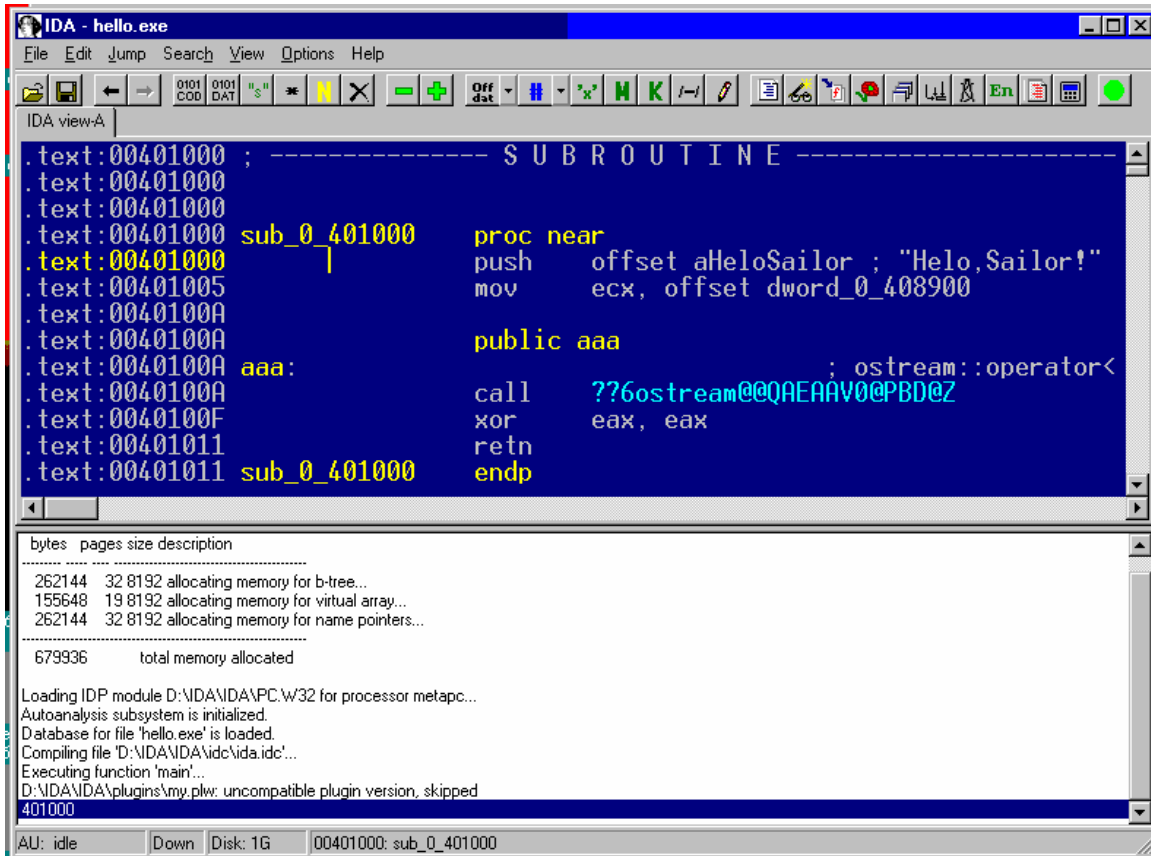
Возвращает линейный адрес в текущей позиции курсора. Очень широко используется в скриптах, в том числе и приведенных в этой книге.

Позволяет организовать взаимодействие между скриптом и пользователем, а так же облегчает вычисление линейного адреса в произвольной точке. Вместо того, что бы искать адрес начала сегмента по имени и суммировать его с необходимым смещением можно просто ткнуть курсором в нужное место и вызвать эту функцию.

Однако обратите внимание, что возвращаемый адрес всегда округляется до начала строки. Невозможно выбрать элемент массива, отличный от первого. Особенно это доставляет много неудобств при просмотре дизассемблируемого файла в шестнадцатеричном виде, когда независимо от положения курсора в строке функция всегда возвращает адрес ее начала.

Пример использования:

```
text:004010F4 sub_0_4010F4 proc near ; DATA XREF:
text:004010F4 mov esp, [ebp-18h]
text:004010F7 push dword ptr [ebp-20h]
text:004010FA call __exit
text:004010FA sub_0_4010F4 endp
```



```

Message ("%x \n",
ScreenEA ()
);

```

4010f7

Return	==return	Пояснения
	!=BADADDR	Линейный адрес начала элемента в текущей позиции курсора
	==BADADDR	Ошибка

**long SelStart ();**

Возвращает линейный адрес начала выделенной области. Широко используется для работы с блоками и позволяет организовать взаимодействие между скриптом и пользователем.

Выделять можно только строки целиком и аналогично функции ScreenEA() можно узнать только адрес начала строки. Если выделение отсутствует, то возвращается ошибка (BADADDR).

Пример использования:

```

seg000:0B50      pusha
seg000:0B51      push     ds
seg000:0B52      push     es
seg000:0B53      call    sub_0_E89

```

```

Message ("%x \n",
 SelStart ()
);

```

10B52

Return	==return	Пояснения
	!=BADADDR	Линейный адрес начала выделенной области
	==BADADDR	Ошибка

### Long SelEnd ();

Возвращает линейный адрес первого байта за концом выделенной области. Если выделение отсутствует, то функция вернет ошибку BADADDR.

Пример использования:

```

seg000:0B50      pusha
seg000:0B51      push     ds
seg000:0B52      push     es
seg000:0B53      call    sub_0_E89

```

```

Message ("%x \n",
 SelEnd ()
);

```

10B53

Return	==return	Пояснения
	!=BADADDR	Линейный адрес следующего байта за концом выделенной области
	==BADADDR	Ошибка

### success Jump (long ea);

Функция перемещает позицию курсора в окне дизассемблера IDA по требуемому адресу.

операнд	Пояснение	
ea	32-разрядный линейный адрес	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

Очень активно используется в пользовательских скриптах. Однако имеет ряд тонкостей.

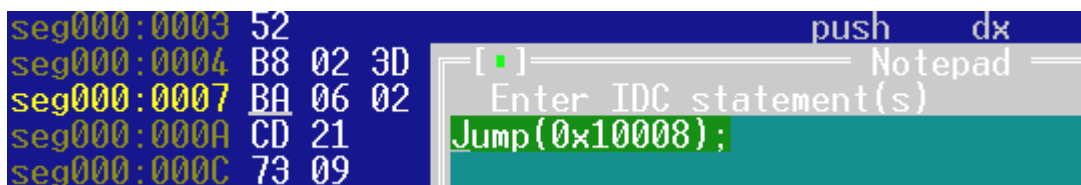
Прежде всего, экран обновляется не в момент выполнения функции, а только после завершения работы скрипта. Поэтому следующий пример не будет правильно выполняться:

```
while(1)
Jump(AskAddr(0x10000,"Введите адрес для перехода"));
```

Это не позволяет динамически иллюстрировать работу скрипта и заставляет изощряться в поиске решений, когда требуется интерактивное взаимодействие вместе с показом нужного кадра окна.

Приходится предусматривать временный выход и последующий вход из скрипта, благо это возможно. Достаточно лишь сохранять значения всех переменных в массиве или виртуальной памяти (ну для любителей экзотики или перестраховщиков - в файле). Но это все же концепция непривычная рядовому программисту.

Другим минусом является округление адреса перехода до целой строки.



Особенно это неудобно при переключении экрана в шестнадцатеричный режим. Jump не позволяет указывать на конкретный байт, а только на всю строку целиком.

При задании несуществующего адреса курсор не изменяет своей позиции, а функция возвращает ноль.

#### **void Wait ();**

Функция ожидает конца авто анализа, после чего возвращает управление. Большинство скриптов не могут работать параллельно с фоновым дизассемблером по тем очевидным причинам, что ожидают полностью готовый к употреблению текст, а не динамически и непредсказуемо изменяющийся.

Авто анализ происходит при загрузке нового файла, а так же при выполнении некоторых операций с исследуемым текстом. В интерактивном режиме можно дождаться окончания авто анализа визуально, но в пакетном так не получится.

Для этого и служит эта функция. Хороший пример ее использования можно найти в файле 'analys.idc', поставляемом вместе с IDA.

#### **long AddHotkey(char hotkey, char idcfunc);**

Функция задает новую комбинацию клавиш для вызова функции IDA, определенной пользователем. Это очень удобное средство для интеграции своих скриптов в интерфейсную оболочку IDA.

Операнд	назначение
Hotkey	Требуемая комбинация клавиш. Записывается в виде символьной строки. Например, "Alt - A". Могут так же использоваться "Ctrl", "Shift", "Enter" а так же их комбинации.
idcfunc	Символьное имя функции. Например, 'MyFunc'.



В файле `idc.idc` содержатся следующие определения констант, связанных с возвращением значением этой функцией.

Определение	константа	назначение
<code>IDCHK_OK</code>	0	успешное завершение
<code>IDCHK_ARG</code>	-1	неверные аргументы
<code>IDCHK_KEY</code>	-2	ошибка в синтаксисе горячей клавиши
<code>IDCHK_MAX</code>	-3	задано слишком много горячих клавиш.

Создадим и откомпилируем для примера следующий файл:

```
static MyFunc()
{
    Message("hello, IDA! \n");
}
```

Введем с консоли `'AddHotkey("ALT-A","MyFunc");'`. Если теперь нажать `'Alt-A'`, то на экране появиться приветствие `'Hello, IDA!'`.

Заметим, что перекрывать существующие клавиатурные комбинации можно совершенно безболезненно, за исключением того, что они автоматически не удаляются и 'кушают' при этом немного ресурсов, да и число "горячих клавиш" ограничено.

Поэтому ненужные в этом момент комбинации рекомендуется предварительно удалять функцией `'DelHotkey'`.

### **success DelHotkey(char hotkey);**

Функция удаляет заданные пользователем "горячие клавиши". При попытке удаления системной или несуществующей комбинации функция возвратит ошибку.

Операнд `'hotkey'` был рассмотрен в описании функции `AddHotKey`.

Пример использования:

```
DelHotkey ("Alt-A");
```

Операнд	назначение
Hotkey	Требуемая комбинация клавиш. Записывается в виде символьной строки. Например, "Alt - A". Могут так же использоваться "Ctrl", "Shift", "Enter" а так же их комбинации.
Return	<code>==return</code> Пояснения
	<code>==1</code> Успешное завершение
	<code>==0</code> Ошибка

## **МАРКИРОВКА ПОЗИЦИЙ ДЛЯ БЫСТРОГО ПЕРЕМЕЩЕНИЯ**

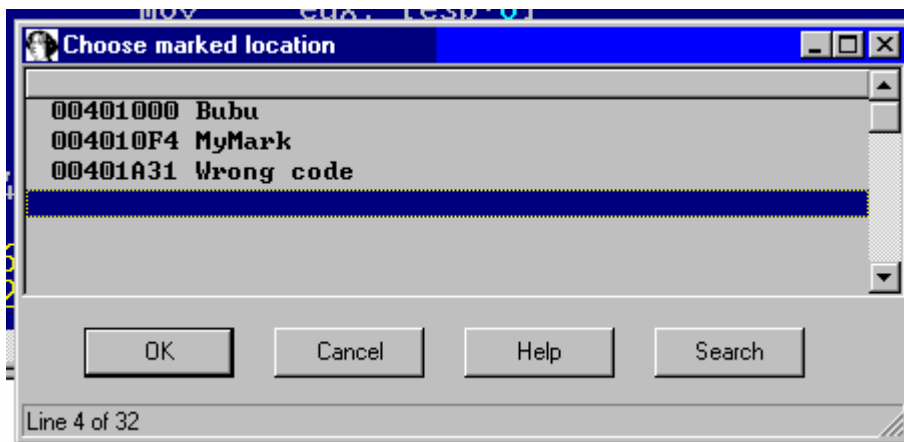
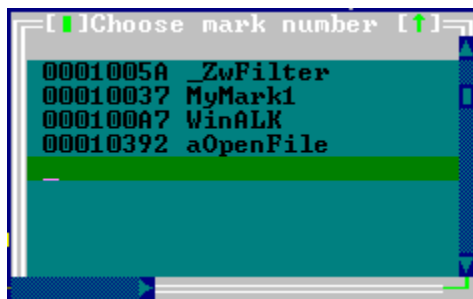
### **МЕТОДЫ**

Функция	Назначение

<code>void MarkPosition(long ea,long Innum,long x,long y,long slot,char comment);</code>	Добавляет элемент в список быстрых переходов
<code>long GetMarkedPos(long slot);</code>	Возвращает линейный адрес закладки
<code>char GetMarkComment(long slot);</code>	Возвращает комментарий к закладке

IDA поддерживает возможность быстрого перемещения между отдельными фрагментами дизассемблируемого текста с сохранением позиции курсора и относительного положения текста в окне.

Для запоминания текущей позиции необходимо нажать <Alt-N>, а для вызова списка всех запомненных ранее позиций <Ctrl-N>. При этом возникнет следующего вида диалоговое окно:



IDA позволяет формировать содержание этого списка не только интерактивно, но и с помощью функций встроенного языка. Это может быть удобно в тех случаях, когда скрипт в результате анализа возвращает требующие внимания со стороны пользователя линейные адреса. Чаще всего их просто выводят в окно сообщений, но это плохое решение. Гораздо удобнее вывести их в список быстрых переходов.

**`void MarkPosition(long ea,long Innum,long x,long y,long slot,char comment);`**

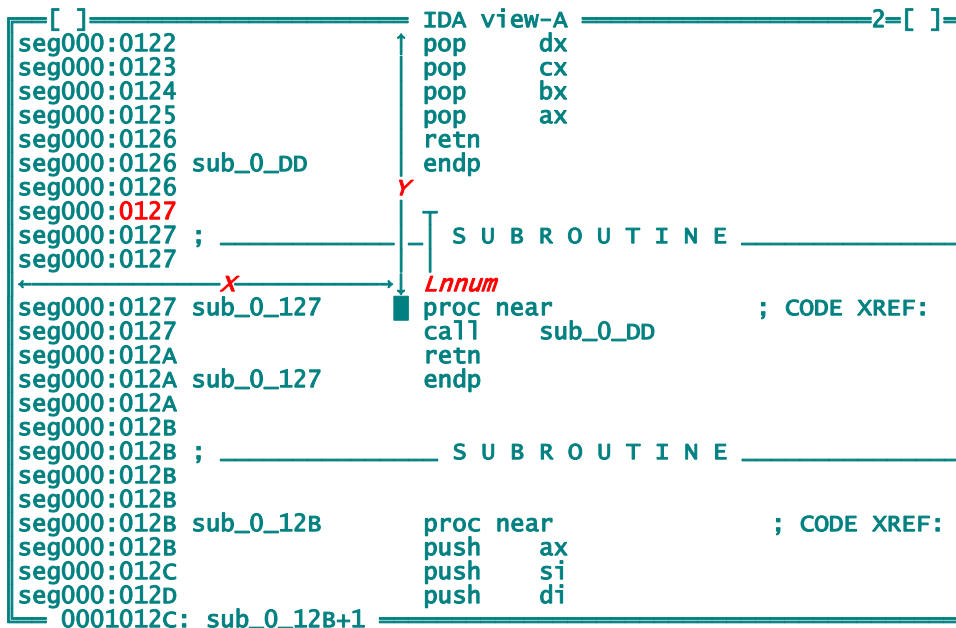
Функция добавляет новый элемент в список быстрых переходов. Каждый элемент характеризуется следующим набором атрибутов.

Прежде всего, это линейный адрес строки, в которой расположен курсор. Поскольку, часто по одному и тому же адресу расположено несколько строк, то атрибут 'lnnum' указывает на требуемую строку, считая от нуля.

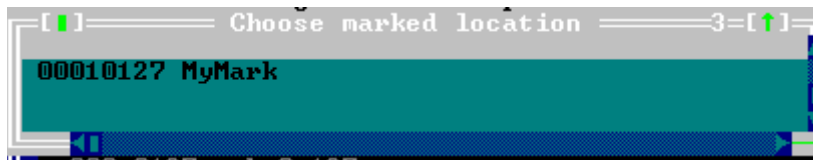
Позиция курсора по горизонтали, начиная от левого края окна, задается атрибутом 'x', а 'y' по вертикали, считая от верхней границы окна. Поскольку курсор жестко связан с выбранной строкой, то IDA прокручивает текст в окне дизассемблера на требуемую величину.

Положение элемента в списке определяется атрибутом Slot. Он может принимать любые значения в интервале от 1 до 20. Элементы не обязательно должны следовать друг за другом. Однако IDA не уничтожает пустые элементы в списке и поэтому задача их упорядочивания ложится на плечи разработчиков скрипта. Ситуация осложняется тем, что существует только один глобальный список, разделяемый одновременно как пользователем, так и всеми скриптами. Прежде, чем заносить новый элемент рекомендуется проверить, что требуемый слот свободен. Если указать слот, выходящий за допустимые границы, то IDA выведет интерактивный диалог для его выбора.

Рассмотрим это подробнее на следующем примере:



MarkPosition(0x10127, 4, 26, 12,1, "MyMark");



Допустимо существование двух и более объектов по одному и тому же линейному адресу, поскольку IDA идентифицирует их по номеру слота.

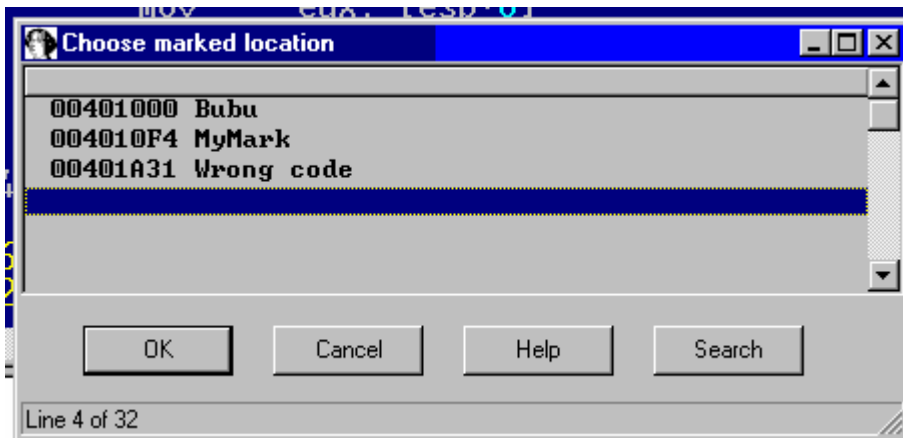
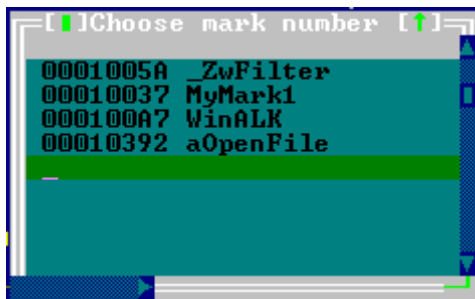
Операнд	Пояснения
ea	Линейный адрес начала строки
lnnum	Номер линии, располагающейся по тому же адресу начиная с нуля

X	Горизонтальное положение курсора относительно левой границы окна	
Y	Вертикальное положение курсора относительно верхней границы окна	
	==slot	Пояснения
slot	==1-20	Номер слота
	0   >20	Интерактивный выбор номера слота
Comment	Комментарий к закладке	

### long GetMarkedPos(long slot);

Функция возвращает линейный адрес закладки по указанному slot. Подробнее о закладках можно прочитать в описании функции SetMarkedPos.

Например:



```
auto a;
for (a=1;a<21;a++)
if (GetMarkedPos(a)!=-1) Message("0x%X \n", GetMarkedPos(a));
```

```
0x1005A
0x10037
0x100A7
0x10392
```

Обратите внимание, что следующий код не является корректным:

```
auto a,x;
a=0;
```

```
while ((x=GetMarkedPos(a++)!=-1)
Message("0x%X \n", x);
```

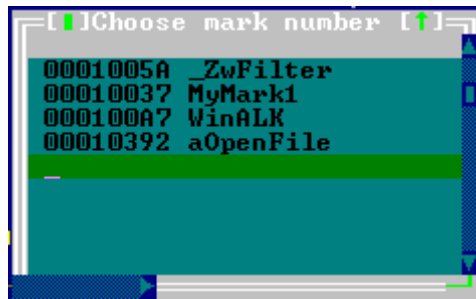
Закладки не обязаны следовать одна за другой, и могут разделяться пустыми слотами. В этом случае приведенный выше код дойдет лишь до первой такой «дырки».

Операнд	Пояснения	
	==slot	Пояснения
slot	==1-20	Номер слота
	0   >20	Интерактивный выбор номера слота
Return	==return	Пояснения
	!=BADADDR	Линейный адрес закладки
	==BADADDR	Ошибка

### char GetMarkComment(long slot);

Функция возвращает комментарий к закладке, расположенной по указанному slot. Подробнее о закладках можно прочитать в описании функции SetMarkedPos.

Например:



```
auto a;
for (a=1;a<21;a++)
if (GetMarkComment(a)!=-1)
Message("%s \n", GetMarkComment (a));
```

```
_ZwFilter
MyMark1
WinALK
aOpenFile
```

Обратите внимание, что следующий код не является корректным:

```
auto a,x;
a=0;
while ((x=GetMarkComment (a++)!=-1)
Message("%s \n", x);
```

Закладки не обязаны следовать одна за другой, и могут разделяться пустыми слотами. В этом случае приведенный выше код дойдет лишь до первой такой «дырки».

Операнд	Пояснения	
	==slot	Пояснения
Slot	==1-20	Номер слота
	0   >20	Интерактивный выбор номера слота
Return	==return	Пояснения
	!="	Комментарий к закладке
	=="	Ошибка

## ГЕНЕРАЦИЯ ВЫХОДНЫХ ФАЙЛОВ

**int GenerateFile(long type, long file\_handle, long ea1, long ea2, long flags);**

Функция генерирует выходной файл. Аналогичена по действию команде меню «~File\Produce output file».

Типичный пример использования приведен в файле Analyst.idc, поставляемым вместе с IDA.

Допустимы следующие типы отчетов:

Определение	Тип файла отчета
OFILE_MAP	файл с отладочной информацией
OFILE_EXE	exe файл
OFILE_IDC	база IDA в виде IDC файла
OFILE_LST	полный файл отчета
OFILE_ASM	готовый к ассемблированию файл
OFILE_DIF	файл различий (более известный как crk)

MAP-файл записывается в стандарте Borland и выглядит приблизительно следующим образом:

```

Start  Stop  Length Name                Class
00000H 032E9H 032E9H seg000          CODE

Address          Publics by Value
0000:0002        MyLabelName
0000:0206        aScreen_log
0000:03EA        aDeifxcblst
0000:22C0        start
0000:2970        aOtkrivaemFail
0000:297F        aMyfile
0000:2980        aYfile
0000:298F        aCalc

Program entry point at 0000:22C0

```

Он обычно используется для облегчения отладки программ. В коде становится легче ориентироваться по символьным меткам, переменным и функциям. Например, вместо абсолютного адреса для точки останова можно указывать его имя. Говорящие названия улучшают восприятие кода и не дают запутаться и повторно возвращаться к уже проанализированным фрагментам.

Указанный формат поддерживают Borland Turbo Debugger, Periscope, а так же другие отладчики. Популярный Soft-ICE имеет в стандартной поставке конвертор, преобразующий такие файлы к своему собственному формату.

Некоторые отладчики не поддерживают сегментацию, а другие имеют ограничение на количество имен, поэтому генерацией файла можно управлять. Для этого определены следующие значения флага 'flag':

определение	пояснения
GENFLG_MAPSEGS	включать в файл карту сегментов
GENFLG_MAPNAME	включать «dummy» имена.

«Dummy» имена представляют собой автоматически генерируемые IDA имена, используемые для определения меток, процедур и данных. Они выглядят в виде sub\_, loc\_, off\_, seg\_ и так далее. Обычно с целью не захламления листинга они не включаются в файл.

EXE файл генерируется после того, как программа была изменена функциями PatchByte или PatchWord. Эти функции не изменяют оригинального файла, а только содержимое базы IDA, и что бы изменения возымели действия необходимо сгенерировать новый файл.

К сожалению IDA поддерживает очень ограниченный список форматов. Вот это и все, что можно получить на выходе:

1. MS DOS .exe
2. MS DOS .com
3. MS DOS .drv
4. MS DOS .sys
5. general binary
6. Intel Hex Object Format
7. MOS Technology Hex Object Format

При этом exe файл генерируется заново. Он содержит ту же таблицу перемещаемых элементов (то есть ее невозможно изменить), а все неиспользуемые структуры заполняются нулями. Следует иметь ввиду, что некоторые программы чувствительны к таким изменениям и откажут в работе.

К сожалению, не поддерживаются PE и другие win32 файлы. В этом случае (а так же когда exe файл чувствителен к неиспользуемым полям, – например, в свободное пространство заголовка иногда может быть помещен оверлей) можно сохранить различия в DIF файле и затем любой из поддерживающих его многочисленных утилит модифицировать оригинальный файл.

IDA позволяет сохранять базу в виде текстового IDC файла. Это обеспечивает ее переносимость между различными версиями. Дело в том, что основной рабочий формат IDB в любой момент может измениться и база перестанет загружаться в новые версии. Для преодоления этой проблемы и был введен текстовой формат.

Заметим, что это далеко не вся база и часть информации оказывается необратимо утерянной, например, отсутствует виртуальная память и для анализа вновь потребуется исходный файл, кроме того, загружаться IDC файл будет гораздо медленнее IDB, поскольку потребуются вновь все заново дизассемблировать. Поэтому применять данный формат в качестве рабочего совершенно бессмысленно.

Но что же представляет из себя IDC файл? Как нетрудно догадаться по его расширению это обыкновенный скрипт!

```
static Segments(void)
{
    SegCreate(0x10000, 0x132ea, 0x1000, 0, 1, 2);
    SegRename(0x10000, "seg000");
    SegClass(0x10000, "CODE");
    SetSegmentType(0x10000, 2);
}
```

И его можно безболезненно редактировать в отличие от бинарного IDB формата. Например, если IDA что-то неправильно дизассемблирует, то положение будет нетрудно исправить, отредактировав нужным образом скрипт.

LST представляет собой копию дизассемблированного файла, в том виде, в каком он отображается на экране IDA. Выглядит он приблизительно так:

```
seg000:0100 loc_0_100:
seg000:0100          cmp     byte ptr [bx+si], 0
seg000:0103          jz     loc_0_108
seg000:0105          inc    bx
seg000:0106          jmp    short loc_0_100
```

Разумеется, он не пригоден для последующего ассемблирования и может использоваться только в качестве «твердой копии экрана». В демонстрационной версии генерация LST файла не поддерживается.

ASM файл – это дизассемблированный файл полностью готовый к ассемблированию. Выглядит он следующим образом:

```
p586n
; -----
; Segment type: Pure code
seg000      segment byte public 'CODE' use16
            assume cs:seg000
            assume es:nothing, ss:nothing, ds:nothing, fs:nothing,
; _____ S U B R O U T I N E _____
sub_0_0     proc near                                ; CODE XREF: sub_0_22DD+1E_p
            push    ax
            push    bx
            push    cx
            push    dx
            mov     ax, 3D02h
```

В демонстрационных версиях вывод дизассемблированного текста в ASM файл не поддерживается.

DIF хранит в себе результаты сравнения оригинального и модифицированного функциями PatchByte и PatchWord файлов. Для некоторых форматов IDA позволяет генерировать исполняемый (или бинарный) файл, с учетом изменений.

Однако в большинстве случаев этих возможностей оказывается недостаточно (например, не поддерживаются win32 форматы) и тогда приходится прибегать к сохранению всех изменений в отдельном файле.

Формат его показан ниже:

This difference file is created by The Interactive Disassembler

```
xsafe-iv.exe
00002390: 0C 11
```

В нем нетрудно распознать типичный crk файл, который поддерживается многими утилитами (например, cra386) или модифицировать исходный файл вручную. Несложно написать скрипт на IDA-си, который будет выполнять такую работу автоматически.

Для генерации любого типа файлов требуется задать виртуальный адрес начала и конца области. Если требуется вывести файл целиком, то в качестве адреса начала



можно задать 0, а в качестве конца константу BADADDR или -1.

Функция GenerateFile не работает с именами файлов, она требует дескриптора уже открытого на запись файла. В упрощенном виде ее вызов может выглядеть так:

```
auto a;  
a=fopen("myfile.ext","wt");  
GenerateFile (OFFILE_ASM, a, 0, -1,0);  
fclose (a);
```

Поскольку только в исключительно редких случаях требуется модификация только что сгенерированного файла, то полезно будет создать макрос или функцию, включающую в себя приведенный выше текст. Это упростит ее вызов и позволит программисту не отвлекаться на посторонние мелочи.

операнд	Пояснения
type	Тип генерируемого файла
file_habdle	Дескриптор открытого файла
ea1	Линейный адрес начала области для отображения в файле
ea2	Линейный адрес конца области для отображения в файле
flags	Флаги, управляющие, генерацией файла

## ФАЙЛОВЫЙ ВВОД – ВЫВОД

IDA обладает развитым файловым вводом \ выводом, что открывает поистине неограниченные возможности. Можно самостоятельно загружать файлы любых форматов, можно создать отчеты и листинги любых видов. Можно распаковывать или модифицировать исполняемые файлы. Но даже при желании работать с принтером, или, например, модемом!

Все это богатство возможностей реализуется относительно небольшим набором стандартных функций Си. Работа с файлами в IDA - Си ничем не отличается от «классического» Си. За тем, может быть, исключением, что ввиду отсутствия массивов в их общепринятом понимании, используется посимвольный, а не блочный обмен.

### long fopen (char file,char mode);

Функция открывает файл и возвращает его обработчик в случае успешного завершения этой операции.

Прототип функции полностью совпадает с аналогичной функцией в стандартной библиотеке Си. Действительно, реализация этой функции в IDA только передает управление библиотечной функции fopen(char \*,char \*) без какой – либо дополнительной обработки аргументов.

Необходимые атрибуты доступа задаются флагом mode в виде простой символьной строки. Их возможные значения будут показаны ниже.

Атрибут	Назначение
w	Открывает файл для записи. Если файл не существует, то он автоматически создается. Содержимое любого непустого файла будет уничтожено начиная с текущей позиции (по умолчанию с начала файла).
r	Открывает файл для чтения. Если указанный файл не существует,

	то функция возвратит ошибку (NULL)
a	Открывает файл для записи и перемещает его указатель в конец, (то есть фактически открывает файл для до записи). Если указанный файл не существует, то он будет автоматически создан.
r+	Открытие файла на запись и чтение. Если файл не существует, то функция возвратит ошибку
w+	Открытие файла на запись и чтение. Если файл не существует, то он будет автоматически создан. Содержимое уже существующего файла будет уничтожено
a+	Открывает файл на чтение и дозапись в конец. Если файл не существует, то он автоматически будет создан

Тип файла	Пояснения
t	Открыть файл, как текстовый. В этом режиме символ CTRL-Z (ASCII 27) трактуется, как конец файла. Так же по-особому транслируется символ переноса 'n'. Компилятор превращает его в код 0xA. При записи же его в текстовый файл функция на самом деле поместит комбинацию 0xD 0xA – интерпретируемую сервисами MS-DOS и некоторыми элементами управления Windows, как перенос на следующую строку. Часто с текстовыми файлами удобнее работать, открывая их как бинарные (смотри ниже)
b	Открыть файл как бинарный. В этом режиме все символы транслируются AS IS, без каких либо изменений.

Функцию необходимо вызвать обязательно с указанием атрибута доступа и типа файла, иначе она завершится с ошибкой.

Если файл по каким-то причинам открыть не удалось, то функция возвратит ноль, в противном случае дескриптор открытого файла.

Примеры использования:

Del file.dem

```
Message("0x%X \n", fopen("file.dem", "wb");
1
```

```
dir file.dem
```

```
file.dem 0 11.11.99 13:33 file.dem
```

```
Message("0x%X \n", fopen("Long File Name", "wb");
```

```
1
```

```
dir longfi~1
```

```
LONGFI~1 0 11.11.99 15:06 Long File Name
```

```
Message("0x%X \n", fopen("myfile", "r+b");
```

```
0
```

Обратите внимание, что IDA возвращает один и тот же обработчик при открытии различных файлов, хотя прежние файлы не были явно закрыты. Это говорит о том, что они закрываются автоматически, после того как скрипт завершит свою работу.

Часто забывают, что в Windows сохранилась поддержка имен устройств, идущая еще со времен CP/M. Поэтому, что бы вывести данные на печать достаточно открыть на

запись устройство "PRN" и направить в него необходимые данные.  
Например:

```
writestr(fopen("PRN","wt"),"Hello,Printer!");
```

Необходимо лишь учитывать, что эта печать идет в обход менеджера печати и, кроме того, так нельзя получить доступ к сетевому принтеру. Но в большинстве случаев и этих возможностей окажется достаточно.

Точно так же можно читать данные с консоли или выводить их на нее. Конечно, при первой же перерисовке окна сообщений они будут стерты, но это наоборот, скорее достоинство, чем недостаток. Действительно, зачем загромождать окно сообщений?

Операнд	Пояснения	
File	Имя файла (при необходимости с полным или частичным путем). Обе версии IDA (консольная и GUI) поддерживает длинные файлы.	
mode	Атрибуты доступа и типа файла.	
Return	Завершение	Пояснения
	Успешное	Дескриптор открытого файла (!=0)
	Ошибка	==0

Закрытие всех открытых файлов гарантируется только при корректном выходе из IDA. И хотя, несмотря на то, что операционная система гарантированно закроем все файлы, порожденные любым процессом (в том числе и IDA) при его завершении, может потеряться часть данных, которая в этот момент находилась во внутренних буферах IDA и еще не была записана на диск.

#### **void fclose (long handle);**

Функция закрывает файлы, открытые с помощью fopen. В момент закрытия файла в него записываются все данные, находящиеся в этот момент во внутренних буферах, а файловый объект (то есть то, на что ссылается дескриптор) уничтожается, предотвращая утечку ресурсов.

Файлы автоматически закрываются в момент завершения работы породившего их скрипта (при условии, что обладающая дескриптором процедура не описана как static), а так же при корректном завершении работы IDA.

В противном же случае операционная система все равно освободит все ресурсы, принадлежащие процессу, но при этом не будут записаны данные, оставшиеся во внутренних буферах.

Функция не возвращает результата успешности операции.

Операнд	Пояснения
handle	Дескриптор открытого файла

Пример использования:

```
Auto a;  
A=fopen("PRN","wt");  
If (a!=-1)  
writestr(a,"Hello,Printer!");  
fclose(a);
```

Обратите внимание, что в приведенном примере `fclose` выполняется даже тогда, когда файл не был успешно открыт. Это не ошибка, поскольку `fclose(0)` не приводит ни к каким побочным последствиям.

### **long filelength (long handle);**

Функция возвращает логическую длину открытого файла. То есть длину с учетом не записанных внутренних буферов, которая может не совпадать с физическим размером файла на диске в данный момент.

Длина символьных устройств (таких, как PRN, например) всегда равна нулю. Например:

```
Message("0x%X \n",filelength(fopen("PRN", "wt")));
0x0
```

Операнд	Пояснения
handle	Дескриптор открытого файла

### **long fseek (long handle,long offset,long origin);**

Функция позиционирует указатель в открытом файле. Флаг `origin`, задающий необходимое позиционирование может принимать следующие значения, перечисленные ниже в таблице:

origin	Значение
0	Позиционировать относительно начала файла
1	Позиционировать относительно текущей позиции
2	Позиционировать относительно конца файла

Правда вплоть до версии 4.0 эта функция реализована с ошибкой, приводящей к тому, что флаг '1' трактуется точно так, как и '0' – то есть относительно начала файла.

Это видно на следующем примере:

```
auto a;
a=Fopen("myfile","wt");
fseek(a,0x10,0);
Message("0x%X \n",ftell(a));
fseek(a,0x0,1);
Message("0x%X \n",ftell(a));
fclose(a);

0x10
0x0
```

Так же не поддерживается отрицательная адресация относительно начала файла. Относительно конца файла можно свободно позиционироваться в двух направлениях.

```
auto a;
a=Fopen("myfile","wt");
fseek(a,0x0,2);
```

```

Message("0x%X \n", ftell(a));
fseek(a, 0x5, 2);
Message("0x%X \n", ftell(a));
fseek(a, -0x5, 2);
Message("0x%X \n", ftell(a));
fclose(a);

```

```

0x100
0x105
0x100

```

Напомним, что ранние версии DOS содержали ошибку, приводящую к тому, что наращивание размера файла с помощью функции позиционирования сверх определенного размера (зависящего от многих обстоятельств) нарушалась целостность FAT16.

Та же ошибка повторена в первых реализациях FAT32 (Windows 95 OSP0, в народе прозванная «Лебединая редакция»)

При перемещении указателя за конец файла в него подает, информация, расположенная в выделяемых ему дисковой подсистемой кластерах. При этом файл не должен быть открыт на запись, иначе игнорируя флаг, функция будет вычислять смещение относительно файла и его содержание окажется утерянным!

Так же оно окажется утерянным, если указать неверное отрицательное смещение или origin > 2.

Операнд	Пояснение	
Handle	Обработчик открытого файла	
Offset	Смещение (относительно конца файла - знаковое)	
Origin	Указывает, относительно чего отсчитывается смещение (смотри таблицу выше)	
Return	Завершение	Возвращаемое значение
	Успешно	0
	Ошибка	!=0

### **long ftell (long handle);**

Функция возвращает текущую позицию указателя в открытом файле относительно его начала.

Операнд	Пояснения	
handle	Дескриптор открытого файла	
Return	Завершение	Возвращаемое значение
	Успешно	Текущая позиция
	Ошибка	-1

### **success loadfile (long handle, long pos, long ea, long size);**

Функция позволяет загружать бинарный файл (или его часть) в произвольный регион виртуальной памяти IDA. Это позволяет писать свои динамические загрузки, но и эмулировать работу оверлеев, а так же многое другое.

Перед началом операции искомым файл необходимо открыть в бинарном режиме функцией fopen с правами только на чтение. Если открыть на запись, то его содержимое окажется необратимо уничтожено!

Затем указать позицию в файле для чтения (аргумент pos). Позиция всегда считается относительно начала файла, не зависимо от текущего положения указателя.

Далее указать виртуальный линейный адрес, по которому будет скопирован фрагмент файла. Операция завершится независимо от того, производится загрузка в границах существующего сегмента или вне оных. При необходимости IDA выделяет дополнительную виртуальную память для загрузки.

Последний аргумент, передаваемый функции – это число загружаемых из файла байт. Если оно превосходит длину «хвоста» файла (то есть от указанной позиции до конца), то IDA выдаст предупреждение:

```
Can't read input file (file structure error?), only part
of file will be loaded...
```

И загрузит столько байт, сколько сможет.

```
seg000:2C93 aWatchAvialable db 'Watch avialable DOS memory.....'

auto a;
a=fopen("readme.txt","rb");
loadfile(a,0,0x12C93,0x40);

seg000:2C93 aWatchAvialable db 'This patch allows you to permanently access the bonus'
```

Обратите внимание, что загрузка не вызывает реассемблирования исследуемой программы. При этом только перезаписывается соответствующий регион виртуальной памяти, но не меняются связанные с ней флаги!

Это хорошо видно на следующем примере:

```
seg000:02E4 sub_0_2E4      proc near                ; CODE XREF: seg000:232Ep
seg000:02E4                push ds
seg000:02E5                xor ax, ax
seg000:02E7                mov ds, ax                ; DS == NULL
seg000:02E9                assume ds:nothing
seg000:02E9 MyLabel:
seg000:02E9                mov ax, ds:413h
seg000:02EC                shl ax, 6
seg000:02EF                cmp ax, 0A000h
seg000:02F2                pop ds
seg000:02F3                assume ds:seg000
seg000:02F3                retn
seg000:02F3 sub_0_2E4      endp

auto a;
a=fopen("readme.txt","rb");
loadfile(a,0,0x102E4,0x40);

seg000:02E4 sub_0_2E4      proc near                ; CODE XREF: seg000:232Ep
seg000:02E4                push sp
seg000:02E5                push 7369h
seg000:02E7                jnb loc_0_309             ; DS == NULL
seg000:02E9                assume ds:nothing
seg000:02E9 MyLabel:
seg000:02E9                jo loc_0_34C
seg000:02EC                arpl [bx+si+20h], bp
seg000:02EF                popa
seg000:02F2                outsw
seg000:02F3                assume ds:seg000
```

```

seg000:02F3          ja      near ptr loc_0_367+1
seg000:02F3 sub_0_2E4  endp

```

Обратите внимание, что не только сохранились прежние метки, комментарии и перекрестные ссылки, но и оказался неверно дизассемблированным код! Но это не ошибка IDA, а ее архитектурная особенность. Вместе с обычными перекрестными ссылками сохранились и так называемые ссылки на следующую инструкцию. Поэтому вновь загруженный код был дизассемблирован с учетом прежнего «каркаса» то есть линейный адресов начала инструкций.

Что бы исправить ситуацию, необходимо пометить измененный фрагмент, как undefined и потом его заново ассемблировать. В результате получится следующее:

```

seg000:02E4          push   sp
seg000:02E5          push   7369h
seg000:02E8          and    [bx+si+61h], dh
seg000:02EB          jz     loc_0_350
seg000:02ED          push   6120h
seg000:02F0          ins   byte ptr es:[di], dx
seg000:02F1          ins   byte ptr es:[di], dx
seg000:02F2          outsw

```

Чаще всего эту функцию используют для частичного дизассемблирования файла. Например, если внутри много мегабайтовой DLL необходимо исследовать лишь небольшой фрагмент, то нет нужды несколько часов ждать пока IDA дизассемблирует ее целиком – достаточно лишь загрузить требуемый фрагмент.

Кроме того, многие приложения во время работы подгружают различные свои компоненты с диска. Если их так же необходимо исследовать, то для этого можно воспользоваться loadfile.

Иногда даже не требуется создавать для этого дополнительный сегмент, загрузив данные за его границы.

```

seg000:32A0          db  0E2h, 20h, 0A4h, 0A0h, 2 dup(0ADh), 0EBh, 0A9h, 20h
seg000:32A0          db  0ACh, 0A5h, 0E5h, 0A0h, 0ADh, 0A8h, 0A7h, 0ACh, 21h
seg000:32A0          db  0
seg000:32A0 seg000  ends
seg000:32A0
seg000:32A0
seg000:32A0          end start

```

```

auto a;
a=fopen("readme.txt","rb");
loadfile(a,0,0x102E4,0x10);

```

```

seg000:32A0          db  0E2h, 20h, 0A4h, 0A0h, 2 dup(0ADh), 0EBh, 0A9h, 20h
seg000:32A0          db  0ACh, 0A5h, 0E5h, 0A0h, 0ADh, 0A8h, 0A7h, 0ACh, 21h
seg000:32A0          db  0
seg000:32A0 seg000  ends
seg000:32A0          end start
0:000132EA          db  54h ; T
0:000132EB          db  68h ; h
0:000132EC          db  69h ; i
0:000132ED          db  73h ; s
0:000132EE          db  20h ;
0:000132EF          db  70h ; p
0:000132F0          db  61h ; a
0:000132F1          db  74h ; t
0:000132F2          db  63h ; c
0:000132F3          db  68h ; h

```

Доступ к загруженным данным может быть осуществлен, например, вызовами Byte. Для интерактивной же работы (например, что бы преобразовать загруженные данные в строку) все же придется создать сегмент (как это сделать рассказано в описании функции SegCreate)

### MySeg:000A ; Segment type: Regular

```
MySeg:000A MySeg      segment byte public '' use16
MySeg:000A           assume cs:MySeg
MySeg:000A           ;org 0Ah
MySeg:000A           assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
MySeg:000A aThisPatchAllow db 'This patch allows you to permanently access the bonus track '
MySeg:000A MySeg      ends
MySeg:000A
```

Строго говоря, приведенный пример дизассемблирован не правильно. Если программа подгружала ресурсы из текстового файла динамически во время работы, то перемещение их в сегмент может даже нарушить ее работоспособность после ассемблирования и уж точно не изменит алгоритм, так что бы файловый обмен заменился обращением к памяти.

Но на этапе исследования дизассемблируемого кода это невероятно удобно. Можно даже вычислить какие инструкции, какой код загружают, и создать перекрестные ссылки для обеспечения быстрого перехода между различными фрагментами дизассемблируемого текста.

Операнд	Пояснения	
handle	Обработчик открытого только на чтение файла	
pos	Позиция в файле, относительно его начала	
ea	Линейный адрес начала региона виртуальной памяти	
Size	Число байт для чтения	
Return	Завершение	Пояснения
	0	Функция завершилась неуспешно
	1	Функция завершилась успешно

### success savefile (long handle,long pos,long ea,long size);

Функция, обратная loadfile (смотри описание выше). Она позволяет сохранить фрагмент виртуальной памяти на диске в виде файла.

Например:

```
seg000:03D3 sub_0_3D3      proc near                ; CODE XREF: seg000:03C7p
seg000:03D3          push     ax
seg000:03D4          push     bx
seg000:03D5          mov     al, byte ptr es:loc_0_F+1
seg000:03D9          mov     bx, 3EAh
seg000:03DC
seg000:03DC loc_0_3DC:          ; CODE XREF: seg000:03E4j
seg000:03DC          cmp     [bx], al
seg000:03DE          jz     loc_0_3E7
seg000:03E0          inc     bx
seg000:03E1          cmp     byte ptr [bx], 0
seg000:03E4          jnz    loc_0_3DC
seg000:03E6          inc     bx
seg000:03E7
seg000:03E7 loc_0_3E7:          ; CODE XREF: seg000:03DEj
seg000:03E7          pop     bx
seg000:03E8          pop     ax
seg000:03E9          retn
seg000:03E9 sub_0_3D3      endp
seg000:03E9          db     'DEIFXCBLST',0
seg000:03EA aDeifxcblst
```

```
auto a;
a=fopen("fileme","wb");
```



```
savefile(a, 0, 0x103D9, 0x200);
```

```
[ ] F:\IDAF\fileme 23:28:03  
00000000: 50 53 26 A0 10 00 BB EA 03 38 07 74 07 43 80 3F | PS&a.+.ъ8tCA?  
00000010: 00 75 F6 43 5B 58 C3 44 45 49 46 58 43 42 4C 53 | .uЎC[X+DEIFXCBLs
```

Возможность сохранения отдельных фрагментов файла очень полезна и может стать основой для множества утилит (например, такой, что извлекает все текстовые строки, встретившиеся в программе в отдельный файл)

Кроме того, она пригодится, если необходимо сохранить модифицированный вызовами PatchByte файл на диск. Дело в том, что IDA не поддерживает экспорт ни во что другое, кроме com и MS-DOS EXE. И то, и другое, очевидно, давным-давно устарело. И поддержку формата популярных сегодня PE файлов придется реализовывать самостоятельно.

Перед началом операции необходимо открыть целевой файл на запись с помощью функции fopen и передать savefile его дескриптор.

Позиция в файле для записи может быть выбрана любая, как внутри, так и вне него. Однако, в последнем случае возможно разрушение FAT, поэтому необходимо соблюдать дополнительные меры предосторожности.

Размер записываемого фрагмента может превосходить длину дизассемблируемого файла, в этом случае «хвост» будет заполнен символами 0xFF, (именно такое значение возвращает функция Byte при попытке чтения несуществующих адресов), но функция, не смотря на это завершится без ошибки.

Операнд	Пояснения	
handle	Обработчик открытого на запись файла	
pos	Позиция в файле, относительно его начала	
ea	Линейный адрес начала региона виртуальной памяти	
Size	Число байт для записи	
Return	Завершение	Пояснения
	0	Функция завершилась неуспешно
	1	Функция завершилась успешно

## long fgetc (long handle);

Функция читает один байт из файла. При этом файл должен быть предварительно открыт вызовом fopen с правами, разрешающими чтение. Относится к функциям стандартной библиотеки Си.

При неуспешном возвращении возвращает ошибку BADADDR – иначе очередной считанный символ. Если не достигнут конец файла, то указатель увеличивается на единицу.

Пример использования:

```
auto a, ch;  
a=fopen("readme.txt", "rt");  
while((ch=fgetc(a))!=-1)  
Message(ch);  
fclose(a);
```

This patch allows you to permanently access the bonus track and bonus car without winning the tournaments.

Операнд	Пояснения
handle	Дескриптор открытого с правами на чтения файла

Return	Завершение	Пояснения
	Норма	Считанный символ
	Ошибка	BADADDR

### long fputc (long byte,long handle);

Функция записывает один байт в файл. Файл должен быть предварительно открыт с правами на запись функцией fopen.

При неуспешной записи возвратит ошибку BADADDR, иначе ноль.

Операнд	Пояснения	
byte	Записываемый символ	
handle	Дескриптор открытого с правами на запись файла	
Return	Завершение	Пояснения
	Норма	0
	Ошибка	BADADDR

### long fprintf (long handle,char format,...);

Ближайший аналог известной функции sprintf, однако, вместо буфера результат копируется в файл. Очевидно, что файл должен быть предварительно открыт с правами на запись вызовом fopen.

Например:

```
auto a,s0;
s0=0x123;
a=fopen("CON","wt");
fprintf(a,"%x\n",s0);
123
```

Управляющие символы стандартные, и частично совместимые с 'printf' и полностью совместимы со спецификаторами функции Message встроенного языка IDA.

Сф	пояснение
%d	десятичное длинное знаковое целое Пример: <code>Message("%d",0xF);</code> 15
%x	шестнадцатеричное длинное целое строчечными символами Пример: <code>Message("%x",10);</code> a
%X	шестнадцатеричное длинное целое заглавными символами Пример: <code>Message("%X",10);</code> A
%o	восьмеричное длинное знаковое целое Пример: <code>Message("%o",11);</code> 13
%u	десятичное длинное беззнаковое целое

	<p>Пример:</p> <pre>Message("%u", -1); 4294967295</pre>
%f	<p>десятичное с плавающей точкой</p> <p>Пример:</p> <pre>Message("%f", 1000000); 1.e6</pre>
%c	<p>символьное значение</p> <p>Пример:</p> <pre>Message("%c", 33); !</pre>
%s	<p>строковое значение</p> <p>Пример:</p> <pre>Message("%s", "Hello, Word! \n"); Hello, Word!</pre>
%e	<p>вывод чисел в экспоненциальной форме</p> <p>Пример:</p> <pre>Message("%e", 1000000); 1.e6</pre>
%g	<p>вывод чисел в экспоненциальной форме</p> <p><b>ЗАМЕЧАНИЕ:</b> В оригинале спецификатор '%g' заставляет функцию саму решать, в какой форме выводить число - с десятичной точкой или в экспоненциальной форме, из соображений здравомыслия и удобочитаемости. IDA всегда при задании этого спецификатора представляет числа в экспоненциальной форме.</p>
	<p>вывод указателя (не поддерживается)</p> <p><b>ЗАМЕЧАНИЕ:</b> вместо спецификатора '%p' IDA использует '%a', преобразующее линейный адрес в строковой сегментный, и автоматически подставляет имя сегмента.</p> <p>Так, например, 'Message("%a \n", 0x10002)' выдаст 'seg000:2'. Обратите внимание, что таким способом <b>нельзя</b> узнать адрес переменной.</p> <p>Пример:</p> <pre>auto a; a="Hello!\n"; Message("%a \n", a); 0</pre> <p>Возвращается <b>ноль</b>, а не указатель на переменную.</p>
%p	<p>вывод десятичного целого всегда со знаком, не опуская плюс.</p>
%+d	<p>в оригинале - вывод шестнадцатеричного целого всегда со знаком, но ida воспринимает эту конструкцию точно так же как и 'x'.</p>
%+x	<p>'n' длина выводимого десятичного числа, при необходимости дополняемая слева пробелами.</p> <p>Например:</p> <pre>Message("Число-%3d \n", 1); Число- 1</pre>
	<p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Например:</p> <pre>Message("число-%3d \n", 10000); число-10000</pre>
%nd	<p>'n' длина выводимого шестнадцатеричного числа, при необходимости дополняемая слева пробелами.</p> <p>Например:</p>

	<pre>Message("Число-%3x \n", 1);</pre> <p>Число- 1</p> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Например:</p> <pre>Message("Число-%3x \n", 0x1234);</pre> <p>Число-1234</p>
%nd	<p>'n' длина выводимого десятичного числа, при необходимости дополняемая слева незначащими нулями.</p> <p>Пример:</p> <pre>Message("Число-%03d", 1);</pre> <p>Число-001</p> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Пример</p> <pre>Message("Число-%03d", 1000)</pre> <p>Число-1000</p>
%0nx	<p>'n' длина выводимого шестнадцатеричного числа, при необходимости дополняемая слева незначащими нулями.</p> <p>Пример:</p> <pre>Message("Число-%03x", 0x1);</pre> <p>Число-001</p> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Пример:</p> <pre>Message("число-%03x", 0x1234);</pre> <p>число-1234</p>
%#x	<p>Вывод префикса '0x' перед шестнадцатеричными числами</p> <p>Пример:</p> <pre>Message("%#x", 123);</pre> <p>0x123</p>
%#o	<p>Вывод префикса '0' перед восьмеричными числами</p> <p>Пример:</p> <pre>Message("%#o", 1);</pre> <p>01</p>
%n	Количество выведенных символов (не поддерживается)

### **long readshort (long handle,long mostfirst);**

Функция считывает два байта из файла. До начала операции файл должен быть открыт функцией fopen с правами на чтение.

Примечательной особенностью данной функции является возможность трансляции знакового бита во время чтения.

Если флаг mostfirst равен нулю, то функция будет полагать, что знаковый бит, расположен «слева», то есть, идет самым старшим в слове. Наоборот, если флаг mostfirst равен единице, то функция будет ожидать, что знаковый бит, расположен «справа» то есть идет самым младшим в слове.

В случае если во время выполнения функции возникнут ошибки, то будет возвращена константа BADADDR – иначе 16-битное прочитанное значение.

Операнд	Пояснения	
handle	Дескриптор открытого с правами на чтение файла	
mostfirst	==0	Знаковый байт самый старший в слове
	==1	Знаковый байт самый младший в слове
Return	Завершение	Пояснения
	Норма	Прочитанное 16-битное знаковое слово
	Ошибка	BADADDR

### **long readlong (long handle,long mostfirst);**

Функция считывает четыре байта из файла. До начала операции файл должен быть открыт функцией `foren` с правами на чтение.

Примечательной особенностью данной функции является возможность трансляции знакового бита во время чтения.

Если флаг `mostfirst` равен нулю, то функция будет полагать, что знаковый бит, расположен «слева», то есть, идет самым старшим в двойном слове. Наоборот, если флаг `mostfirst` равен единице, то функция будет ожидать, что знаковый бит, расположен «справа» то есть идет самым младшим в двойном слове.

В случае если во время выполнения функции возникнут ошибки, то будет возвращена константа `BADADDR` – иначе 32-битное прочитанное значение. Формально функция не возвращает ошибку, потому что она неотличима от возможного 32-битного значения.

Однако в результате ошибки `BADADDR` все же возвращается. Например:

```
Message ("0x%X \n", readlong(123));
0xFFFFFFFF
```

Операнд	Пояснения	
handle	Дескриптор открытого с правами на чтение файла	
mostfirst	==0	Знаковый байт самый старший в слове
	==1	Знаковый байт самый младший в слове
Return	Завершение	Пояснения
	Норма	Прочитанное 16-битное знаковое слово
	Ошибка	BADADDR

### **long writeshort (long handle,long word,long mostfirst);**

Функция записывает два байта в файл. До начала операции файл должен быть открыт функцией `foren` с правами на запись.

Примечательной особенностью данной функции является возможность трансляции знакового бита во время чтения.

Если флаг `mostfirst` равен нулю, то функция будет полагать, что знаковый бит, расположен «слева», то есть, идет самым старшим в слове. Наоборот, если флаг `mostfirst` равен единице, то функция будет ожидать, что знаковый бит, расположен «справа» то есть идет самым младшим в слове.

В случае если во время выполнения функции возникнут ошибки, то будет возвращено ненулевое значение.

Операнд	Пояснения	
Handle	Дескриптор открытого с правами на запись файла	
Mostfirst	==0	Знаковый байт самый старший в слове
	==1	Знаковый байт самый младший в слове
Return	Завершение	Пояснения
	Норма	0
	Ошибка	!=0

### **long writelong (long handle,long dword,long mostfirst);**

Функция записывает четыре байта в файл. До начала операции файл должен быть открыт функцией fopen с правами на запись.

Примечательной особенностью данной функции является возможность трансляции знакового бита во время чтения.

Если флаг mostfirst равен нулю, то функция будет полагать, что знаковый бит, расположен «слева», то есть, идет самым старшим в двойном слове. Наоборот, если флаг mostfirst равен единице, то функция будет ожидать, что знаковый бит, расположен «справа» то есть идет самым младшим в двойном слове.

В случае если во время выполнения функции возникнут ошибки, то будет возвращено ненулевое значение.

Операнд	Пояснения	
Handle	Дескриптор открытого с правами на запись файла	
Mostfirst	==0	Знаковый байт самый старший в слове
	==1	Знаковый байт самый младший в слове
Return	Завершение	Пояснения
	Норма	0
	Ошибка	!=0

### **char readstr (long handle);**

Функция читает строку из файла (с текущей позиции до символа EOL). До начала операции файл должен быть открыт функцией fopen с правами на чтение.

Не зависимо от заданного типа при открытии файла (текстовый или двоичный) readstr всегда правильно распознает конец строки представленный как 0xD 0xA, так и 0xA. Однако если файл открыт как текстовый, то функция будет преобразовывать все символы 0xA в 0xD 0xA. Что можно наблюдать на следующем примере:

```
auto a;
a=fopen("readme.txt","rb");
Message(readstr(a));
```

This patch allows you to permanently access the bonus track and bonus car 🎵

```
auto a;
a=fopen("readme.txt","rt");
Message(readstr(a));
```

This patch allows you to permanently access the bonus track and bonus car

Операнд	Пояснения
Handle	Дескриптор открытого с правами на чтение файла

Return	Завершение	Пояснения
	Норма	Считанная строка
	Ошибка	""

**long writestr (long handle, char str);**

Функция записывает строку в файл. До начала операции файл должен быть открыт функцией fopen с правами на запись.

Если файл открыт как текстовый, то функция будет преобразовывать все символы 0xA в 0xD 0xA.

Операнд	Пояснения	
Handle	Дескриптор открытого с правами на чтение файла	
str	Записываемая строка	
Return	Завершение	Пояснения
	Норма	0
	Ошибка	!=0

## ВИРТУАЛЬНЫЕ МАССИВЫ

### ОРГАНИЗАЦИЯ МАССИВОВ

IDA поддерживает два типа массивов, и это иногда порождает небольшую путаницу.

Первое, массив как структура данных дизассемблируемого файла, (см ~ Edit \ Array) для повышения их удобно читаемости, но принципиально ничем ни отличающийся от тех же данных записанных построчно.

```
seg000:0006      db 0A0h,0ACh,0AEh,0A3h,0AEh, 20h,0ADh,0A0h
seg000:0006      db 0A0h, 20h,0ADh,0A0h,0A4h,0AEh, 20h,0AEh
```

И массивы как выделенные области памяти под нужды скриптовых программ. Вот их-то мы сейчас и рассмотрим. Они концептуально очень сильно отличаются от привычных для нас массивов языков Си и Паскаль.

Скорее это объект, который в Microsoft непременно бы назвали CArray, предоставляющий соответствующие API, но скрывающий реализацию всех своих методов.

Уникальность массивов IDA в том, что они поддерживают смешанный тип данных. В одном и том же массиве можно хранить как числа, так и строки. Правда обработку типов (или в принятой терминологии тегов) IDA возлагает на наши плечи и нам придется явно указывать строковое ли это значение или нет.

Очень приятно, что IDA поддерживает разряженные массивы, то есть индексированные произвольным образом. С первого взгляда они могут напоминать списки, но на самом деле это не так. Обыкновенные разряженные массивы.

Это дает очень большую экономию в тех случаях, когда диапазон индексов значительно превосходит реально используемые данные.

Однако, как уже упоминалось выше, в ряде случаев выгоднее не пользоваться массивами, а создать для этих целей сегмент в виртуальной памяти. Это может, например, упростить ввод \ вывод данных, т.к. эту задачу можно возложить на файловый загрузчик IDA, точно так же и вывод готовых данных можно осуществить штатными функциями, - например, всего одной командой сохранить данные в файле – в любом из многочисленных поддерживаемых IDA форматов.

Но есть задачи, в которых массивы несравненно удобнее. Например, это уже отмечавшаяся работа со списками или строковыми данными и, кроме того, массивы хорошо подходят в качестве долговременного хранилища данных.

Массивы сохраняются в базе IDA (а точнее в Btree) до момента их принудительного удаления. Это же, разумеется, относится и к сегментам, но массивы в отличие от последних не загромождают дизассемблируемый текст.

Попробуем составить нехитрый скрипт, нечто вроде "мимоходных заметок". Некоторых пришедших в голову программиста мыслей, которые и с одной стороны забывать не хочется, но и с другой не имеющим никакого отношения к собственно дизассемблируемому тексту.

Что-то в стиле боевого крика "Пусик хочет кушать", который приходит в голову программиста на восемнадцатом часу изнуряющей работы и не уйдет, пока не будет записан.

Для этого нам потребуется познакомиться с базовыми операциями над массивами. Начнем с создания.

Что бы как-то различать массивы, каждый из них дается уникальное имя (до 120 символов, при этом может начинаться с цифры) и связанный с ним идентификатор, который возвращает функция создания сегмента в случае успешного завершения:

```
long CreateArray(char name);
```

Если массив с таким именем уже существует, то функция возвратит BADADDR. Иначе же мы получим идентификатор массива, который необязательно сохранять, ибо в любой момент при первой необходимости его можно узнать по имени массива. Но что-то одно из двоих сохранить все же придется.

Как узнать идентификатор ранее созданного массива при перезапуске скрипта? Конечно, можно его сохранить как в самой базе, так и во внешнем файле, но удобнее получить его по имени массива, воспользовавшись следующей функцией:

```
long GetArrayId(char name);
```

Если указанного массива не существует, то она возвратит BADADDR, в противном случае идентификатор. С помощью идентификатора массив в любой момент можно переименовать функцией:

```
success RenameArray(long id,char newname);
```

С другой стороны, если Вам не нравятся конструкции типа:

```
auto ID;  
ID=GetArrayId("MyArray");  
RenameArray(ID,"MyRenamedArray");
```

то можно непосредственно получить идентификатор "на лету" типа:



```
RenameArray(GetArrayId("MyArray"),"MyRenamedArray");
```

это экономит одну переменную и ставит под вопрос удобочитаемость листинга (с одной стороны видеть каждый раз перед глазами имя массива удобнее, а с другой одноименная переменная ничуть не хуже)

Кроме того, подобный подход может изрядно понизить скорость работы особенно в цикле. Но он имеет какую-то особую притягательность, и многие программисты часто используют его вопреки здравому смыслу (Вообще-то программисты и здравый смысл понятия мало совместимые)

Создавая массивы, необходимо помнить, что они располагаются не в оперативной памяти, исчезая после перезапуска IDA, а в базе. И перезапуск не разрушает их.

Увлечись созданием массивов, особенно на этапе знакомства и экспериментирования с ними, можно не только "скушать" порядочно ресурсов, но и заблокировать многие имена, так что потом при попытке создания массива с идентичным именем возникнет непонятно с чем связанная на первый взгляд ошибка.

Поэтому сразу же, как необходимость в массиве отпадет, его следует удалить функцией:

```
void DeleteArray(long id);
```

Жалко, что не предусмотрено возможности создания массивов, автоматически удаляющихся при выходе из IDA. Однако, немного поразмыслив, можно найти не очень красивое, но, тем не менее успешно работающее решение.

При запуске выполним следующие действия (для этого достаточно включить эту строку в файл ida.idc):

```
CreateArray("SysListTempArray");
```

Теперь определим функцию:

```
static CreateTempArray(Name)
{
    auto a,temp;
    temp=GetLastIndex('S',GetArrayId("SysListTempArray"));
    a=CreateArray(Name);
    if (a>0) SetArrayString(GetArrayId("SysListTempArray",++temp,Name);
    return a;
}
```

При выходе из IDA уже нетрудно будет удалить все временные массивы из базы автоматически.

Однако, в этом на первый взгляд логичном поступке, есть одна ошибка. Давайте подумаем, а что случится, если сеанс работы будет аварийно завершен? Правильно, наш скрипт не получит управления и временные массивы не будут удалены!

Поэтому необходимо очищать их при **выходе** (запуске) IDA. При этом массив "SysListTempArray" будет необходимо создавать только один раз для каждой новой базы.

Этот пример еще раз наглядно демонстрирует всю мощь интегрированного языка IDA. **Любые** ваши пожелания и фантазии могут быть воплощены в простой или сложный (но чаще всего все же простой) скрипт, который выполнить большую часть работы за вас.

При этом нет никакой необходимости связываться с автором и ждать исполнения пожеланий в последующих версиях (или попросту говоря через неопределенное время).

Массивы IDA имеют и другую уникальность. Один и тот же элемент (а точнее индекс) может одновременно содержать строковое и числовое значения, причем оба не перекрывают друг друга. Т.е. вместо одного массива мы как бы получаем целых два!

Для задания значений элементов используется пара функций:

```
success SetArrayLong (long id,long idx,long value);
```

```
success SetArrayString(long id,long idx,char str);
```

Причем обе функции могут принимать как символьный, так и числовой тип значения.

SetArrayString(id,idx,0x21) занесет в ячейку знак '!' и соответственно SetArrayLong (id,idx,!\*) - 0x2A21.

Это бывает очень удобно для преобразования типов данных, которое IDA выполняет автоматически.

Примечательно, что не нужно предварительно каким-либо образом задавать размер массива. Просто указываете требуемый индекс вот и все.

Всего доступно 0x100000000 индексов (32 бита), что позволяет расширять массивы, не только "вперед", но и "назад".

IDA прекрасно справляется с отрицательными указателями. Не стоит, однако забывать, что отрицательные указатели на самом деле трактуются как беззнаковые и расширение массива "назад" происходит по кольцу.

Чтение элементов массива выполняется несколько неожиданным способом. Вместо двух функций GetArrayLong и GetArrayString используется одна:

```
char or long GetArrayElement(long tag,long id,long idx);
```

Уточнение типа, требуемого элемента выполняется через тег. Если он равен 'A', то функция возвратит числовое значение, и строковое в противном случае.

Впрочем, в IDC.IDC для строковых значений рекомендуется явно указывать тег 'S', поскольку логика его обработки в последующих версиях может быть изменена.

Можно так же использовать и определения AR\_LONG и AR\_STR, однако, на мой взгляд, их несколько утомительнее писать. С другой стороны, использовать непосредственные значения более рискованно в плане возможной несовместимости с последующими версиями.

idx - это индекс элемента массива. Традиционно в большинстве языков программирования (например, Си) нет никаких средств навигации по индексам и даже невозможно узнать, сколько элементов содержит массив и какие из них инициализированные, а какие нет.

Всех этих проблем нет в IDA.. Индекс первого элемента поможет узнать функция:

```
long GetFirstIndex(long tag,long id);
```

Если массив не содержит ни одного элемента, то она возвращает значение -1, в противном случае индекс первого элемента. Обратите внимание, что он не обязательно будет равен нулю, а может принимать любое значение. Первым считается инициализированный элемент с наименьшим индексом.

Соответственно, индекс последнего элемента поможет найти функция:

```
long GetLastIndex(long tag,long id);
```

Следующий или предыдущий индекс в цепочке можно найти с помощью функций:

```
long GetNextIndex(long tag,long id,long idx);
```

и

```
long GetPrevIndex(long tag,long id,long idx);
```

Заметим, что список элементов не замкнут в кольцо и при достижении обоих его концов функции возвратят ошибку, а не "перескочат" на следующий конец.

Ну и, наконец, удалить любой элемент массива можно с помощью функции:

```
success DelArrayElement(long tag,long id,long idx);
```

Теперь можно попробовать реализовать наш проект "Записная книжка". Начнем с создания массива. С первого взгляда стоило бы реализовать такую конструкцию:

```
if (GetArrayId("Notepad")==-1) CreateArray("Notepad");
```

однако, можно ограничиться вызовом `CreateArray("Notepad")`, т.к. если массив уже существует, то функция вернет ошибку вот и все. И если обращаться к массиву по имени, то совершенно необязательно сохранять его ID.

Реализуем функцию "NotepadAdd" для внесения новых записей:

```
static NotepadAdd(s0)
{
    SetArrayString(GetArrayId("Notepad"),
        GetLastIndex(GetArrayId("Notepad"))+1,
        s0);
}
```

И естественно просмотр онных:

```
static NotepadPrint()
{
    auto a;
    a=0;
    Message("Блокнот: \n");
    while((a=GetNextIndex('S',GetArrayId("Notepad"),a))>0)
        Message("%s \n",GetArrayElement('S',GetArrayId("Notepad"),a));
}
```

Чуть позже мы добавим к "Блокноту" соответствующий интерфейс, а пока будем пользоваться его функциями с консоли. Нажмем <Shift-F2> и введем

```
NotepadAdd("Это только тест");
```

и нажмем <Ctrl-Enter>. Затем вызовем консоль еще раз и введем еще одну строку

```
NotepadAdd("Пусик хочет кушать");
```

Попробуем посмотреть содержимое блокнота командой

```
NotepadPrint();
```

```
Блокнот:
Это только тест
Пусик хочет кушать
```

А теперь реализуем наш "универсальный расшифровщик" на массивах и сравним с предбудущими результатами.

```

auto a,temp;
CreateArray("MyArray");
for (a=SegStart(0x10000);a<SegEnd(0x10000);a++)
SetArrayLong(GetArrayId("MyArray"),
Byte(a),GetArrayElement('A',GetArrayId("MyArray"),
Byte(a))+1);
a=GetFirstIndex('A',GetArrayId("MyArray"));
temp=0;
while(1)
{
    if
    (GetArrayElement('A',GetArrayId("MyArray"),a)>GetArrayElement('A',GetArrayId("MyArra
y"),a)) temp=a;
    a=GetNextIndex('A',GetArrayId("MyArray"),a);
}

// процедура дешифровки
//
DeleteArray(GetArrayId("MyArray"));

```

Как видно, массивы имеют определенные преимущества перед использованием виртуальной памяти сегментов для своих нужд.

Поскольку созданный массив заполнен едва ли не на треть, то переход по элементам списка функцией `GetNextIndex()` заметно быстрее перебора всего массива в цикле, как это было в предбудущем примере.

Кроме того, нет никакого риска прочитать неинициализированные элементы массива, что позволяет избежать многих трудноуловимых ошибок.

Большой неожиданностью явилась поддержка IDA Perl-подобных ассоциативных массивов. Кардинальное отличие их обычных заключается в возможности индексирования элементов строковыми значениями.

Например:

```

a["Москва"] = "Москва-Столица";
a["Кремль"] = "Старинное здание в Москве";

```

Конечно, IDA использует другой синтаксис, и данный пример приведен только для облегчения понимания сущности этой возможности.

При этом внутренне представление индексов таких массивов очень компактно и быстродействующие.

Ассоциативные массивы можно считать полу документированной особенностью IDA. В контекстной помощи им уделено всего несколько строк, а в прототипах функций отсутствуют комментарии (впрочем, разработчик IDA считает, что этого вполне достаточно)

Впрочем, с какой-то стороны это и оправдано. Ассоциативные массивы реализованы "поверх" существующих, и отличаются только тем, что используют строковые, а не числовые индексы.

К ним применимы функции `CreateArray`, `GetArrayID`, `RenameArray` и остальные. Различие лишь в том, что все функции ассоциативных массивов не имеют тегов. Это означает, что один и тот же индекс не может одновременно ссылаться на строковое и числовое значение. Поэтому следующий пример вернет '0x1234', т.к. последнее присвоение затирает предыдущее.

```

SetHashString(GetArrayId("MyArray"),"1st","Это строка");
SetHashLong (GetArrayId("MyArray"),"1st",0x1234);
Message("%x \n",GetHashLong(GetArrayId("MyArray"),"1st"));

```

Можно так же безболезненно присваивать ячейке строковое значение, а считывать числовое (и, естественно, наоборот). Это бывает иногда полезно для преобразования данных. Чтение значений осуществляется функциями:

```
long GetHashLong(long id,char idx);
```

```
char GetHashString(long id,char idx);
```

Не смотря на то, что эти функции работают непосредственно с массивом созданным CreateArray они не могу видеть или модифицировать элементы, заданные SetArrayLong\SetArrayString, поэтому можно безбоязненно использовать один массив для разных нужд.

Удалить любой элемент ассоциативного массива можно с помощью функции:

```
success DelHashElement(long id,char idx);
```

При удалении элементов IDA неявно инициализирует случайным значением, в отличие от DelArrayElement, которая в этом случае как и следует ожидать обнуляет ячейку.

Однако, это не вызовет проблемы, если для доступа к элементам использовать функции:

```
char GetFirstHashKey(long id);
```

```
char GetNextHashKey(long id,char idx);
```

```
char GetLastHashKey(long id);
```

```
char GetPrevHashKey(long id,char idx);
```

Все они возвращают строковое представление индексов, и могут быть использованы в функциях GetHashLong\GetHashString.

## МЕТОДЫ

Функция	Назначение
long CreateArray(char name)	Создает новый массив
long GetArrayId(char name)	Возвращает идентификатор массива по его имени
success RenameArray(long id,char newname)	Переименовывает массив
void DeleteArray(long id)	Удаляет массив
success SetArrayLong(long id,long idx,long value)	Присваивает значение индексу массива типа «длинное целое»
success SetArrayString(long id,long idx,char str)	Присваивает значение индексу массива типа «строка»
char or long GetArrayElement(long id,char idx)	Возвращает значения обоих типов

tag,long id,long idx	элементов
success DelArrayElement(long tag,long id,long idx)	Удаляет один элемент массива
long GetFirstIndex(long tag,long id);	Возвращает индекс первого элемента
long GetLastIndex(long tag,long id);	Возвращает индекс последнего элемента
long GetNextIndex(long tag,long id,long idx)	Возвращает следующий индекс элемента массива
long GetPrevIndex(long tag,long id,long idx)	Возвращает предыдущий индекс элемента массива
success SetHashLong(long id,char idx,long value)	Присваивает значение элементу массива типа «длинное целое»
success SetHashString(long id,char idx,char value);	Присваивает значение элементу массива типа «строка»
long GetHashLong(long id,char idx)	Возвращает значение элемента типа «длинное целое»
char GetHashString(long id,char idx)	Возвращает значение элемента типа «строка»
success DelHashElement(long id,char idx)	Удаляет элемент ассоциативного массива
char GetFirstHashKey(long id)	Возвращает индекс первого элемента массива
char GetLastHashKey(long id)	Возвращает индекс последнего элемента массива
char GetNextHashKey(long id,char idx)	Возвращает индекс следующего элемента массива
char GetPrevHashKey(long id,char idx);	Возвращает индекс предыдущего элемента массива

### **long CreateArray(char name);**

Функция создает новый разряженный массив, в котором можно будет хранить данные обоих типов – как строковые, так и длинные целые.

Массив сохраняется в базе IDA как элемент Btree до тех пор, пока не будет принудительно удален из нее. Никаких ограничений на размер массива не налагается.

Каждый массив должен иметь свое уникальное имя (два массива с одинаковыми именами существовать не могут). Массивы имеют собственное пространство имен (то есть

можно создать метку или сегмент совпадающую с именем уже существующего массива и наоборот)

На имена наложены следующие ограничения – длина до 120 символов, может начинаться с цифры, но не должен содержать пробелов.

При успешном завершении функция возвращает идентификатор массива, в противном случае (массив с таким именем уже существует?) BADADDR.

Пример:

```
Message("0x%X \n",
CreateArray("MyArray")
);

0xFF000041
```

Операнд	Пояснения	
name	Имя массива	
Return	==return	Пояснения
	!=BADADDR	Идентификатор массива
	==0	Ошибка

### long GetArrayId(char name);

Функция возвращает идентификатор массива по его имени. Это позволяет не сохранять идентификаторы массивов, полученные при их создании, а обращаться к массивам по имени.

Например:

```
Message("0x%X \n",
CreateArray("MyArray")
);

Message("0x%X \n"
GetArrayId("MyArray")
);

DeleteArray(
GetArrayId("MyArray")
);

Message("0x%X \n"
GetArrayId("MyArray")
);

0xFF000041
0xFF000041
0xFFFFFFFF
```

Операнд	Пояснения	
name	Имя массива	
Return	==return	Пояснения
	!=BADADDR	Идентификатор массива
	==0	Ошибка

### success RenameArray(long id,char newname);

Функция позволяет изменить имя массива, заданного идентификатором. Обычно используется редко.

Например:

```
Message("0x%X \n",
CreateArray("MyArray")
);

0xFF000041

RemaneArray(
GetArrayId("MyArray"),
"MyNewname"
);

Message("0x%X \n"
GetArrayId("MyNewName")
);

0xFF000041
```

Операнд	Пояснения	
id	Идентификатор массива	
Newname	Новое имя массива	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

### void DeleteArray(long id);

Функция удаляет массив, заданный идентификатором. Необходимо помнить, что массивы, как элемент Btree хранятся в базе IDA то того момента, пока не будут удалены.

Это можно сделать, например, следующим образом:

```
DeleteArray(
GetArrayId("MyArray")
);
```

Операнд	Пояснения
id	Идентификатор массива

### success SetArrayLong(long id,long idx,long value);

Функция присваивает значение типа «длинное целое» элементу массива, заданного идентификатором.

Индекс массива выражается 32-битным целым числом. Разреженные массивы позволяют эффективно хранить данные, не резервируя памяти под несуществующие элементы.



Поэтому индексы не обязательно должны следовать один за другим. Так, например, массив может состоять всего из двух индексов, – скажем 0x0 и 0x10000, – при этом будет потрачено всего две ячейки памяти.

Необходимо помнить, что один и тот же индекс, одного и того же массива может хранить одновременно данные двух типов – как строковые, так и длинные целые и никакого «затирания при этом не происходит».

Пример использования:

```
SetArrayLong(
  GetArrayId("MyArray"),
  0x100,
  0x666);
```

Операнд	Пояснения	
id	Идентификатор массива	
idx	Индекс массива	
value	Присваиваемое значение типа «длинное целое»	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

#### **success SetArrayString(long id,long idx,char str);**

Функция присваивает значение типа «строка» элементу массива, заданного идентификатором.

Индекс массива выражается 32-битным целым числом. Разреженные массивы позволяют эффективно хранить данные, не резервируя памяти под несуществующие элементы.

Поэтому индексы не обязательно должны следовать один за другим. Так, например, массив может состоять всего из двух индексов, – скажем 0x0 и 0x10000, – при этом будет потрачено всего две ячейки памяти.

Необходимо помнить, что один и тот же индекс, одного и того же массива может хранить одновременно данные двух типов – как строковые, так и длинные целые и никакого «затирания при этом не происходит».

Пример использования:

```
SetArrayString(
  GetArrayId("MyArray"),
  0x100,
  "MyString");
```

Операнд	Пояснения	
id	Идентификатор массива	
idx	Индекс массива	
str	Присваиваемое значение типа «строка»	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

### **char or long GetArrayElement(long tag,long id,long idx);**

Функция служит для чтения обоих типов элементов разряженных массивов. Выбор интересующего типа осуществляется тегом tag.

Он может принимать следующие значения:

Определение		Значение
AR_LONG	'A'	Элемент типа «длинное целое»
AR_STR	'S'	Элемент типа «строка»

Запрашиваемый индекс должен быть инициализирован ранее, иначе функция вернет ошибку.

Пример использования:

```
SetArrayLong (
GetArrayId ("MyArray"),
0x100,
0x666);

SetArrayString (
GetArrayId ("MyArray"),
0x100,
"MyString");

Message ("%s \n0x%X\n",
GetArrayElement (AR_STR,
GetArrayId ("MyArray"),
0x100),
GetArrayElement (AR_LONG,
GetArrayId ("MyArray"),
0x100),
);

MYString
0x666
```

Операнд	Пояснения	
tag	==tag	Значение
	AR_STR	Элемент типа «строка»
	AR_LONG	Элемент типа «длинное целое»
id	Идентификатор массива	
idx	Индекс массива	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

### **success DelArrayElement(long tag,long id,long idx);**

Функция удаляет указанный тип элемента разряженного массива. Тип задается тегом tag, который может принимать следующие значения, перечисленные ниже в таблице:

Определение		Значение
AR_LONG	'A'	Элемент типа «длинное целое»
AR_STR	'S'	Элемент типа «строка»

Пример использования:

```
DelArrayElement (AR_LONG,
GetArrayId ("MyArray"),
0x100);
```

Операнд	Пояснения	
tag	==tag	Значение
	AR_STR	Элемент типа «строка»
	AR_LONG	Элемент типа «длинное целое»
id	Идентификатор массива	
idx	Индекс массива	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

### long GetFirstIndex(long tag,long id);

Функция возвращает индекс первого элемента разряженного массива. В отличие от «обычных» массивов, известных нам по языками C и Pascal, разряженные массивы могут начинаться с любого индекса, а не обязательно с нулевого.

«Первым» индексом разряженного массива будет индекс инициализированного элемента с наименьшим числовым значением.

Например:

```
auto a;
DeleteArray (GetArrayId ("MyArray"));
a=CreateArray ("MyArray");
SetArrayLong (a, 0x100, 0x666);
SetArrayLong (a, 0x77, 0x67);
SetArrayLong (a, 0x210, 0x777);
Message ("0x%X \n",
GetFirstIndex (AR_LONG, a)
);
DeleteArray (a);
```

0x77

Операнд	Пояснения	
tag	==tag	Значение
	AR_STR	Элемент типа «строка»
	AR_LONG	Элемент типа «длинное целое»
id	Идентификатор массива	
Return	==return	Пояснения
	!=BADADDR	Индекс первого элемента разряженного массива
	==BADADDR	Ошибка

### long GetLastIndex(long tag,long id);

Функция возвращает индекс последнего элемента разряженного массива. Обратите внимание, что число элементов разряженного массива обычно много меньше, чем индекс последнего из них.

Так, например, массив может состоять всего из трех элементов с индексами, скажем, (0x5, 0x777, 0x666777) – тогда функция GetLastIndex возвратит 0x666777, тогда как размер массива равен всего лишь трем.

К сожалению не предоставлено никаких функций, позволяющих узнать размер массива. Все что можно сделать это пройтись по цепочке элементов функциями GetNextIndex (GetPrevIndex).

Поэтому, вызов GetLastIndex используется очень редко, так как в нем особой нужды обычно и не бывает.

Пример использования:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetArrayLong(a,0x100,0x666);
SetArrayLong(a,0x77,0x67);
SetArrayLong(a,0x210,0x777);
Message("0x%X \n",
GetLastIndex(AR_LONG,a)
);
DeleteArray(a);
```

0x210

Операнд	Пояснения	
tag	==tag	Значение
	AR_STR	Элемент типа «строка»
	AR_LONG	Элемент типа «длинное целое»
id	Идентификатор массива	
Return	==return	Пояснения
	!=BADADDR	Индекс последнего элемента разряженного массива
	==BADADDR	Ошибка

### long GetNextIndex(long tag,long id,long idx);

Функция возвращает следующий индекс разряженного массива. Как уже было сказано выше, в разряженных массивах индексы не обязательно следуют друг за другом, а могут быть разделены «дырами» произвольного размера.

Поэтому, для «путешествия» по цепочке **инициализированных** элементов массива и была введена функция GetNextIndex.

Передаваемый текущий индекс (idx) не обязательно должен существовать в природе, - функция возвращает первый же инициализированный за ним элемент.

Это дает возможность отказаться от использования функции GetFirstIndex, поскольку GetNextIndex(.,0) ему полностью эквивалентен, что и показано на примере, приведенном ниже:

```
auto a,b;
b=0;
DeleteArray(GetArrayId("MyArray"));
```

```

a=CreateArray("MyArray");
SetArrayLong(a,0x100,0x666);
SetArrayLong(a,0x77,0x67);
SetArrayLong(a,0x210,0x777);
while(1)
{
b=GetNextIndex(AR_LONG,a,b);
if (b==-1) break;
Message("0x%X \n",b);
}
DeleteArray(a);

0x77
0x100
0x210

```

Операнд	Пояснения	
tag	==tag	Значение
	AR_STR	Элемент типа «строка»
	AR_LONG	Элемент типа «длинное целое»
id	Идентификатор массива	
idx	Индекс массива	
Return	==return	Пояснения
	!=BADADDR	Следующий индекс
	==BADADDR	Ошибка

### long GetPrevIndex(long tag,long id,long idx)

Функция возвращает предыдущий индекс разряженного массива. Как уже было сказано выше, в разряженных массивах индексы не обязательно следуют друг за другом, а могут быть разделены «дырами» произвольного размера.

Поэтому, для «путешествия» по цепочке **инициализированных** элементов массива и была введена функция GetPrevIndex.

Передаваемый текущий индекс (idx) не обязательно должен существовать в природе, - функция возвращает первый же предшествующий ему инициализированный элемент.

Это дает возможность отказаться от использования функции GetPrevIndex, поскольку GetPrevIndex(,-1) ему полностью эквивалентен, что и показано на примере, приведенном ниже:

```

auto a,b;
b=0;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetArrayLong(a,0x100,0x666);
SetArrayLong(a,0x77,0x67);
SetArrayLong(a,0x210,0x777);
while(1)
{
b=GetPrevIndex(AR_LONG,a,b);
if (b==-1) break;
Message("0x%X \n",b);
}
DeleteArray(a);

```

0x210  
0x100  
0x77

Операнд	Пояснения	
tag	==tag	Значение
	AR_STR	Элемент типа «строка»
	AR_LONG	Элемент типа «длинное целое»
id	Идентификатор массива	
idx	Индекс массива	
Return	==return	Пояснения
	!=BADADDR	Предыдущий индекс
	==BADADDR	Ошибка

## АССОЦИАТИВНЫЕ МАССИВЫ

### ОБ АССОЦИАТИВНЫХ МАССИВАХ

Ассоциативные массивы это маленькое чудо IDA. Большинству программистам возможно ранее никогда не приходилось сталкиваться ни с чем подобным. К сожалению, о них очень мало (ну совсем ничего) не сказано в контекстной помощи, поэтому будет не лишним остановиться на них и рассмотреть подробнее.

Один из распространенных языков, поддерживающим такие конструкции можно назвать Perl, который ввиду своей кросс - платформенности стал очень популярен среди разработчиков Internet – приложений. Но многие тысячи поклонников Си и Pascal могли и вовсе не слышать о нем, а уж тем более изыскивать время для досконального изучения.

Когда-то авторитет программиста зависел от числа освоенных им языков. Сегодня же ситуация перевернулась – разработчики предпочитают сосредоточиваться не только на одном языке, но даже компиляторы, зачастую совершенно не интересуясь, что происходит у «соседей».

К сожалению, ставшие популярными языки многие конструкции реализуют далеко не лучшим образом, если вообще реализуют это. Так случилось и с ассоциативными массивами.

Чем они отличаются от своих братьев? И какие возможности предоставляют при этом?

Уникальность ассоциативных массивов в том, что они индексируются **строковыми** значениями. Причем строковыми в буквальном смысле этого слова, - никакого хеширования или иного преобразования в численный вид не проводится.

Это позволят значительно упрощать многие алгоритмы, раньше выглядевшие громоздкими, - например, пусть нам требуется составить таблицу, перечисляющую имена сотрудников и получаемую ими зарплату.

Традиционно потребовалось бы создавать три массива, а потом обвязывать все это кодом, гарантирующим согласованность и непротиворечивость данных.

Насколько же проще решается задача с использованием ассоциативных массивов. Кроме этого можно придумать массу других примеров.

### АРХИТЕКТУРА АССОЦИАТИВНЫХ МАССИВОВ

Ассоциативные массивы представляют собой всего лишь подкласс обычных разреженных массивов, которые уже были подробно рассмотрены выше.

Ассоциативные массивы создаются вызовом функции `CreateArray`, точно так же как и обычные разреженные. Кроме того, любой массив по сути одновременно является и разреженным и ассоциативным. Точнее, массивов на самом деле все же несколько, но идентификатор (ID) у всех может быть один.

Поэтому к ассоциативным массивам применимы описанные выше функции `RenameArray` и `DeleteArray`.

Новым является набор менее чем из десятка функций работающий со строковыми индексами. Вообще же в терминологии ассоциативных массивов индексы называются **ключами**.

Каждый ключ представляет собой строковое значение без явных ограничений (во всяком случае в контексте помощи IDA на этот счет ничего не сказано) и может содержать один элемент типа «строка» или «длинное целое»

Обратите внимание, тогда как каждый индекс разреженного массива может одновременно содержать элементы двух типов, то каждый ключ ассоциативного массива ссылается лишь на одно значение того или иного типа.

В остальном же операции с элементами ассоциативных массивов ничем не отличаются от уже описанных выше. Собственно всеми манипуляциями заведут всего три функции.

Элемент можно присвоить, прочитать, и, наконец, удалить. На этом предоставляемые IDA возможности заканчиваются.

Более интересны операции, выполняемые над ключами. Поскольку теперь они представляют собой строковые значения, то может быть не совсем понятно с первого взгляда, как можно выполнить даже такую тривиальную операцию, как перечислить все элементы массива.

На самом деле для этого существуют традиционные функции `GetNext` и `GetPrev`. Очевидно, что они перечисляют ключи (то есть индексы) массива один за другим, но вот в каком порядке?

Вообще-то это не документировано, и поэтому стоит проектировать код скрипта так, чтобы его работоспособность не зависела от подобного рода деталей. Но как нетрудно убедиться, все текущие версии IDA упорядочивают список индексов в алфавитном порядке, не зависимо от очередности создания элементов.

Необходимо заметить, что некоторые языки упорядочивают индексы ассоциативных массивов именно в порядке их создания, и никем не гарантируется, что так не будет и в следующих версиях IDA (хотя, это конечно, очень маловероятно).

Однако, ваш скрипт не должен завистеть от этих деталей.

### **success SetHashLong(long id,char idx,long value);**

Функция присваивает значение элементу ассоциативного массива. Ассоциативные массивы выгодно отличаются от остальных тем, что могут индексироваться **строковыми** значениями!

В остальном же с ними могут использоваться те же подходы, что и для разреженных массивов.

Однако, в отличие от разреженных массивов один и тот же индекс не может содержать числовое и строковое значение одновременно.

Пример использования:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashLong(a,"Ivanov",0x66);
SetHashLong(a,"Cheputilo",0x77);
SetHashLong(a,"Alushta",0x67);
DeleteArray(a);
```

Операнд	Пояснения	
id	Идентификатор массива	
idx	Индекс массива (строковой!)	
value	Присваиваемое значение типа «длинное целое»	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

### success SetHashString(long id,char idx,char value);

Функция присваивает значение элементу ассоциативного массива. Ассоциативные массивы выгодно отличаются от остальных тем, что могут индексироваться *строковыми* значениями!

В остальном же с ними могут использоваться те же подходы, что и для разреженных массивов.

Однако, в отличие от разреженных массивов один и тот же индекс не может содержать числовое и строковое значение одновременно.

Пример использования:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashLong(a,"Ivanov","Patron");
SetHashLong(a,"Cheputilo","Mouse");
SetHashLong(a,"Alushta","Metro Station");
DeleteArray(a);
```

Операнд	Пояснения	
id	Идентификатор массива	
idx	Индекс массива (строковой!)	
value	Присваиваемое значение типа «строка»	
Return	==return	Пояснения
	==1	Успешное завершение
	==0	Ошибка

### long GetHashLong(long id,char idx);

Функция возвращает значение элемента ассоциативного массива типа «длинное целое»

Как уже говорилось выше один и тот же элемент не может содержать значения «строка» и «длинное целое» одновременно. Поэтому возникает возможность присвоить значение одного типа, а попытаться прочитать другого. Эта операция завершится корректно и типы будут автоматически преобразованы.

При этом функция GetHashLong всегда возвращает четыре первые байта, сколько бы строка ни занимала на самом деле. Если она была короче, - то остаток будет представлять собой мусор и содержать непредсказуемые значения.

Например:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
```



```

a=CreateArray("MyArray");
SetHashString(a,"Ivanov","Patron");
Message("%s \n",
GetHashLong(a,"Ivanov")
);
DeleteArray(a);

```

Patr

Необходимо помнить, что IDA учитывает регистр индексов. Так "Ivanov" и "ivanov" это два разных индекса и при попытке прочитать присвоенное значение функция возвратит ноль.

Например:

```

auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashLong(a,"Ivanov",0x66);
Message("%x \n",
GetHashLong(a,"ivanov")
);
DeleteArray(a);

```

0

Возникает неоднозначность, – то ли действительно возникла ошибка, то ли такое значение имеет элемент?

В остальном же никаких проблем с использованием этой функции не возникает. Например:

```

auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashLong(a,"Ivanov",0x66);
Message("%x \n",
GetHashLong(a,"Ivanov")
);
DeleteArray(a);

```

0x66

Операнд	Пояснения	
id	Идентификатор массива	
idx	Индекс массива (строковой!)	
Return	==return	Пояснения
	!=0	Значение элемента массива
	==0	Ошибка
		Значение элемента массива

**char GetHashString(long id,char idx);**

Функция возвращает значение элемента ассоциативного массива типа «строка»

Как уже говорилось выше один и тот же элемент не может содержать значения «строка» и «длинное целое» одновременно. Поэтому возникает возможность присвоить значение одного типа, а попытаться прочитать другого. Эта операция завершится корректно и типы будут автоматически преобразованы.

При этом функция `GetHashString` возвращает четыре первые байта, если ни один из них не является нулем, в противном случае весь отрезок до первого нуля.

Например:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashLong(a,"Ivanov",0x66776677);
Message("%s \n",
GetHashString(a,"Ivanov"));
DeleteArray(a);
```

`wfwf`

Необходимо помнить, что IDA учитывает регистр индексов. Так `"Ivanov"` и `"ivanov"` это два разных индекса и при попытке прочитать присвоенное значение функция возвратит пустую строку.

Например:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashString(a,"Ivanov",0x66);
Message("%s \n",
GetHashLong(a,"ivanov")
);
DeleteArray(a);
```

Возникает неоднозначность, – то ли действительно возникла ошибка, то ли такое значение имеет элемент?

В остальном же никаких проблем с использованием этой функции не возникает.

Например:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashString(a,"Ivanov","Patron");
Message("%s \n",
GetHashLong(a,"Ivanov")
);
DeleteArray(a);
```

`Patron`

Операнд	Пояснения	
<code>id</code>	Идентификатор массива	
<code>idx</code>	Индекс массива (строковой!)	
Return	<code>==return</code>	Пояснения
	<code>!=""</code>	Значение элемента массива

	=="	Ошибка
		Значение элемента массива

### success DelHashElement(long id,char idx);

Функция удаляет один элемент ассоциативного массива. Поскольку один и тот же элемент не может содержать значения «строка» и «длинное целое» одновременно, отпадает необходимость указывать теги (смотри описание функции DelArrayElement для разряженных массивов)

Необходимо помнить, что IDA учитывает регистр индексов. Так "Ivanov" и "ivanov" это два разных индекса и представляют собой два разных элемента.

В остальном же функция ничем не отличается от аналогичной, использующийся для удаления элементов разряженных массивов.

Например:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashString(a,"Ivanov","Patron");
Message("%s \n",
DelHashElement(a,"Ivanov")
);
DeleteArray(a);
```

1

Операнд	Пояснения	
id	Идентификатор массива	
idx	Индекс массива (строковой!)	
Return	==return	Пояснения
	==1	Операция выполнена успешно
	==0	Ошибка

### char GetFirstHashKey(long id);

Функция возвращает индекс первого элемента ассоциативного массива. Поскольку ассоциативные массивы индексируются строковыми значениями, то привычные для нас способы обращения к элементам не подходят.

Что бы понять принципы функционирования этой (и некоторых других) функций, необходимо познакомиться с техническими деталями архитектуры ассоциативных массивов.

Индексы ассоциативных массивов хранятся в списке, упорядоченном в алфавитной последовательности, не зависимо от порядка добавления в него новых элементов, что и доказывает следующий пример:

```
auto a;
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashLong(a,"Ivanov",0x66);
SetHashLong(a,"Cheputilo",0x77);
SetHashLong(a,"Alushta",0x67);
Message("%s \n",
```

```

GetFirstHashKey (a)
);
DeleteArray (a);

```

Alushta

Операнд	Пояснения	
id	Идентификатор массива	
Return	==return	Пояснения
	!=""	Индекс первого элемента массива
	=="	Ошибка

**char GetLastHashKey(long id);**

Функция возвращает индекс последнего элемента ассоциативного массива. Поскольку ассоциативные массивы индексируются строковыми значениями, то привычные для нас способы обращения к элементам не подходят.

Что бы понять принципы функционирования этой (и некоторых других) функций, необходимо познакомиться с техническими деталями архитектуры ассоциативных массивов.

Индексы ассоциативных массивов хранятся в списке, упорядоченном в алфавитной последовательности, не зависимо от порядка добавления в него новых элементов, что и доказывает следующий пример:

```

auto a;
DeleteArray (GetArrayId ("MyArray") );
a=CreateArray ("MyArray") ;
SetHashLong (a, "Ivanov", 0x66) ;
SetHashLong (a, "Cheputilo", 0x77) ;
SetHashLong (a, "Alushta", 0x67) ;
Message ("%s \n",
GetLastHashKey (a)
);
DeleteArray (a);

```

Ivanov

Операнд	Пояснения	
id	Идентификатор массива	
Return	==return	Пояснения
	!=""	Индекс последнего элемента массива
	=="	Ошибка

**char GetNextHashKey(long id,char idx);**

Функция возвращает индекс следующего элемента ассоциативного массива. Поскольку ассоциативные массивы индексируются строковыми значениями, то привычные для нас способы обращения к элементам не подходят.

Что бы понять принципы функционирования этой (и некоторых других) функций, необходимо познакомиться с техническими деталями архитектуры ассоциативных массивов.

Индексы ассоциативных массивов хранятся в списке, упорядоченном в алфавитной последовательности, не зависимо от порядка добавления в него новых элементов, и

функция `GetNextHashKey` последовательно возвращает элементы ассоциативного массива один за другим.

Передаваемый ей текущий индекс не обязательно должен существовать в природе – функция просматривает список всех элементов и возвращает следующий в алфавитном порядке за ним.

Это дает возможность отказаться от вызовов `GetFirstHashKey`, поскольку `GetNextHashKey(,")` будет его полным эквивалентом.

Например:

```

auto a,b;
b="";
DeleteArray(GetArrayId("MyArray"));
a=CreateArray("MyArray");
SetHashLong(a,"Ivanov",0x66);
SetHashLong(a,"Cheputilo",0x77);
SetHashLong(a,"Alushta",0x67);
for(;;){
b=GetNextHashKey(a,b);
if(b=="")break;
Message("%s\n",b);}
DeleteArray(a);

```

```

Alushta
Cheputilo
Ivanov

```

Операнд	Пояснения	
id	Идентификатор массива	
idx	Индекс массива (строковой!)	
Return	==return	Пояснения
	!="	Очередной индекс массива
	=="	Ошибка

#### **char GetPrevHashKey(long id,char idx);**

Функция возвращает индекс предыдущего элемента ассоциативного массива. Поскольку ассоциативные массивы индексируются строковыми значениями, то привычные для нас способы обращения к элементам не подходят.

Что бы понять принципы функционирования этой (и некоторых других) функций, необходимо познакомиться с техническими деталями архитектуры ассоциативных массивов.

Индексы ассоциативных массивов хранятся в списке, упорядоченном в алфавитной последовательности, не зависимо от порядка добавления в него новых элементов, и функция `GetPrevHashKey` последовательно возвращает элементы ассоциативного массива один за другим.

Передаваемый ей текущий индекс не обязательно должен существовать в природе – функция просматривает список всех элементов и возвращает предшествующий в алфавитном порядке за ним.

Это дает возможность отказаться от вызовов `GetLastHashKey`, поскольку `GetNextHashKey(-1)` будет его полным эквивалентом.

Например:

```

auto a,b;
b=-1;

```

```

DeleteArray (GetArrayId ("MyArray") );
a=CreateArray ("MyArray");
SetHashLong (a, "Ivanov", 0x66);
SetHashLong (a, "Cheputilo", 0x77);
SetHashLong (a, "Alushta", 0x67);
for (;) {
b=GetPrevHashKey (a, b) ;
if (b=="") break;
Message ("%s \n", b);}
DeleteArray (a);

```

Ivanov  
Cheputilo  
Alushta

Операнд	Пояснения	
Id	Идентификатор массива	
idx	Индекс массива (строковой!)	
Return	==return	Пояснения
	!="	Очередной индекс массива
	=="	Ошибка

## ОПЕРАЦИИ С ГЛОБАЛЬНЫМИ НАСТРОЙКАМИ

### МЕТОДЫ

Функция	Назначение
long GetLongPrm (long offset)	Возвращает длинный целый параметр
long GetShortPrm(long offset);	Возвращает короткий целый параметр
long GetCharPrm (long offset)	Возвращает байтовый параметр
success SetLongPrm (long offset,long value);	Задаёт длинный целый параметр
success SetShortPrm(long offset,long value);	Задаёт короткий целый параметр
success SetCharPrm (long offset,long value);	Задаёт байтовый параметр
success SetPrcsr (char processor);	Задаёт тип процессора для дизассемблирования
long Batch (long batch);	Устанавливает или снимает пакетный режим работы
char GetIdeaDirectory ()	Возвращает путь к директории, в которой расположена IDA

char GetInputFile ()	Возвращает имя дизассемблируемого файла
----------------------	---

```

long  GetLongPrm (long offset);
long  GetShortPrm(long offset);
long  GetCharPrm (long offset);
success SetLongPrm (long offset,long value);
success SetShortPrm(long offset,long value);
success SetCharPrm (long offset,long value);

```

IDA предоставляет возможность не только чтения глобальных установок из скриптов, но даже их модификации. Для этого служат целых шесть приведенных выше функций.

Это может показаться излишне сложным, но на самом деле все они сводятся к простому чтению \ записи непрерывного фрагмента памяти, в котором IDA и хранит свои настройки.

Три функции чтения GetLongPrm, GetShortPrm, GetCharPrm отличаются только возвращаемым значением. Первая возвращает длинное целое, вторая короткое целое и последняя строку.

В каком-то смысле все они взаимозаменяемы. Т.е. можно использовать GetLongPrm для чтения короткого целого, если потом «врачую» маскировать старшие биты возвращенного значения.

Обратите внимание, что GetCharPrm возвращает **не** строку, оканчивающуюся нулем, а только один байт.

Чтение же всей строки целиком приходится осуществлять пошагово в цикле байт за байтом. Или можно воспользоваться GetLongPrm, читая строку по четыре байта за раз (это удобнее да и быстрее).

'offset' это смещение внутри структуры, в которой IDA и хранит настройки. Ниже это будет подробно описано. А пока обратим внимание на то, что IDA не запрещает передавать функции произвольное смещение внутри структуры.

Это часто становится источником неочевидных ошибок. Обычно такое происходит, когда используются фиксированные смещения элементов структуры, вместо определенных IDA значений. Ввиду того, что в следующих версиях эта структура, скорее всего, не останется без изменений, то необходимо использовать только определения IDA, иначе за работоспособность скрипта трудно будет поручиться.

## НАСТОЙКИ IDA

Первые три байта хранят в себе слово 'IDA'. Хотя это и не сообщается в документации, но нетрудно убедиться, что это именно так и есть.

```

Message ("%s%s\n",
GetShortPrm(0) ,
GetCharPrm(2)
);

```

IDA

## INF VERSION

Содержит, переменную типа **Short** хранящую версию базы IDA. Например:

```

Message ("%x \n",

```

```
GetShortPrm(INF_VERSION)
);
```

22

## INF\_PROCNAME

Хранит **ВОСЬМИСИМВОЛЬНОЕ** имя выбранного типа процессора дизассемблируемого текста. Для процессоров серии 80x86 предусмотрены следующие соответствия:

Intel 8086	8086
Intel 80286 real	80286r
Intel 80286 protected	80286p
Intel 80386 real	80386r
Intel 80386 protected	80386p
Intel 80486 real	80486r
Intel 80486 protected	80486p
Intel Pentium real with MMX	80586r
Intel Pentium protected with MMX	80586p
Intel Pentium Pro (P6) with MMX	80686p
Intel Pentium II	p262
AMD K6-2 with 3DNow!	K62p3
Intel Pentium III	p3ntel

Наиболее разумным (и быстрым) способом извлечения этого поля, вероятно, окажется чтение его с помощью GetLongPrm как показано ниже:

```
Message("%s%s \n",
GetLongPrm(INF_PROCNAME),
GetLongPrm(INF_PROCNAME+4)
);
```

p3ntel

**ПРИМЕЧАНИЕ:** Это поле может только считываться. Все попытки его модификации посредством SetXXXPrm будут проигнорированы.

## INF\_LFLAGS

Это **однобайтовое** поле хранит флаги, определяемые по умолчанию в IDP модуле для конкретной модели процессора, которые могут принимать следующие значения:

LFLG\_PC\_FPP (0x1)

Декодировать инструкции с плавающей запятой для арифметического сопроцессора.

LFLG\_PC\_FLAT (0x2)

Плоская модель памяти



Обратите внимание, что изменение последнего параметра может повлечь за собой непредсказуемую работу дизассемблера и привести к неверному анализу исследуемого файла!

Пример использования:

```
auto a;
a=GetCharPrm(INF_LFLAGS);
Message("%x \n", a);
if (a & LFLG_PC_FPP)
    Message ("Decode FPP \n");
if (a & LFLG_PC_FLAT)
    Message ("FLAT MODEL \n");
```

```
1
Decode FPP
```

## INF\_DEMNames

**Однобайтовое** поле, определяющие каким образом IDA будет «замангловать» имена. Безболезненно может как считываться, так и модифицироваться.

### DEMNAME\_CMNT (0):

Отображать заманглованные имена как комментарии. Например:

```
SetCharPrm(INF_DEMNames, DEMNAME_CMNT);

.text:00403E79 ?sputc@streambuf@@QAEHH@Z proc near
; streambuf::sputc(int)
```

### DEMNAME\_NAME (1)

Замангловать в имена. Например:

```
SetCharPrm(INF_DEMNames, DEMNAME_NAME);

.text:00403E79 public: int __thiscall streambuf::sputc(int) proc
near
```

### DEMNAME\_NONE (2)

Не замангловать и представлять имена как есть.

```
SetCharPrm(INF_DEMNames, DEMNAME_NONE);

.text:00403E79 ?sputc@streambuf@@QAEHH@Z proc near
```

При установке нового значения IDA автоматически начнет реанализ и внесет все изменения в дизассемблируемый текст.

## INF\_FILETYPE

Это поле содержит **короткое целое**, хранящее тип дизассемблируемого файла. Видимо не используется IDA, поскольку может безболезненно модифицироваться произвольными значениями, что не нарушает работоспособности.

Так же следует учитывать, что в версии 3.84 функция возвращает **неверные** значения. Дело в том, что IDC.IDC не был изменен, тогда как были расширены типы файлов, начиная с середины таблицы, от чего все типы «посыпались».

MS-DOS exe файл стал определяться как программный файл PalmPilot, что хотя и не нарушало работоспособности IDA, но не позволяло определить тип текущего файла.

В IDC.IDC содержится ссылка на файл 'core.hpp'. На самом деле это опечатка и нужно смотреть 'ida.hpp', входящий в IDA SDK. Там мы обнаружим прелюбопытную структуру, описывающую типы поддерживаемых файлов.

Сравнив ее с IDC.IDC можно обнаружить различие, которое показано ниже:

f_EXE_old,	FT_EXE_OLD
f_COM_old,	FT_COM_OLD
f_BIN,	FT_BIN
f_DRV,	FT_DRV
f_WIN,	FT_WIN
f_HEX,	FT_HEX
f_MEX,	FT_MEX
f_LX,	FT_LX
f_LE,	FT_LE
f_NLM,	FT_NLM
f_COFF,	FT_COFF
f_PE,	FT_PE
f_OMF,	FT_USER
f_SREC,	FT_OMF
f_ZIP,	FT_SREC
f_OMFLIB,	FT_ZIP
f_AR,	FT_OMFLIB
f_LOADER,	FT_AR
f_ELF,	FT_LOADER
f_W32RUN,	FT_ELF
f_AOUT,	FT_W32RUN
f_PRC,	FT_AOUT
	FT_PRC
	FT_EXE
	FT_COM
	FT_AIXAR

Что бы функция возвращала правильный результат, необходимо исключить тип **FT\_USER** и перенумеровать остаток списка.

Расшифровка всех значений приведена ниже:

FT_EXE_OLD	MS DOS EXE файл
FT_COM_OLD	MS DOS COM файл
FT_BIN	Двоичный файл (дамп ROM например)

FT_DRV	MS DOS драйвер (drv или sys)
FT_WIN	New Executable (NE)
FT_HEX	Intel Hex Object File
FT_MEX	MOS Technology Hex Object File
FT_LX	Linear Executable (LX)
FT_LE	Linear Executable (LE)
FT_NLM	Netware Loadable Module (NLM)
FT_COFF	Common Object File Format (COFF)
FT_PE	Portable Executable (PE)
FT_USER	Файл, загруженный посредством загрузчика IDP
FT_OMF	Object Module Format
FT_SREC	R-records
FT_ZIP	ZIP file (никогда не бывает загружен в базу IDA)
FT_OMFLIB	Библиотека OMF Модулей
FT_AR	ar library
FT_LOADER	Файл загружен посредством LOADER DLL
FT_ELF	Executable and Linkable Format (ELF)
FT_W32RUN	Watcom DOS32 Extender (W32RUN)
FT_AOUT	Linux a.out (AOUT)
FT_PRC	PalmPilot программный файл
FT_EXE	MS DOS EXE File
FT_COM	MS DOS COM File
FT_AIXAR	AIX ar library

Пример использования:

```
Message ("%d \n",
GetShortPrm(INF_FILETYPE)
);
```

23

## INF\_OSTYPE

**Короткое целое** хранящее тип операционной системы для загруженного файла (не среды запуска дизассемблера!)

Должна принимать следующие значения:

OSTYPE_MSDOS	0x0001	MS-DOS
OSTYPE_WIN	0x0002	MS Windows
OSTYPE_OS2	0x0004	OS/2
OSTYPE_NETW	0x0008	Novell NetWare

Да, именно должна, ибо MS-DOS файлы возвращают ноль, а не единицу и, следовательно, OSTYPE\_MSDOS не работает.

Пример использования:

```
Message ("%d \n",
GetShortPrm(INF_OSTYPE)
);
```

INF\_APPTYPE

**Короткое целое**, содержащее информацию о типе дизассемблируемого приложения. Часть полей (APPT\_CONSOLE, APPT\_GRAPHIC, APPT\_1THREAD, APPT\_MTHREAD) инициализируются FLIRT. Если исследуемой программе не соответствует ни одна библиотека сигнатур и FLIRT не сработал, то все вышеперечисленные поля будут содержать нулевые значения.

Тип приложения (EXE\DLL\DRIVER) не актуален для MS-DOS программ, как и разрядность (16 или 32 бит). В этом случае функция всегда возвращает нулевое значение.

APPT_CONSOLE	0x0001	Console
APPT_GRAPHIC	0x0002	Graphics
APPT_PROGRAM	0x0004	EXE
APPT_LIBRARY	0x0008	DLL
APPT_DRIVER	0x0010	DRIVER
APPT_1THREAD	0x0020	Singlethread
APPT_MTHREAD	0x0040	Multithread
APPT_16BIT	0x0080	16 bit application
APPT_32BIT	0x0100	32 bit application

Пример использования:

```
Message ("%x \n",
GetShortPrm(INF_APPTYPE)
);
```

104

INF\_START\_SP

**Длинное целое**, содержащее значение регистра SP (ESP) при запуске программы. Для получения этой информации IDA читает соответствующие поля заголовка файла. В противном случае (например, для сом или дампов памяти) она устанавливает SP на верхушку сегмента, то есть присваивает ему значение -1.

Для бинарных файлов и дампов памяти это оказывается не всегда справедливо (в самом деле, откуда IDA может знать значение указателя стека в каждом конкретном случае) Тогда рекомендуется установить требуемое значение вручную, функцией SetLongPrm.

Однако, обычно точное значение SP (ESP) не критично и в общем случае не влияет на правильность дизассемблирования кода.

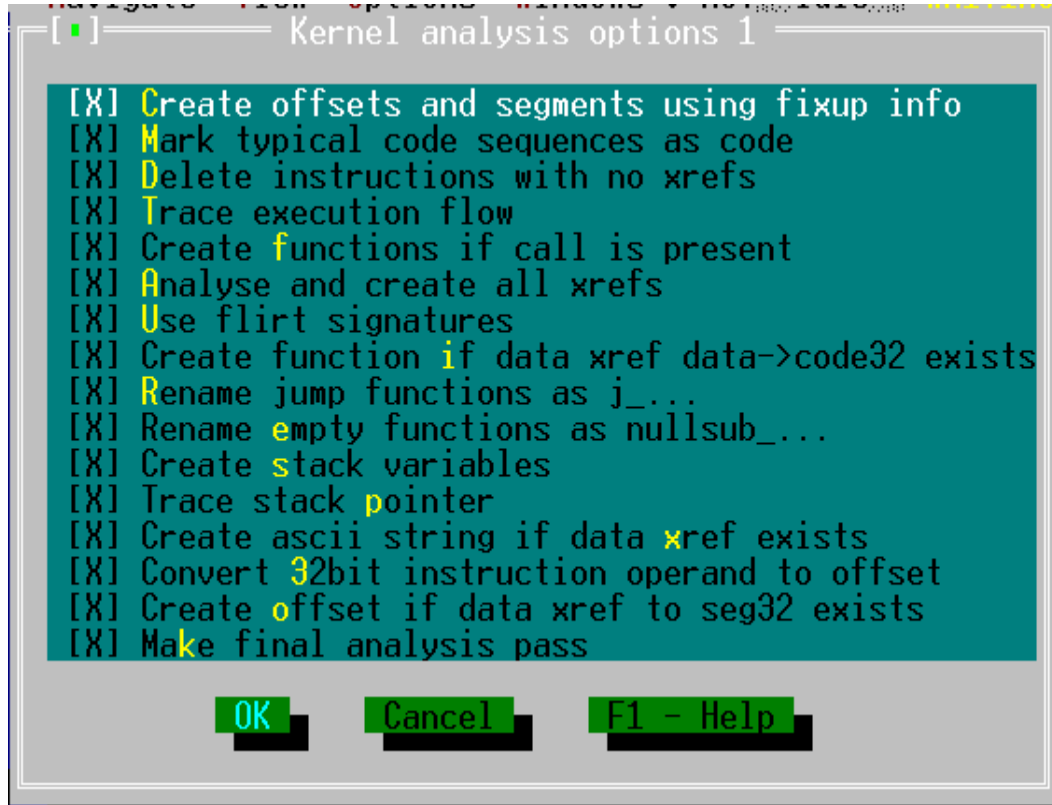
Пример использования:

```
Message ("%x \n",
GetShortPrm(INF_START_SP)
);
```

ffff

## INF\_START\_AF

Это поле содержит короткое целое, управляющие настройками анализатора IDA. Иначе к ним можно добраться через меню «Options\ Analysis options\ Kernel analyser options 1»



Все они доступны как для чтения, так и для модификации. Назначение битов флагов приведены ниже.

AF_FIXUP	0x0001	Создавать сегменты и смещения, используя информацию из таблицы перемещаемых элементов
AF_MARKCODE	0x0002	Автоматически преобразовывать типичные последовательности инструкций в код
AF_UNK	0x0004	Удалять инструкции без ссылок
AF_CODE	0x0008	Трассировать выполнение
AF_PROC	0x0010	Автоматически создавать функции
AF_USED	0x0020	Поверхностный анализ программы
AF_FLIRT	0x0040	Использовать FLIRT сигнатуры
AF_PROCPTR	0x0080	Создавать функции в 32-битном сегменте, если это ссылка на сегмент данных

AF_JFUNC	0x0100	Переименовывать jump-функции как j_...
AF_NULLSUB	0x0200	Переименовывать пустые функции как nullsub_...
AF_LVAR	0x0400	Создавать стековые переменные
AF_TRACE	0x0800	Отслеживать указатель стека
AF_ASCII	0x1000	Автоматически создавать строки
AF_IMMOFF	0x2000	Преобразовывать операнды 32-инструкций в смещения
AF_DREFOFF	0x4000	Преобразовывать 32-данные в смещения
AF_FINAL	0x8000	Сворачивать все unexplored регионы

### AF\_FIXUP

если этот бит установлен, то IDA будет использовать информацию из таблицы перемещаемых элементов и представлять соответствующие непосредственные операнды в виде смещений или сегментов.

Например:

Код	AF_FIXUP == 1	AF_FIXUP == 0
B8 01 00	mov ax, seg dseg	mov ax, 1001h
8E D8	mov ds, ax	mov ds, ax

Значение перемещаемого элемента, выделенного красным цветом, равно 0x1. В любом случае IDA автоматически суммирует его с адресом загрузки (в нашем примере 0x10000).

Если флаг AF\_FIXUP установлен, то IDA преобразует непосредственный операнд в сегмент, в противном же случае оставит его без изменений.

### AF\_MARKCODE

Установка этого флага приведет к тому, что IDA будет находить все типичные для выбранного процессора последовательности инструкций и будет преобразовывать их в код, даже если на него отсутствуют явные ссылки.

Такой прием не совсем безгрешен, но позволяет заметно поднять качество дизассемблирования и переложить часть рутинной работы на плечи дизассемблера.

Например, для 80x86 процессоров типичной последовательностью инструкций будет инициализация регистра BP (EBP) при входе в процедуру.

```
.text:00401020          push    ebp
.text:00401021 8B EC    mov     ebp, esp
```

Обратите внимание, что этот механизм запускается только во время загрузки файла и динамическое его изменение во время работы дизассемблера будет проигнорировано.

### AF\_UNK

Этот флаг будучи установленным приводит к тому, что IDA будет каждый раз при пометке инструкции (инструкций) как unexplored автоматически

отслеживать все потерянные перекрестные ссылки, пометчая соответствующие регионы как unexplored.

## AF\_CODE

IDA умеет трассировать следование инструкций, отслеживая условные переходы и вызовы процедур. Например, если встретится:

```
seg000:22C3 E8 5F 00          call    sub_0_2325
```

то можно быть уверенным, что IDA преобразует в инструкции и код, находящийся по смещению 0x2325. В противном случае (если бит AF\_CODE сброшен) это выполнено не будет. Более того, при загрузке IDA не дизассемблирует ни одной инструкции, предоставляя это пользователю сделать самостоятельно.

Этот флаг имеет смысл сбрасывать всякий раз, когда IDA неправильно отслеживает ссылки или же вам нужно изучить только отдельно взятый фрагмент кода и дизассемблировать весь файл не к чему.

## AF\_PROC

Автоматически создавать функции на месте вызова инструкцией call. В противном случае функции будут создаваться только для библиотечных процедур. Например:

AF_PROC == 0	AF_PROC == 1
<pre>Seg00:0124 call    loc_0_284 Seg00:0284 loc_0_284: seg00:0284 push   ds seg00:0285 mov    ax, 3500h seg00:0288 int    21h seg00:028A ret</pre>	<pre>Seg00:0124 call    loc_0_284 Seg00:0284 sub_0_284 proc near seg00:0284 push   ds seg00:0285 mov    ax, 3500h seg00:0288 int    21h seg00:028A ret seg00:02C6 sub_0_284 endp</pre>

## AF\_USED

В документации на IDA сообщается, что сброс этого бита приводит к тому, что IDA выполняет поверхностный анализ программы. То есть примитивное дизассемблирование, без создания перекрестных ссылок и дополнительных проверок.

Однако, практически значение этой опции никак не влияет на дизассемблируемый текст и в обоих случаях получаются идентичные листинги.

## AF\_FLIRT

Уникальная FLIRT технология позволяет IDA определять имена библиотечных функций наиболее популярных компиляторов по их сигнатурам. Сравните два примера:

AF_FLIRT == 1	AF_FLIRT == 0
<pre>dseg:039A push offset aHelloSailor dseg:039D call    _printf</pre>	<pre>dseg:039A pushoffset aHelloSailor dseg:039D call    sub_0_1035</pre>

dseg:03A0 pop cx dseg:03A1 retn	dseg:03A0 pop cx dseg:03A1 retn
------------------------------------	------------------------------------

## AF\_PROCPTR

Установка этого флага приведет к тому, что IDA будет проверять все перекрестные ссылки из 32-разрядного сегмента данных в сегмент кода. Если ссылка указывает на машинную инструкцию, то IDA автоматически преобразует ее в код и создаст на этом месте функцию.

Например:

AF_PROCPTR == 1	AF_PROCPTR == 0
.data:004085E0 dd <b>offset sub_0_405AAC</b>	.data:004085E0 dd 405AACh
.text:00405AAC sub_0_405AAC proc near	.text:00405AAC db 55h
.text:00405AAC push ebp	.text:00405AAD db 8Bh
.text:00405AAD mov ebp, esp	.text:00405AAE db 0ECh

Данный метод не безгрешен и в некоторых случаях может приводить в к ошибкам (тем более возможно предположить умышленное противодействие автора дизассемблируемого текста против IDA) поэтому иногда его приходится отключать.

## AF\_JFUNC

Установка этого флага приведет к тому, что IDA будет переименовывать функции, состоящие из одной только команды `jmp somewhere` в `j_somewhere`. Это заметно улучшает читабельность листинга и ускоряет анализ алгоритма его работы.

AF_JFUNC == 1	AF_JFUNC == 0
seg000:22DD <b>j_MyJumpTrg</b> proc near	seg000:22DD sub_0_22DD proc near
seg000:22DD jmp short MyJumpTrg	seg000:22DD jmp short MyJumpTrg
seg000:22DD <b>j_MyJumpTrg</b> endp	seg000:22DD sub_0_22DD endp

## AF\_NULLSUB

Установка этого флага приведет к тому, что IDA будет переименовывать «пустые», то есть состоящие только из одной инструкции возврата, процедуры в `nullsub_xx`.

Это облегчает читабельность и восприятие листинга, а так же ускоряет анализ исследуемого текста дизассемблера.

AF_NULLSUB == 1	AF_NULLSUB == 0
seg000:22DF <b>nullsub_1</b> proc near	seg000:22DF sub_0_22DF proc near
seg000:22DF retn	seg000:22DF retn
seg000:22DF <b>nullsub_1</b> endp	seg000:22DF sub_0_22DF endp

## AF\_LVAR

Механизм отслеживания текущего значения регистра SP (ESP) дает возможность поддержки локальных переменных. То есть тех, что лежат в стеке с отрицательным смещением относительно BP (EBP).

Это становится невероятно полезным при дизассемблировании кода,



сгенерированного оптимизирующими компиляторами, которые уже не опираются на BP (EBP), а адресуют локальные переменные относительно стекового регистра ESP. Это приводит к тому, что невозможно понять к какой именно переменной обращается та или иная инструкция, до тех пор пока не будет вычислено значение указателя стека в конкретной точке кода.

IDA взяла на себя эту рутину работу и поддерживает оба типа стековых переменных самостоятельно.

AF_LVAR == 1	AF_LVAR == 0
<code>.text:0040112A mov ecx, [esp+40h+var_1C]</code>	<code>.text:0040112A mov ecx, [esp+24h]</code>

#### AF\_TRACE

Установка этого флага заставляет IDA отслеживать значение регистра указателя стека в каждой точке кода. Главным образом это необходимо для поддержки локальных переменных (см. AF\_LVAR)

AF_PROCPTR == 1	AF_PROCPTR == 0
<code>dseg:187A off_0_187A dw offset loc_0_B45</code>	<code>dseg:187A word_0_187A dw 0B45</code>
<code>dseg:0B45 mov dx, 183Ch</code>	<code>dseg:0B45 mov dx, 183Ch</code>

#### AF\_ASCII

IDA может автоматически создавать строки, если элемент на который указывает ссылка состоит более чем из четырех символом ASCII для 16-сегмента (и шестнадцати символов в остальных случаях).

Стиль строки определяется настройками о которых будет сказано ниже.

#### AF\_IMMOFF

Этот флаг имеет смысл только для 32-разрядных сегментов. Если он установлен, то IDA будет преобразовывать 32-разрядные операнды в смещения. Для этого необходимо, что бы операнд был больше минимально возможного адреса загрузки 0x10000.

Значительно облегчает дизассемблирование 32-разрядных приложений, автоматически корректно распознавая большинство смещений и указателей. Поскольку большинство приложений редко оперируют подобными величинами, то вероятность ложных срабатываний (то есть ошибочного преобразования константы в смещение) относительно невелика.

AF_IMMOFF == 1	AF_IMMOFF == 0
<code>.text:00401000 push offset aHeloSailor</code>	<code>.text:00401000 push 408040h</code>
<code>.text:00401005 mov ecx, offset ord_0_408900</code>	<code>.text:00401005 mov ecx, 408900h</code>

#### AF\_DREFOFF

Если этот флаг установлен, то IDA будет автоматически пытаться преобразовать в смещения все двойные слова, содержащие ссылки из 32-разрядных сегментов.

Преобразование в общем случае осуществляется успешно, если содержимое двойного слова больше, чем 0x10000

AF_DREFOFF == 1	AF_DREFOFF == 0
<code>.data:00408330 off_0_408330 dd offset unk_0_408980 ; DATA XREF: .text:00404758o</code>	<code>.data:00408330 dword_0_408330 dd 408980h</code>

Поясним этот пример. Допустим, в 32-сегменте кода встретится следующая инструкция:

```
.text:00404758          mov     eax, 408330h
```

Если флаг AF\_IMMOFF (см. выше) установлен, то константа 0x408440 будет автоматически преобразована в смещение, так как 0x408440 > 0x10000.

По этому смещению находится следующая ячейка:

```
.data:00408330 dword_0_408330 dd 408980h
```

Поскольку 0x408980 больше 0x10000, то, скорее всего, оно представляет собой смещение, в которое и может быть преобразовано, если флаг AF\_DREFOFF будет установлен.

## AF\_FINAL

Если этот флаг установлен, то дизассемблер в последнем проходе анализа автоматически преобразует все байты, помеченные как `unexplored`, в данные или инструкции.

Правила, по которым происходит это преобразование, не документированы и меняются от версии к версии. В идеале IDA должна была бы практически во всех случаях «угадать» чем является каждый элемент – данными или инструкцией. Однако, на практике она часто допускает ошибки, особенно `com` файлах, где данные и код могут быть сложным образом перемешаны.

Для win32 файлов с отдельными сегментами кода и данных, эта проблема отсутствует.

Рекомендуется сбрасывать этот флаг (по умолчанию он установлен). И вот почему – рассмотрите пример, приведенный ниже. Очевидно, что по адресу `seg000:210D` расположены строки:

```
seg000:210D aDir          db  '..',0
seg000:2110 aMask        db  '*.*',0
```

Но IDA, не найдя на них ссылок (поскольку невозможно для 16-разрядных приложений отличить смещения от констант) превратила их в малоосмысленный массив.

Очевидно, что программа была дизассемблирована *не правильно*. Поэтому лучше не полагаться на автоматически алгоритмы IDA, а исследовать `unexplored` байты самостоятельно.

**ЗАМЕЧАНИЕ:** для некоторых типов файлов (например, PE) значение этого флага в ряде случаев игнорируется и остаются `unexplored` байты.

AF_FINAL == 1	AF_FINAL == 0
<code>seg000:210D db 2 dup(2Eh), 0, 2Ah, 2Eh, 2Ah, 0</code>	<code>seg000:210D db 2Eh ; .</code> <code>seg000:210E db 2Eh ; .</code> <code>seg000:210F db 0 ;</code> <code>seg000:2110 db 2Ah ; *</code> <code>seg000:2111 db 2Eh ; .</code> <code>seg000:2112 db 2Ah ; *</code> <code>seg000:2113 db 0 ;</code>

## INF\_START\_IP

Это длинное поле содержит в себе значение регистра IP (EIP) при запуске программы. Для бинарных файлов не имеет смысла и возвращает ошибку (BADADDR). В остальных случаях IDA извлекает необходимую информацию из соответствующих полей заголовка файла или же эмулятора загрузчика (например, для com-файлов).

Это поле доступно как на чтение, так и на запись. Однако, модификация начального значения IP (EIP) не приведет к изменению точки входа (Entry point) файла (для этого необходимо изменить значение INF\_BEGIN\_EA)

Пример использования:

```
Message ("%x \n",  
GetLongPrm (INF_START_IP)  
);
```

401020

## INF\_BEGIN\_EA

Это длинное поле хранит линейный адрес точки входа в файл. Доступно для модификации, однако, все изменения не возымеют никакого эффекта для уже существующей точки входа.

Пример использования:

```
Message ("%x \n",  
GetLongPrm (INF_BEGIN_EA)  
);
```

401020

## INF\_MIN\_EA

Это длинное поле хранит минимальный линейный адрес, используемый программой.

Пример использования:

```
Message ("%x \n",  
GetLongPrm (INF_MIN_EA)  
);
```

401000

## INF\_MAX\_EA

Это длинное поле хранит самый старший адрес, используемый программой. Никогда не бывает равно минус единице (не смотря на то, что это число присутствует в IDC.IDC). Не зависимо от того, загружен файл как бинарный или нет, всегда возвращается максимальный адрес, встретившийся в программе.

```
Message ("%x \n",  
GetLongPrm (INF_MAX_EA)  
);
```

134EA

### INF\_LOW\_OFF

Это длинное поле хранит самый младший из возможных адресов, в котором непосредственный операнды могут трактоваться как тип void. Иными словами, начиная с этой величины, IDA будет предполагать, что операнд может являться смещением, и поэтому будет выделять его красным цветом, привлекая к нему внимание пользователя.

Рассмотрим это на следующем примере: пусть у в исследуемом файле минимальный из возможных адресов равен 0x100. Следовательно, можно предположить, что все операнды, входящие в диапазон от 0 до 0xFF окажутся константами, а выше 0xFF с равной степенью вероятности могут быть как смещениями, так и константами.

IDA всегда использует беззнаковые значения операндов. Поэтому [BP-2] будет трактоваться как 0xFFFFE, а не -2.

Допустима модификация этого поля, в том числе и интерактивно, через меню «~Options\Text representation\void's low limit». По умолчанию IDA использует минимальный линейный адрес. Как правило, первым располагается сегмент кода. Если достоверно известно, что программа не содержит на него ссылок, то значение INF\_LOW\_OFF можно изменить таким образом, что бы оно указывало на сегмент данных.

```
Message ("%x \n",  
GetLongPrm (INF_LOW_OFF)  
);
```

401000

### INF\_HIGH\_OFF

Это длинное поле хранит самый старший из возможных адресов, до которого непосредственные операнды могут трактоваться как тип void. Подробнее об этом было сказано в описании поля INF\_LOW\_OFF

По умолчанию INF\_HIGH\_OFF равно наибольшему адресу, занимаемому программой. Часто этого оказывается недостаточным для тех программ, что организуют буфера за пределами области статических переменных.

Рассмотрим это на примере типичного EXE файла (SMALL модель памяти - Сегмент стека и заголовок для упрощения не показаны) Все переменные, кроме тех, что инициализируются на стадии компиляции, останутся «не видимы» для IDA. Она посчитает ссылки на них константами, а не смещениями.

Поэтому необходимо изменить значение поля INF\_HIGH\_OFF вручную.

Файл на диске	Исполняемая программа	
КОД	Сегмент кода	
СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ	Сегмент данных	Статические переменные
	Сегмент данных	Динамические переменные

### INF\_MAXREF

Это длинное поле хранит максимальную глубину перекрестных ссылок. По умолчанию 10. Это значение можно изменить через меню (~Options\Cross references)

Right margin 80  
Cross reference depth 0x10

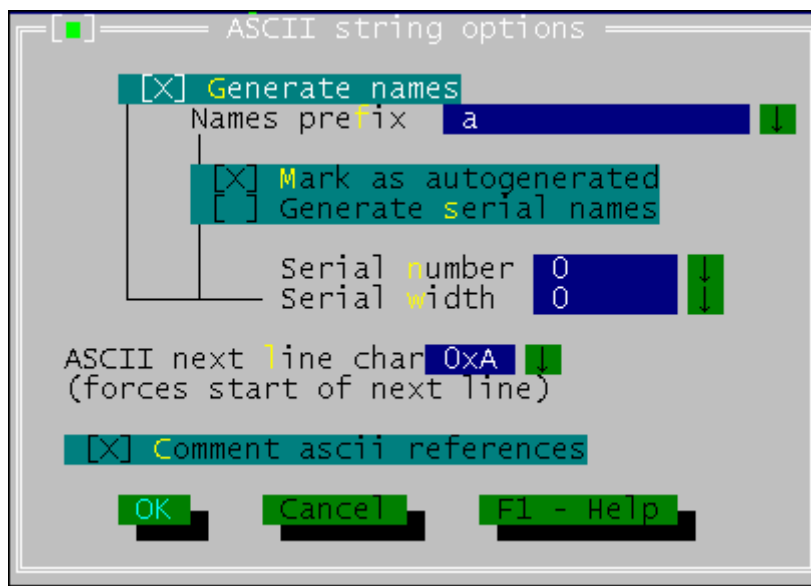
Пример:

```
Message ("%x \n", GetLongPrm (INF_MAXREF));  
10
```

## INF ASCII BREAK

Это однобайтовое поле содержит в себе символ переноса конца строки. Он не будет использоваться IDA при генерации файлов отчета или при выводе на экран. Не влияет он и на трактовку спецификатора 'n'. Единственное его назначение форматирование сток в дизассемблируемом листинге. (Ниже это будет показано на конкретном примере для большей ясности)

Поле может, как читаться, так и модифицироваться. Изменения вступают в силу немедленно, автоматически переформатируя все строки в дизассемблируемом тексте. Интерактивно это значение можно изменить, вызвав следующий диалог командой меню «~Options\ ASCII strings options». «ASCII next line char» и есть то поле, о котором сейчас идет речь.



Пример использования:

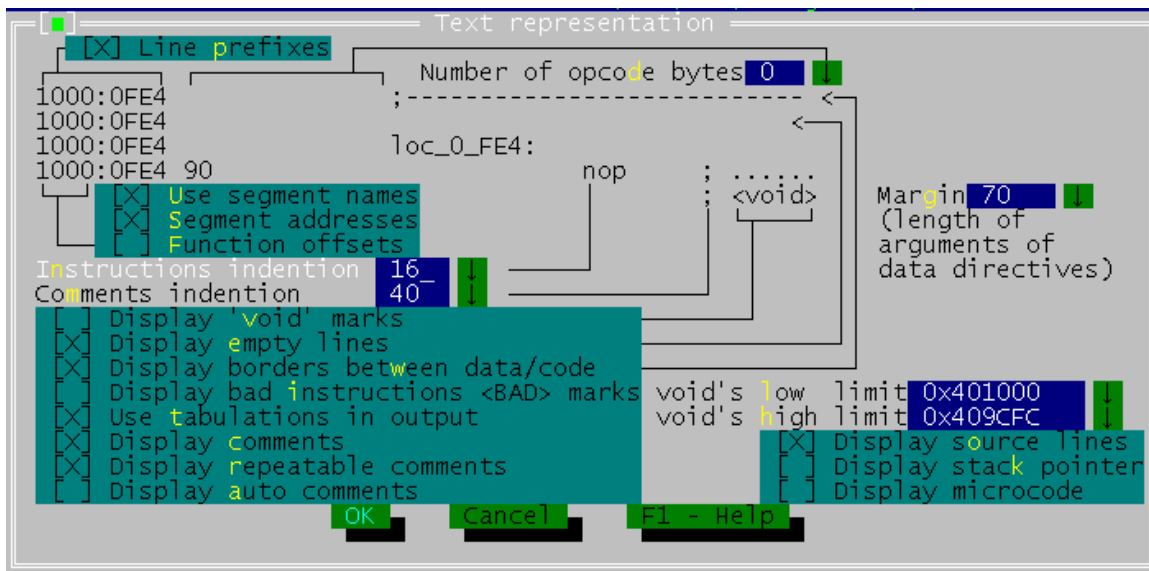
```
Message ("0x%X \n", GetCharPrm (INF_ASCII_BREAK));  
0xA  
  
.rdata:00407384 aRuntimeErrorPr db 'Runtime Error!', 0Ah  
.rdata:00407384 db 0Ah  
.rdata:00407384 db 'Program: ', 0  
  
SetCharPrm (INF_ASCII_BREAK, 0);  
Message ("0x%X \n", GetCharPrm (INF_ASCII_BREAK));  
0x0  
  
.rdata:00407384 aRuntimeErrorPr db 'Runtime Error!', 0Ah, 0Ah, 'Program: ', 0
```

## INF\_INDENT

Это однобайтовое поле содержит отступ, которым IDA предваряет все инструкции в дизассемблируемом листинге.

INF_INDENT == 0x10	INF_INDENT == 0
<code>SetCharPrm(INF_INDENT, 0x10);</code>	<code>SetCharPrm(INF_INDENT, 0x0);</code>
<pre> seg000:22C0 seg000:22C0 seg000:22C0 <b>seg000:22C0 start</b> seg000:22C0 seg000:22C3 seg000:22C6 seg000:22C9 seg000:22CC seg000:22CF seg000:22D2           </pre>	<pre> <b>seg000:22C0 public start</b> <b>seg000:22C0 start proc near</b> seg000:22C0 call sub_0_22DD seg000:22C3 call sub_0_2325 seg000:22C6 call sub_0_235B seg000:22C9 call sub_0_2374 seg000:22CC call sub_0_23B6 seg000:22CF call sub_0_23F8 seg000:22D2 jnz loc_0_22DA           </pre>

По умолчанию отступ равен 0x10, однако, это значение можно изменять, форматруя листинг по своему вкусу. Для этого необходимо воспользоваться функцией `SetCharPrm(INF_INDENT, nn)` или интерактивно через меню «~Options\Text representation\Instructions indention»



## INF\_COMMENT

Это однобайтовое поле содержит отступ, которым IDA предваряет все комментарии. По умолчанию равно 40. Может быть изменено по вкусу пользователя как интерактивно («~Options\Text representation\Comments indention»), так и с помощью функции `SetCharPrm(INF_COMMENT, nn)`

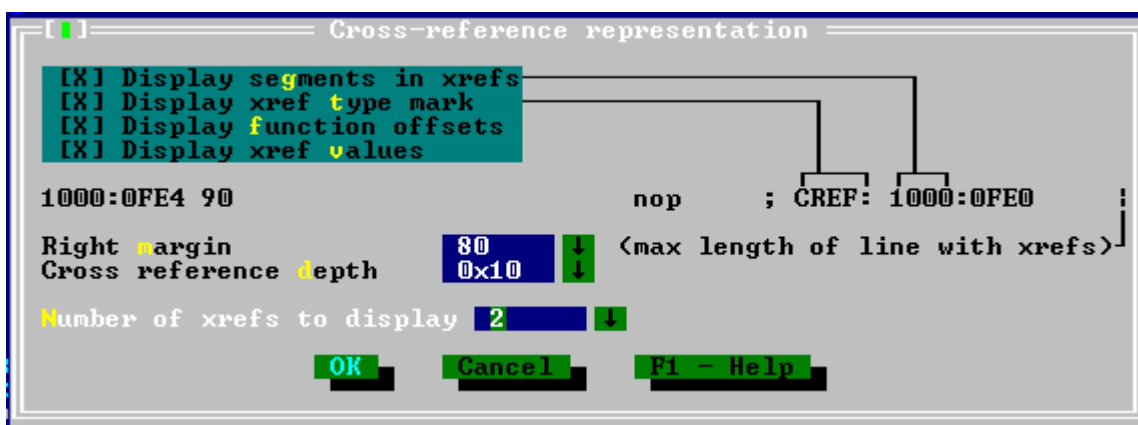
INF_COMMENT == 40	INF_COMMENT == 0
<code>SetCharPrm(INF_COMMENT, 40);</code>	<code>SetCharPrm(INF_COMMENT, 0);</code>
<pre> seg000:22E0 mov ah, 00h ; 000 - 2* - ADJUST MEMORY BLOCK SIZE (SETBLOCK) seg000:22E2 mov bx, 200h ; ES = segment address of block to change seg000:22E5 int 33h ; BK = new size in paragraphs           </pre>	<pre> seg000:22E0 mov ah, 00h ; 000 - 2* - ADJUST MEMORY BLOCK SIZE (SETBLOCK) seg000:22E2 mov bx, 200h ; ES = segment address of block to change seg000:22E5 int 33h ; BK = new size in paragraphs           </pre>

## INF\_XREFNUM

Это однобайтовое поле хранит максимальное возможное число перекрестных ссылок, которые IDA будет отображать в виде комментариев к инструкции. По умолчанию равно двум. При этом, если остальные ссылки не отображаются, но IDA сигнализирует об их наличие в виде двух точек, стоящих за последней отображаемой перекрестной ссылкой.

INF_XREFNUM == 2	INF_XREFNUM == 4
<code>SetCharPrm(INF_XREFNUM, 2);</code>	<code>SetCharPrm(INF_XREFNUM, 4);</code>

Может быть изменено как интерактивно («~Options\ Cross references\ Number of xrefs to display»), так и с помощью функции `SetCharPrm(INF_XREFNUM, xx)`



## INF\_ENTAB

Это однобайтовое поле управляет генерацией выходных файлов. Если оно равно единице, то IDA будет при форматировании использовать символы табуляции. В противном случае все отступы будут выполнены пробелами.

Табуляция позволяет значительно, иногда в два и более раз уменьшить размер файлов. Однако, некоторые редакторы и средства просмотра могут неправильно интерпретировать (или же вовсе игнорировать) символы табуляции. В этих случаях рекомендуется сбрасывать флаг `INF_ENTAB` (по умолчанию он установлен). Это можно сделать как интерактивно («~Options\ Text representation\ Use tabulations in output») так и с помощью следующего вызова:

```
SetCharPrm(INF_ENTAB,0);
```

INF_ENTAB == 1	INF_ENTAB == 0
<code>SetCharPrm(INF_ENTAB, 1);</code>	<code>SetCharPrm(INF_ENTAB, 0);</code>

## INF\_VOIDS

Это однобайтовое поле содержит флаг, указывающий IDA выводить после всех непосредственных операндов «похожих» на смещение (т.е. попадающих в интервал INF\_LOW\_OFF и INF\_HIGH\_OFF) комментарий «void», сигнализирующий пользователю, что тип автоматически не был определен и должен быть уточнен вручную.

По умолчанию этот флаг сброшен, потому что IDA и без комментариев привлекает внимание к операндам, выделяя их красным цветом. Однако, это невозможно осуществить в выходных файлах (ASM и LST), поэтому в этом случае рекомендуется устанавливать флаг INF\_VOIDS. Это можно сделать как интерактивно (~Options\ Text representation\ Display 'void' marks), так и с помощью вызова функции SetCharPrm

INF_VOIDS == 0	INF_VOIDS == 1
<code>SetCharPrm(INF_VOIDS, 0);</code>	<code>SetCharPrm(INF_VOIDS, 0);</code>
<code>mov si, 2C51h</code> <code>call sub_0_DD</code> <code>mov si, 2C4Dh</code> <code>call sub_0_2E2</code>	<code>mov si, 2C51h ; &lt;void&gt;</code> <code>call sub_0_DD</code> <code>mov si, 2C4Dh ; &lt;void&gt;</code> <code>call sub_0_2E2</code>

## INF\_SHOWAUTO

Это однобайтовое поле содержит флаг, управляющий индикатором автоанализа. По умолчанию он установлен. Если возникнет необходимость, то его можно отключить «~Options\ Analysis options\ Indicator enabled» или вызовом функции SetCharPrm

INF_SHOWAUTO == 1	INF_SHOWAUTO == 0
<code>SetCharPrm(INF_SHOWAUTO, 1);</code>	<code>SetCharPrm(INF_SHOWAUTO, 0);</code>
<code>↓ AU: idle READY</code>	<code>↓ READY</code>

Индикатор может принимать следующие значения:

Вид	Значение
AU: idle	Автоанализ завершен
AU:disable	Автоанализ выключен
FL:<адрес>	Трассировка порядка исполнения
PR:<адрес>	По указанному адресу была создана функция
AC:<адрес>	Указатель на текущее положение анализатора
LL:<номер>	Был загружен файл сигнатур
L1:<адрес>	Первый проход FLIRT
L2:<адрес>	Второй проход FLIRT
L3:<адрес>	Третий проход FLIRT
FI:<адрес>	Заключительный проход автоанализа
?:<адрес>	Байт по указанному адресу помечен как unexplored
@:<номер>	Индикатор различных действий

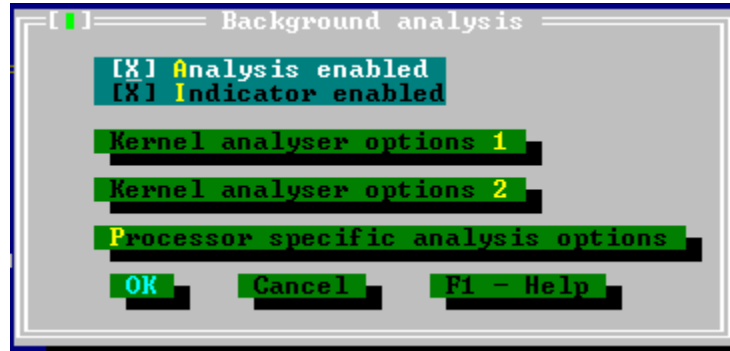
## INF\_AUTO



Это однобайтовое поле содержит флаг, управляющий автоанализом. То есть автоматическим анализом программы. Именно такой режим работы установлен по умолчанию.

Отключать его следуют только в тех случаях, когда результат работы автоматического анализатора не устает или вызывает «подвисание» дизассемблера. Такое часто случается с файлами, полученными с помощью ProcDump и подобных утилит.

Сделать это можно как интерактивно (~Options\ Background analysis\Analysis enabled), так и вызовом функции SetCharPrm(INF\_AUTO,0);



## INF\_BORDER

Это однобайтовое поле хранит флаг, управляющий вставкой линий, разделяющих код и данные в дизассемблере. Значительно улучшает читабельность листинга, поэтому по умолчанию IDA ведет себя именно так.

С другой стороны, дополнительные линии уменьшают число значащих строк, уместающихся на дисплее, а так же приводит к излишнему перерасходу бумаги при выводе дизассемблированного текста на принтер, поэтому в этих случаях эту опцию следует отключить вызовом функции SetCharPrm(INF\_BORDER,0) или интерактивно ~Options\ Text representation \ Display borders between data/code.

INF_BORDER == 1	INF_BORDER == 0
SetCharPrm(INF_BORDER, 1);	SetCharPrm(INF_BORDER, 0);
<pre>.text:00402507 ; .text:0040250E align 4 .text:00402510 .text:00402510 loc_0_402510: .text:00402510 .text:00402510 neg ecx .text:00402512 jmp ds:off_0_4025E0[ecx*4] .text:00402512 ;</pre>	<pre>.text:00402507 jmp ds:off_0_402630[edx*4] .text:0040250E align 4 .text:00402510 .text:00402510 loc_0_402510: .text:00402510 .text:00402510 neg ecx .text:00402512 jmp ds:off_0_4025E0[ecx*4] .text:00402519 align 4 .text:0040251C</pre>

## INF\_NULL

Это однобайтовое поле хранит флаг, управляющий генерацией пустых строк, вставляемых дизассемблером в различных местах для улучшения читабельности листинга. Однако в ряде случаев эту возможность следует отключать (например, при выводе текста на печать). Для этого следует воспользоваться вызовом SetCharPrm(INF\_NULL,0) или сбросить флажок ~Options\ Text representation \ Display empty lines

INF_NULL == 1	INF_NULL == 0
SetCharPrm(INF_NULL, 1);	SetCharPrm(INF_NULL, 0);

<pre>.text:00402507 ; .text:0040250E align 4 .text:00402510 .text:00402510 loc_0_402510: .text:00402510 .text:00402510 neg ecx</pre>	<pre>.text:00402507 ; .text:0040250E align 4 .text:00402510 loc_0_402510: .text:00402510 .text:00402510 neg ecx</pre>
--	---

## INF\_SHOWPREF

Это однобайтовое поле хранит флаг, который управляет выводом префиксов в дизассемблируемом листинге. Префикс – это адрес текущего байта.

Пример префикса: `.text:004024AC pop edi`

По умолчанию этот флаг установлен, и каждая линия предваряется префиксом.

Многоточечные структуры (например, массивы) в каждой строке содержат адрес своего первого элемента.

Например:

```
.text:004023C0 dword_0_4023C0 dd 68AD123h, 468A0788h, 0C102468Ah
.text:004023C0 dd 3C68302h, 8303C783h, 0CC7208F9h
.text:004023C0 dd 3498D00h
```

При этом не зависимо от значения флага INF\_SHOWPREF префиксы в ассемблерный листинг (\*.asm файл) не попадают.

Если по какой-то причине генерацию префиксов необходимо отключить, то это можно сделать с помощью вызова функции `SetCharPrm(INF_SHOWPREF,0)` или интерактивно ~ Options\ Text representation \ Line prefixes

INF_SHOWPREF == 1	INF_SHOWPREF == 0
<code>SetCharPrm(INF_SHOWPREF,1);</code>	<code>SetCharPrm(INF_SHOWPREF,0);</code>
<pre>.text:004024A0 pop edi .text:004024A1 leave .text:004024A2 retn</pre>	<pre>pop edi leave retn</pre>

## INF\_PREFSEG

Это однобайтовое поле содержит флаг, управляющий выводом имени сегмента в префиксе строки. По умолчанию флаг установлен и вместо полного адреса выводится имя сегмента.

Если же возникнет необходимость видеть полный адрес, то этот флаг можно сбросить. Сделать это можно либо интерактивно «~ Options \ Text representation \ Use segment names», либо вызовом функции `SetCharPrm(INF_PREFSEG,0)`

При этом листинг будет выглядеть, как показано ниже:

INF_PREFSEG == 1	INF_PREFSEG == 0
<code>SetCharPrm(INF_PREFSEG,1);</code>	<code>SetCharPrm(INF_PREFSEG,0);</code>
<code>.text:0040100F xor eax, eax</code>	<code>0000:0040100F xor eax, eax</code>

## INF\_ASMTYPE

Это однобайтовое поле хранит номер, начиная с нуля, задающий целевой ассемблер. Для PC всегда равно нулю, и указывает на «Generic for Intel 80x86»

Пример:

```
Message ("%x \n", GetCharPrm(INF_ASMTYPE));
```

## INF\_BASEADDR

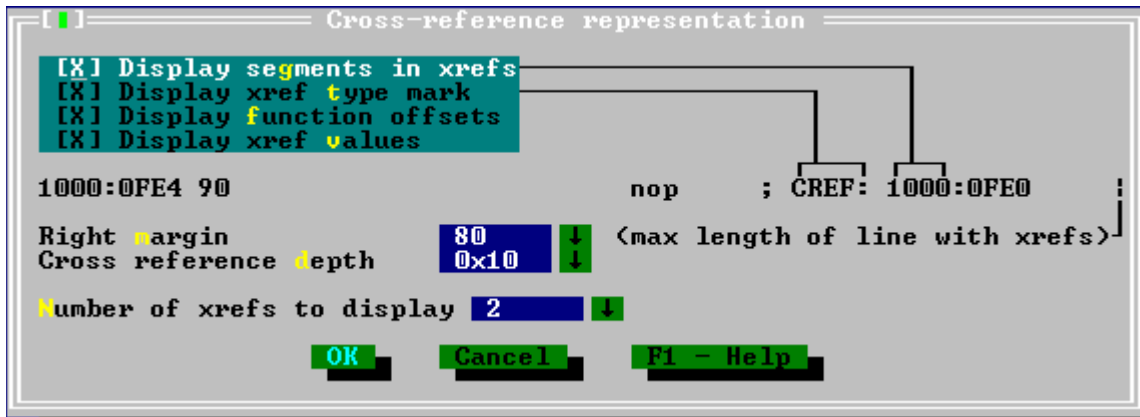
Это длинное поле хранит базовый параграф программы

Пример:

```
Message ("%x \n", GetLongPrm (INF_BASEADDR) );
1000
```

## INF\_XREFS

Это однобайтовое поле управляет представлением перекрестных ссылок в дизассемблируемом листинге. Может быть представлено комбинацией следующего набора битовых флагов:



### SW\_SEGXRF (0x01)

Установка этого флага приводит к тому, что IDA будет указывать полный адрес, включая сегмент, в перекрестных ссылках (по умолчанию).

Интерактивно этим значением можно управлять «~ Options \ Cross-reference representation \ Display segments in xrefs»

SW_SEGXRF == 1	SW_SEGXRF == 0
<code>SetLongPrm(INF_XREF, SW_SEGXRF);</code>	<code>SetLongPrm(INF_XREF, !SW_SEGXRF)</code>
<code>DATA XREF: .rdata:004070C0o</code>	<code>DATA XREF: 004070C0o</code>

### SW\_XRFMRK (0x02)

Установка этого флага приводит к тому, что IDA уточняет тип перекрестной ссылки, — представляет ли источник собой код или данные.

Интерактивно этим значением можно управлять «~ Options \ Cross-reference representation \ Display xref type mark»

SW_XRFMRK == 1	SW_XRFMRK == 0
<code>SetLongPrm(INF_XREF, SW_XRFMRK);</code>	<code>SetLongPrm(INF_XREF, !SW_XRFMRK)</code>
<code>DATA XREF: .rdata:004070C0o</code>	<code>XREF: 004070C0o</code>

## SW\_XRFFNC (0x04)

Установка этого флага приводит к тому, что IDA выражает адрес ссылки через смещение, относительно начала ближайшей функции.

Интерактивно этим значением можно управлять «~ Options \ Cross-reference representation \ Display function offsets»

SW_XRFFNC == 1	SW_XRFFNC == 0
<code>SetLongPrm(INF_XREF, SW_XRFFNC);</code>	<code>SetLongPrm(INF_XREF, !SW_XRFFNC)</code>
CODE XREF: <code>start+AFp</code>	CODE XREF: <code>004010CFp</code>

## SW\_XRFVAL (0x08)

Установка этого флага приводит к тому, что IDA отображает значение перекрестной ссылки в дизассемблируемом листинге. В противном же случае его заменяют три точки.

SW_XRFVAL == 1	SW_XRFVAL == 0
<code>SetLongPrm(INF_XREF, SW_XRFVAL);</code>	<code>SetLongPrm(INF_XREF, !SW_XRFVAL)</code>
CODE XREF: <code>004010CFp</code>	CODE XREF: <code>...</code>

## INF\_BINPREF

Это короткое поле задает число байт, отображающих шестнадцатеричный оп-код инструкции. По умолчанию равно нулю, то есть IDA дамп не отображает. Однако, в ряде случаев потребность в нем все же возникает, кроме того, для кого-то этот может быть вопрос удобства или привычки.

Тогда можно воспользоваться вызовом `SetShortPrm(INF_BINPREF, 0x10)` или изменить то же самое значение интерактивно «~ Options \ Text representation \ Number of opcode bytes»

INF_BINPREF == 0	INF_BINPREF == 0x10
<code>SetShortPrm(INF_BINPREF, 0);</code>	<code>SetShortPrm(INF_BINPREF, 0x10);</code>
<pre>.text:00401000 sub_0_401000 proc near .text:00401000 push offset aHeloSailor .text:00401005 mov ecx, offset dword_0_408900 .text:0040100A call ??6ostream@@QAEAAV0@PBD@Z .text:0040100F xor eax, eax .text:00401011 retn .text:00401011 sub 0_401000 endp</pre>	<pre>.text:00401000 sub_0_401000 proc near .text:00401000 68 40 80 40 00 push offset aHeloSailor .text:00401005 B9 00 89 40 00 mov ecx, offset dword_408900 .text:0040100A E8 72 2B 00 00 call ostream@@QAEAAV0@PBD@Z .text:0040100F 33 C0 xor eax, eax .text:00401011 C3 ketn .text:00401011 sub_0_401000 endp</pre>

## INF\_CMTFLAG

Это короткое поле содержит набор флагов, манипулирующим выводом и представлением комментариев в дизассемблируемом листинге.

## SW\_RPTCMT

Этот флаг управляет выводом повторяемых комментариев. По умолчанию установлен. Если возникнет необходимость отключить генерацию повторяемых комментариев, то это

можно сделать как вызовом функцией SetShortPrm, так и интерактивно «~Options\ Text representation \ Display repeatable comments»

SW_RPTCMT == 1	SW_RPTCMT == 0
<code>SetShortPrm(INF_CMTFLAG, SW_RPTCMT);</code>	<code>SetShortPrm(INF_CMTFLAG, !SW_RPTCMT)</code>
<code>Jb short near ptr dword_4023AC ; repeatable comment</code>	<code>Jb short near ptr dword_0_4023AC</code>

### SW\_ALLCMT

Этот флаг будучи установленным приводит к тому, что IDA комментирует каждую строку в дизассемблируемом тексте (обычно приводит расшифровку мнемоник команд).

По умолчанию этот флаг сброшен, поскольку захламляет листинг малоинтересной информацией. Однако, это может оказаться полезным для начинающих пользователей, или любых других не знающих на память все инструкции микропроцессора (особенно новых моделей). В этом случаи комментарии IDA позволят сэкономить некоторое количество времени.

SW_ALLCMT == 1	SW_ALLCMT == 0
<code>SetShortPrm(INF_CMTFLAG, SW_ALLCMT);</code>	<code>SetShortPrm(INF_CMTFLAG, !SW_ALLCMT)</code>
<code>Call sub_0_2E2 ; Call Procedure</code> <code>jnb loc_0_2321 ; Jump if Not Below (CF=0)</code> <code>nop ; No Operation</code>	<code>call sub_0_2E2</code> <code>jnb loc_0_2321</code> <code>nop</code>

### SW\_NOCMT

Установка этого флага приводит к тому, что IDA вообще не будет отображать комментариев, не зависимо от состояния остальных настроек. По умолчанию флаг сброшен.

### SW\_LINNUM

Этот флаг будучи установленным приводит к тому, что IDA при наличии необходимой отладочной информации в файле будет отображать номера строк исходного текста программы.

### SW\_MICRO

### INF\_NAMETYPE

Это короткое поле содержит флаг, управляющий представлением автогенерируемых имен (в терминологии IDA - dummy names). Эти имена автоматически присваиваются всем созданным меткам и процедурам.

Флаг	значение	пояснения
NM_REL_OFF	0	Относительная база сегмента и полное смещение loc_0_1234
NM_PTR_OFF	1	Базовый адрес сегмента и смещение loc_1000_1234
NM_NAM_OFF	2	Имя сегмента и смещение (по умолчанию)

		loc_dseg_1234
NM_REL_EA	3	Сегмент, относительный базовому адресу и полный адрес loc_0_11234
NM_PTR_EA	4	Базовый адрес сегмента и полный адрес loc_1000_11234
NM_NAM_EA	5	Имя сегмента и полный адрес loc_dseg_11234
NM_EA	6	Полный адрес (без нуля слева) loc_12
NM_EA4	7	Полный адрес (не менее четырех знаков) loc_0012
NM_EA8	8	Полный адрес (не менее восьми знаков) loc_0000012
NM_SHORT	9	Имя сегмента и смещение без спецификатора типа dseg_1234
NM_SERIAL	10	Перечисленные имена (1,2,3...) loc_1

### INF\_SHOWBADS

Это однобайтовое поле, будучи установленным, приводит к тому, что IDA будет оставлять в виде дампа все инструкции, которые могут быть неправильно ассемблированы. Например, в исследуемой программе могут встретиться недокументированные команды процессора (подробнее с ними можно ознакомиться, например, на сайте [www.x86.org](http://www.x86.org))

Разумеется, что распространенные ассемблеры выдадут ошибку и прекратят работу. Однако, это не худшая ситуация. Множество команд 80x86 процессоров могут быть ассемблированы по-разному. Например, ADD bx, 0x10 может быть представлена как опкодом 81 C3 01 00, так и 83 C3 10

Разница здесь в том, что последняя команда добавляет к ВХ байт, автоматически расширяя его до слова с учетом знака. Следовательно, возникает неоднозначность, – часто приводящая к неработоспособности программы. Даже если не использовался самомодифицирующийся код, изменение длины инструкции «потянет» за собой все метки и абсолютные адреса в программе. Впрочем, при условии правильного преобразования типов непосредственных операндов это не нарушит работоспособность программы. Поэтому по умолчанию это опция отключена.

Если же в ней возникнет необходимость, то нужно воспользоваться функцией SetCharPrm(INF\_SHOWBADS,1) или интерактивно ~Options \ Text representation \ Display bad instructions <BAD> marks.

INF_SHOWBADS == 1	INF_SHOWBADS == 0
SetCharPrm(INF_SHOWBADS,1)	SetCharPrm(INF_SHOWBADS,0)
seg000:0220 db 0E9h,0,0 ; <BAD> jmp \$+3	seg000:0220 jmp \$+3

**ЗАМЕЧАНИЕ:** все же часть инструкций IDA не помечает, как <BAD>, хотя их отказываются ассемблировать распространенные ассемблеры.

Например, команду JMP FAR segment:offset IDA, в отличие от большинства ассемблеров, не считает не правильной. В результате чего приходится вручную менять ее в листинге на последовательность:

```
DB 0Eah
    DW offset
```

### INF\_PREFFLAG

Это однобайтовое поле хранит флаг, задающий формат вывода префикса на экран. Имеет смысл только когда вывод префиксов разрешен, (то есть флаг INF\_SHOWPREF установлен).

Флаг представляет собой возможные комбинации из трех битов. При этом бит PREF\_FNCOFF имеет приоритет над PREF\_SEGADR. То есть, комбинация (PREF\_FNCOFF | PREF\_SEGADR) равносильна PREF\_FNCOFF. Однако, это не документировано и возможно в последующих версиях IDA будет вести себя иначе.

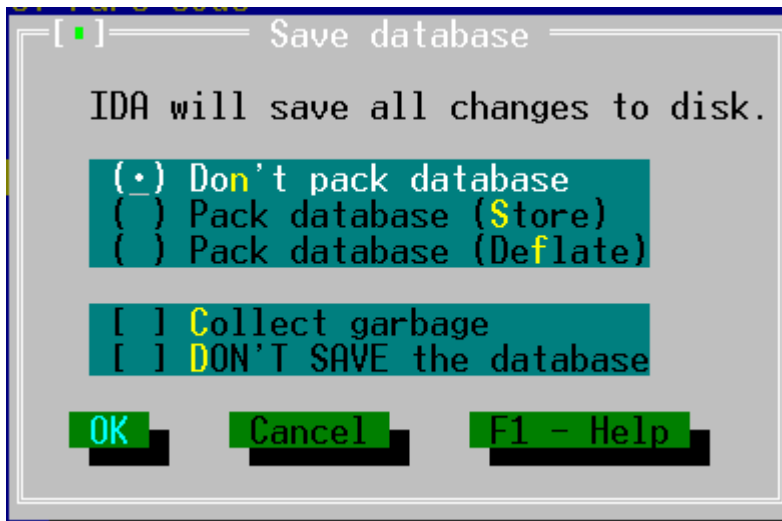
Интерактивно это можно изменить с помощью следующих опций диалога настройки:

- Options \ Text representation \ Segment addresses
- Options \ Text representation \ Function offsets
- Options \ Text representation \ Display stack pointer

Флаг	значение	Пояснение	Пример
PREF_SEGADR	0x01	Представлять префикс в виде сегментного адреса	seg000:2190
PREF_FNCOFF	0x02	Представлять префикс в виде смещения внутри функции	Sub_0_22DD+1B
PREF_STACK	0x04	Завершать префикс указателем стека	Seg000:2190 008 Sub_0_22DD+1B 008

### INF\_PACKBASE

Это однобайтовое поле хранит тип упаковки базы IDA, предлагаемый по умолчанию при выходе из дизассемблера.



INF_PACKBASE == 0	INF_PACKBASE == 1	INF_PACKBASE == 2
SetCharPrm(INF_PACKBASE, 0);	SetCharPrm(INF_PACKBASE, 1);	SetCharPrm(INF_PACKBASE, 2);
<input checked="" type="checkbox"/> Don't pack database <input type="checkbox"/> Pack database (Store) <input type="checkbox"/> Pack database (Deflate)	<input type="checkbox"/> Don't pack database <input checked="" type="checkbox"/> Pack database (Store) <input type="checkbox"/> Pack database (Deflate)	<input type="checkbox"/> Don't pack database <input type="checkbox"/> Pack database (Store) <input checked="" type="checkbox"/> Pack database (Deflate)

**ЗАМЕЧАНИЕ:** оба типа упаковки обладают слабым сжатием, поэтому при возникновении потребности в дисковом пространстве или передачи базы по

коммуникационным каналам рекомендуется ее упаковать с помощью любого подходящего архиватора (например, zip, arj) при этом выигрыш может быть более, чем десятикратный.

## INF\_ASCIIIFLAGS

Это однобайтовое беззнаковое поле задает стиль автогенерируемых строковых имен (или вообще запрещает создание таковых)

Представляет собой комбинацию следующих битов.

### ASCF\_GEN

Если этот флаг установлен, то IDA автоматически генерирует имена для всех ASCII строк, состоящее из читабельных символов этой строки.

Так, например, встретив строку «Hello, Word» IDA создаст имя «aHello\_Word». В противном случае что-то наподобие «asc\_0\_206». Разумеется, что генерация осмысленных имен улучшает читабельность листинга и ускоряет анализ.

По умолчанию IDA ведет себя именно так. Если по какой-то причине возникнет желание отключить эту возможность, то можно воспользоваться функцией SetCharPrm(INF\_ASCIIIFLAG,0) или интерактивно через «~ Options \ ASCII string options \ Generate names»

SetCharPrm(INF_ASCIIIFLAG,1);	SetCharPrm(INF_ASCIIIFLAG,0);
seg000:2192 a123456789abcde db '123456789ABCDEFGF',0	seg000:2192 db '123456789ABCDEFGF',0

### ASCF\_AUTO

Этот флаг, будучи установленным, приводит к тому, что IDA будет помечать все создаваемые имена, как 'autogenerated'.

Это приведет к отображению их другим цветом и автоматическому удалению при преобразовании имени к 'unexplored'.

По умолчанию флаг установлен. Если возникнет необходимость его изменить, то это можно сделать с помощью функции SetCharPrm(INF\_ASCIIIFLAGS,IASCF\_AUTO) или интерактивно «~Options \ ASCII string options \ Mark as autogenerated»

ASCF_AUTO == 1			ASCF_AUTO == 0		
seg000:2192	db	31h ; 1	seg000:2192	db	31h ; 1
seg000:2193	db	32h ; 2	seg000:2193	db	32h ; 2
seg000:2194	db	33h ; 3	seg000:2194	db	33h ; 3
seg000:2195	db	34h ; 4	seg000:2195	db	34h ; 4
seg000:2196	db	35h ; 5	seg000:2196	db	35h ; 5
seg000:2197	db	36h ; 6	seg000:2197	db	36h ; 6
seg000:2198	db	37h ; 7	seg000:2198	db	37h ; 7
seg000:2199	db	38h ; 8	seg000:2199	db	38h ; 8
seg000:219A	db	39h ; 9	seg000:219A	db	39h ; 9
seg000:219B	db	41h ; A	seg000:219B	db	41h ; A
seg000:219C	db	42h ; B	seg000:219C	db	42h ; B
seg000:219D	db	43h ; C	seg000:219D	db	43h ; C
seg000:219E	db	44h ; D	seg000:219E	db	44h ; D
seg000:219F	db	45h ; E	seg000:219F	db	45h ; E
seg000:21A0	db	46h ; F	seg000:21A0	db	46h ; F
seg000:21A1	db	47h ; G	seg000:21A1	db	47h ; G
seg000:21A2	db	0 ;	seg000:21A2	db	0 ;
Создается имя					
seg000:2192	a123456789abcde	db '123456789ABCDEFGF',0	seg000:2192	a123456789abcde	db '123456789ABCDEFGF',0



Преобразуем регион в unexplored			
seg000:2192	db	31h ; 1	seg000:2192 a123456789abcde db 31h ; 1
seg000:2193	db	32h ; 2	seg000:2193 db 32h ; 2
seg000:2194	db	33h ; 3	seg000:2194 db 33h ; 3
seg000:2195	db	34h ; 4	seg000:2195 db 34h ; 4
seg000:2196	db	35h ; 5	seg000:2196 db 35h ; 5
seg000:2197	db	36h ; 6	seg000:2197 db 36h ; 6
seg000:2198	db	37h ; 7	seg000:2198 db 37h ; 7
seg000:2199	db	38h ; 8	seg000:2199 db 38h ; 8
seg000:219A	db	39h ; 9	seg000:219A db 39h ; 9
seg000:219B	db	41h ; A	seg000:219B db 41h ; A
seg000:219C	db	42h ; B	seg000:219C db 42h ; B
seg000:219D	db	43h ; C	seg000:219D db 43h ; C
seg000:219E	db	44h ; D	seg000:219E db 44h ; D
seg000:219F	db	45h ; E	seg000:219F db 45h ; E
seg000:21A0	db	46h ; F	seg000:21A0 db 46h ; F
seg000:21A1	db	47h ; G	seg000:21A1 db 47h ; G
seg000:21A2	db	0 ;	seg000:21A2 db 0 ;

## ASCF\_SERIAL

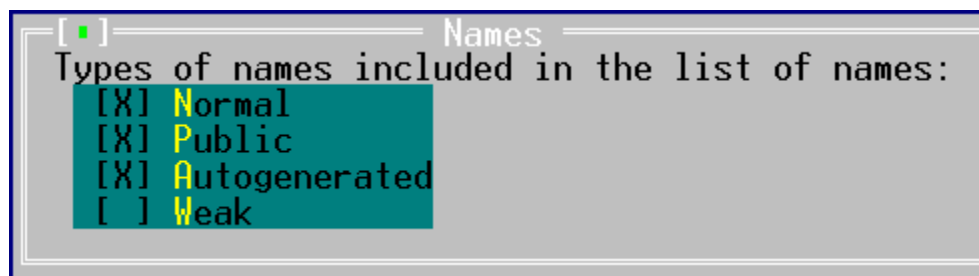
Если этот флаг будет установлен, то IDA будет генерировать следующие (в терминологии IDA *последовательные*) имена 'pref0','pref1','pref2'... где 'pref' – префикс имени, который для строк по умолчанию равен 'a'

По умолчанию этот флаг сброшен, но если возникнет необходимость, то его можно установить с помощью функции SetCharPrm( или же интерактивно «~Options \ ASCII string options\ Generate serial names»

ASCF_SERIAL == 1	ASCF_SERIAL == 0
SetCharPrm(INF_ASCII_FLAGS, ASCF_SERIAL);	SetCharPrm(INF_ASCII_FLAGS, !ASCF_SERIAL);
seg000:2192 a0 db '123456789ABCDEFG',0	seg000:2192 a123456789abcde db '123456789ABCDEFG',0

## INF\_LISTNAMES

Это однобайтовое беззнаковое поле содержит атрибуты имен, автоматически включаемых в Список Имен (Name List).



LN_NORMAL	0x01	Имя без атрибутов (по умолчанию)
LN_PUBLIC	0x02	Имя с атрибутом public
LN_AUTO	0x04	Автогенерируемое имя
LN_WEAK	0x08	Имя с атрибутом weak

## INF\_START\_SS

Это длинное поле хранит значение регистра SS при запуске программы. Для того, что бы его получить IDA прибегает к эмуляции загрузки и действует в соответствии с исследуемым типом файла и декларированным правилам загрузки его операционной системой.

Однако, это не гарантирует, что полученное значение будет тождественно действительному. Впрочем, такая точность на практике и не требуются. В крайнем случае можно принудительно указать требуемый базовый адрес загрузки или изменить непосредственно само значение INF\_START\_SS.

Пример:

```
Message("0x%x \n",GetLongPrm(INF_START_SS));  
0x1000  
SetLongPrm(INF_START_SS,0);  
Message("0x%x \n",GetLongPrm(INF_START_SS));  
0
```

## INF\_START\_CS

Это длинное поле хранит значение регистра CS при запуске программы. Для того, что бы его получить IDA прибегает к эмуляции загрузки и действует в соответствии с исследуемым типом файла и декларированным правилам загрузки его операционной системой.

Пример:

```
Message("0x%x \n",GetLongPrm(INF_START_CS));  
0x1000
```

## INF\_MAIN

В файле определений IDC.IDC сообщается, что это длинное поле содержит адрес процедуры main(), однако, при попытке его чтения всегда возвращается ошибка BADADDR.

Например:

```
Message("0x%X \n",GetLongPem(INF_MAIN));  
0xFFFFFFFF
```

## INF\_SHORT\_DN

Это длинное поле хранит короткую форму вывода «замангленных» имен. Назначение отдельных битов не описано в файле 'idc.idc', в котором содержится ссылка на 'demangle.hpp', входящий в состав IDA SDK.

ФЛАГ	БИТ	ЗНАЧЕНИЕ
MNG_DEFNEAR	0x00000000	Подавлять в именах ключевое слово near
MNG_DEFFAR	0x00000002	Подавлять в именах ключевое слово far
MNG_DEFHUGE	0x00000004	Подавлять в именах ключевое слово huge
MNG_DEFNONE	0x00000006	Выводить все
MNG_NODEFINIT	0x00000008	Подавлять вывод всего, кроме основного имени
MNG_NOUNDERSCORE	0x00000010	Подавлять вывод подчеркивов для __scall, __pascal...
MNG_NOTYPE	0x00000020	Подавлять вывод типа параметров
MNG_NORETTYPE	0x00000040	Подавлять вывод типа возвращаемого функцией значения
MNG_NOBASEDT	0x00000080	Подавлять вывод базовых типов

MNG_NOCALLC	0x00000100	Подавлять вывод слов <code>__pascal\__ccall</code> и подобных
MNG_NOPOSTFC	0x00000200	Подавлять вывод постфиксного <code>const</code>
MNG_NOSCTYP	0x00000400	Подавлять вывод ключевых слов <code>public\private\protected</code>
MNG_NOTHROW	0x00000800	Подавлять вывод описания <code>throw</code>
MNG_NOSTVIR	0x00001000	Подавлять вывод ключевых слов <code>static</code> и <code>virtual</code>
MNG_NOECSU	0x00002000	Подавлять вывод ключевых слов <code>class\struct\union\enum</code>
MNG_NOCSVOL	0x00004000	Всюду подавлять вывод ключевых слов <code>const</code> и <code>volatile</code>
MNG_NOCLOSUR	0x00008000	Подавлять вывод <code>__closure</code> (для компиляторов Borlncand)
MNG_SHORT_S	0x00010000	Выводить <code>signed (int)</code> в формате <code>s(int)</code>
MNG_SHORT_U	0x00020000	Выводить <code>unsigned (int)</code> в формате <code>u(int)</code>
MNG_ZPT_SPACE	0x00040000	Выводить пробел после запятой в аргументах функций
MNG_IGN_ANYWAY	0x00080000	Игнорировать постфикс <code>'_nn'</code> в конце строки
MNG_IGN_JMP	0x00100000	Игнорировать префикс <code>'j_'</code> в начале строки
MNG_MOVE_JMP	0x00200000	Сохранять префикс <code>'j_'</code> в размангленных именах

Подробнее об этом можно найти в описании функции `Demangle`.

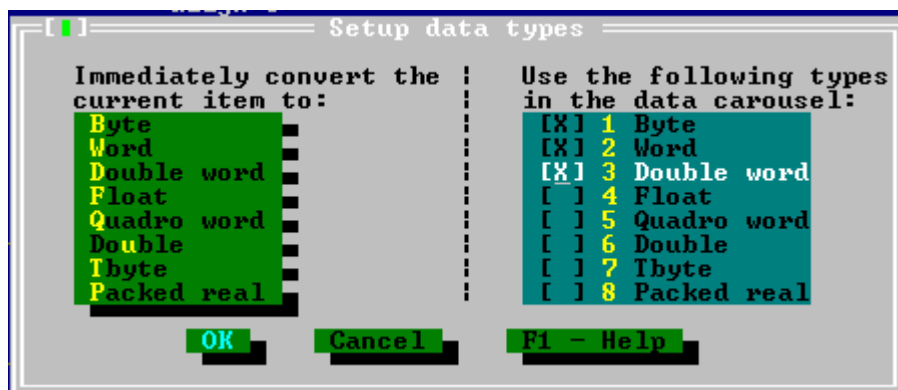
### INF\_LONG\_DN

Это длинное поле хранит полный формат вывода «замангленных» имен. Представляет собой комбинацию флагов, назначение которых описано выше.

### INF\_DATATYPES

Это длинное поле хранит разрешенные к использованию типы данных, то есть такие, что будут поочередно перебираться, скажем, при нажатии 'D'.

Файл `IDC.IDC` не содержит расшифровки отдельных битов этого поля, но в интерактивно диалоге настройки (~Options \ Setup data types ) типы данных перечислены в порядке следования флагов в этом поле!



БИТ	ЗНАЧЕНИЕ
0x1	Байт
0x2	Слово
0x4	Двойное слово
0x8	Float
0x10	Четвертное слово
0x20	Double

0x40	Tbyte
0x80	Упакованное real

Пример использования:

```
Message ("%b \n", GetLongPrm(INF_DATATYPES));
111
```

## INF\_STRTYPE

Это длинное поле хранит текущий стиль ASCII – строк. Первый байт представляет собой один из следующих флагов.

ФЛАГ	ЗНАЧЕНИЕ
0	Строка не претворена полем длины
ASCSTR_PASCAL	Стиль Pascal – строка предваряется байтом длины .data:00408040 aHeloSailor db 0Ch, 'Helo,Sailor!'
ASCSTR_LEN2	Стиль WinPascal – строка предваряется словом длины .data:00408040 aHeloSailor dw 0Ch, db 'Helo,Sailor!'
ASCSTR_UNICODE	Стиль UNICODE `Н`,0,`е`,0,`л`,0,`о`,0,`,`0,`S`,0,`а`,0,`и`,0,`л`,0,`о`,0,`р`,0,`!`
ASCSTR_LEN4	Стиль Delphi – 4 байта на длину .data:00408040 aHeloSailor dw 0Ch, dw 0, db 'Helo,Sailor!'

Если первый байт не равен нулю, то, значит, впереди строки находится поле, определяющее его длину. Иначе используется символ конца строки. Его определяют второй и третий байт поля INF\_STRTYPE.

Если второй символ будет равен '\0', то он будет проигнорирован. Поэтому если необходимо задать два типа завершающих символов (например, ноль и '\$'), то '\0' следует указывать первым из них.

Пример:

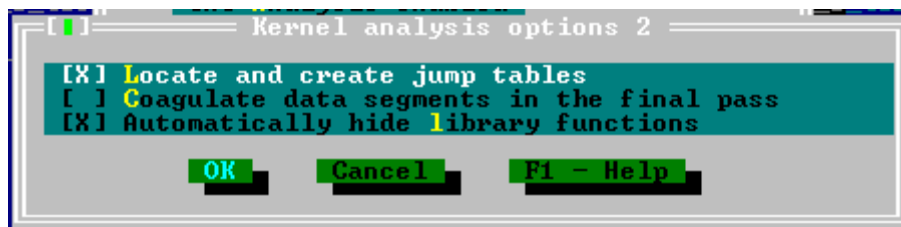
```
Message ("%x \n", GetLongPrm(INF_STRTTYPE));
0
```

```
SetLongPrm(INF_STRTTYPE, '$'>>0x10);
```

## INF\_AF2

Это длинное беззнаковое поле хранит дополнительные флаги анализатора.

ФЛАГ	БИТ	ЗНАЧЕНИЕ
AF2_JUMPTBL	0x1	Выявлять и создавать таблицы переходов
AF2_DODATA	0x2	Сворачивать сегмент данных в последнем проходе
AF2_HFLIRT	0x4	Автоматически скрывать стандартные библиотечные функции (этот флаг не описан в файле IDC.IDC)



### success SetPrCSR (char processor);

Функция позволяет изменить тип процессора, отличный от выбранного при загрузке дизассемблируемого файла. Однако возможности динамической смены процессора несколько ограничены. Допустим выбор лишь в границах текущей линейки (серии) микропроцессоров.

Это объясняется тем, что на стадии загрузки происходят неустраняемые средствами последующих уровней изменения и настройки на конкретный процессор. (Большой частью это относится к невозможности IDA перезагружать микропроцессорные модули, отвечающие за дизассемблирование. Такой модуль может быть загружен один только раз и на весь сеанс работы окажутся доступными лишь поддерживаемые им типы микропроцессоров)

Для линейки Intel в силу их полной обратной совместимости, динамический выбор модели процессора не критичен, поскольку можно выбрать самого позднего из доступных представителей, и можно быть уверенным, что все команды будут дизассемблированы. Аналогичный результат можно получить, выбрав тип «meta pc», включающий в себя команды всех моделей микропроцессоров.

Рассмотрим работу этой функции на следующей примере. Загрузим для дизассемблирования бинарный или com-файл (для которого IDA по умолчанию выбирает 8086 микропроцессор), но содержащий инструкции более поздних моделей.

Разумеется, они окажутся не дизассемблированными и результат работы IDA может выглядеть, например, так:

```

seg000:02E9      mov     ax, ds:413h
seg000:02EC      db     0C1h ; -
seg000:02ED      db     0E0h ; p
seg000:02EE      db     6 ;
seg000:02EF      cmp     ax, 0A000h

```

### SetPrCSR ("metapc");

```

seg000:02E9      mov     ax, ds:413h
seg000:02EC      shl     ax, 6
seg000:02EF      cmp     ax, 0A000h

```

Смена типа процессора, привела к тому, что IDA заново проанализировала изучаемый файл и автоматически дизассемблировала интересующие нас инструкции. Разумеется, что обратная смена на 8086 модель приведет к тому, что листинг будет приведен к первоначальному виду

При этом IDA может так же изменять целевой ассемблер, поэтому рекомендуется на всякий случай удостовериться в приемлемости ее выбора, вызвав диалог «~Options \ Target Assembler». Впрочем, для линии IBM PC он имеется в единственном числе – «Generic for Intel 80x86» и беспокоиться нет никакой необходимости.

Generic for Intel 80x86

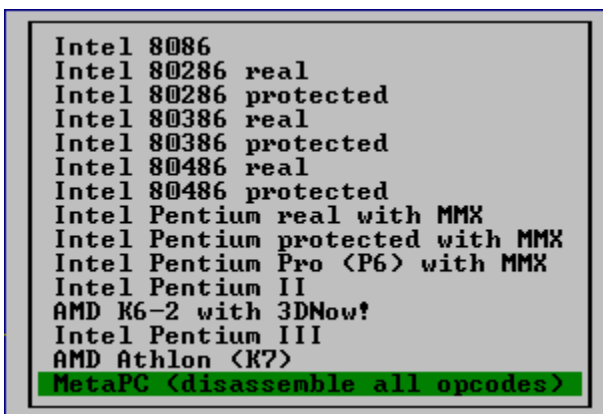
Операнд processor может принимать следующие значения, перечисленные ниже в таблице. К регистру функция не чувствительна, поэтому 'metarc' и 'MetaPC' задают один и то же тип процессора.

Отбивка слева указывает, что объединяемые ее микропроцессоры могут динамически выбираться во время дизассемблирования файла.

Операнд	Процессор	Серия
8086	Intel 8086	Линейка IBM PC
80286r	Intel 80286 real mode	
80286p	Intel 80286 protected mode	
80386r	Intel 80386 real mode	
80386p	Intel 80386 protected mode	
80486r	Intel 80486 real mode	
80486p	Intel 80486 protected mode	
80586r	Intel Pentium & MMX real mode	
80586p	Intel Pentium & MMX prot mode	
80686p	Intel Pentium Pro & MMX	
k62	AMD K6-2 with 3DNow!	
p2	Intel Pentium II	
p3	Intel Pentium III	
athlon	AMD K7	
metarc	Дизассемблировать все инструкции IBM PC	
8085	Intel 8085	
z80	Zilog 80	Линейка Zilog 80
z8	Zilog 8	Линейка Zilog 8
860xr	Intel 860 XR	Линейка Intel 860
860xp	Intel 860 XP	
8051	Intel 8051	Линейка Intel 51
80196	Intel 80196	Линейка Intel 80196
m6502	6502	Линейка 65xx line
m65c02	65c02	
64180	Hitachi HD64180	
pdp11	DEC PDP/11	Линейка PDP line
68000	Motorola MC68000	Линейка Motorola 680x0
68010	Motorola MC68010	
68020	Motorola MC68020	
68030	Motorola MC68030	
68040	Motorola MC68040	
68330	Motorola CPU32 (68330)	
68882	Motorola MC68020 with MC68882	
68851	Motorola MC68020 with MC68851	
68020EX	Motorola MC68020 with both	
6800	Motorola MC6800	Линейка Motorola 8bit
6801	Motorola MC6801	
6803	Motorola MC6803	
6301	Hitachi HD 6301	
6303	Hitachi HD 6303	
6805	Motorola MC6805	
6808	Motorola MC6808	
6809	Motorola MC6809	

6811	Motorola MC6811	
java	java	Серия Java
ppc	PowerPC	Линейка PowerPC
arm710a	ARM 7xx серия	Линейка ARM
arm	То же самое, что и arm710a	
armb	ARM big endian	
tms320c2	TMS320C2x серия	Серия TMS 16bit адресации
tms320c5	TMS320C5x серия	Линейка TMS VLIW I
tms320c6	TMS320C6x серия	
sh3	Hitachi SH3 (little endian)	Hitachi SH line
sh3b	Hitachi SH3 (big endian)	
sh4	Hitachi SH4 (little endian)	
sh4b	Hitachi SH4 (big endian)	
avr	ATMEL AVR	Серия ATMEL
mipsl	MIPS little endian	Линейка MIPS: R2000, 3000, R4000, R4200, R4300, 4400, R4600, R8000, R10000
mipsb	MIPS big endian	
mipsr	MIPS & RSP	
h8300	H8/300x in normal mode	Hitachi H8 line
h8300a	H8/300x in advanced mode	
h8s300	H8S in normal mode	
h8s300a	H8S in advanced mode	
pic16cxx	Michrochip PIC	Серия микроконтроллеров

Все вышесказанное остается верным и для интерактивного выбора типа процессора посредством команды меню «~Options \ Processor type»



При попытке смены типа процессора IDA может выдать ошибку, например: «The processor type "metapc" isn't included in the standard version of IDA Pro. Please check our web site for information about ordering additional processor modules»

Это обозначает, что необходимый для дизассемблирования модуль отсутствует или не найден. Его можно получить, обратившись к вашему поставщику IDA или на сайте разработчика IDA ([www.idapro.com](http://www.idapro.com))

Поскольку для DOS, OS\2 и Windows версий дизассемблера используются разные модули, то вполне возможно, что один из них отсутствует или поврежден, когда остальные вполне работоспособны.

Расшифровка расширений приводится ниже в таблице.

расш	Платформа
d32	Процессорный модуль для OS\2 версии дизассемблера
dll	Процессорный модуль для MS-DOS версии дизассемблера

Четвертая версия IDA в полной поставке, включает в себя следующие файлы:

Файл	Семейство процессоров
ARM	Семейство ARM (серия ARM 7xx)
AVR	Линейка чипов ATMEL AVR
H8	Линейка чипов Hitachi H8 (H8/300x и H8S серии)
I196	Микропроцессор Intel 80196
I51	Микропроцессор Intel 8051
I860	Микропроцессор Intel 860 XR
JAVA	Java Virtual Machine
M65	Микропроцессоры серии 65xx
MC8	Семейство 8-разрядных микропроцессоров фирмы Motorola (MC6800, MC6801, MC6803, MC6805, MC6808, MC6809, MC6811)
	Семейство 8-разрядных микропроцессоров фирмы Hitachi (HD 6301, HD 6303)
MC68	Микропроцессоры серии Motorola 680x0
PC	Микропроцессоры линейки IBM PC
PDP11	DEC PDP-11
PIC	Микроконтроллеры Microchip серий PIC16C5x PIC16Cxx PIC17Cxx
Z8	Микропроцессоры линейки Zilog 8
Z80	Микропроцессоры линейки Zilog 80

Сравнивая эту таблицу с приведенным выше перечнем поддерживаемых IDA процессоров, можно заметить, что часть из них в поставку не входит.

С другой стороны, если вам не нужно дизассемблировать ничего, кроме программ для IBM PC, то все остальные модули можно удалить, освободив немного дискового пространства.

Если в заголовке загружаемого файла отсутствует информация о типе процессора, то IDA выбирает его, руководствуясь расширением.

Соответствия расширений и типов микропроцессора перечислены в файле IDA.CFG в секции DEFAULT\_PROCESSOR:

расширение	Тип процессора
"com"	"8086"
"exe"	"metapc"
"dll"	"metapc"
"drv"	"metapc"
"sys"	"metapc"
"bin"	"metapc"
"ovl"	"metapc"
"ovr"	"metapc"
"ov?"	"metapc"
"nlm"	"metapc"
"lan"	"metapc"
"dsk"	"metapc"
"obj"	"metapc"
"prc"	"68000" (PalmPilot программы)
"axf"	"arm710a"
"h68"	"68000" (MC68000 для *.H68 файлов)
"i51"	"8051" (i8051 для *.I51 файлов)
"sav"	"pdp11" (PDP-11 для *.SAV файлов)



"rom"	"z80" (для *.ROM файлов)
"class"	"java"
"cls"	"java"
"s19"	"6811"
"**"	"metapc"

### long Batch (long batch);

Функция позволяет устанавливать (или снимать) *пакетный* режим работы. При этом IDA не выводит никаких диалоговых окон и не выдает предупреждений. Это может быть полезным при автономной работе и во время выполнения скриптов.

В версии IDA 4.0 присутствуют некоторые ошибки в реализации пакетного режима. Так, например, попытка вызова калькулятора вызовет зависание IDA, поэтому пользоваться им следует с осторожностью и всегда обращать внимание, что бы скрипты, использующие его, возвращались в обычный режим при возврате в IDA.

==batch	Режим
0	Обычный режим
1	Пакетный режим

Функция возвращает прежний режим работы. Следующий скрипт определяет его текущее значение без изменений режима работы.

```
auto a,s;
s="нормальный";
a=Batch(0);
Batch(a);
if (a) s="пакетный";
Message("Режим работы %s \n",s);
Режим работы нормальный
```

### char GetIdaDirectory ();

Функция возвращает полный путь к директории, в которой расположена IDA, без завершающего слеша в конце.

Например:

```
Message ("%s \n", GetIdaDirectory ());
```

```
D:\DEBUG\IDA384
```

Return	Пояснения
	полный путь к директории, в которой расположена IDA

Точнее это путь к исполняемому файлу IDA.EXE (idaw.exe\ idax.exe). Расположение остальных файлов зависит от версии.

Так, например, IDA 3.6 хранила все скрипты в базовом каталоге, а последние версии в каталоге IDC.

Эти различия необходимо учитывать при поиске требуемых файлов. Желательно предусмотреть возможность диалога с пользователем и «ручным» указанием путей, а не полагаться на то, что требуемый файл окажется на месте.

## char GetInputFile ();

Возвращает имя дизассемблируемого файла вместе с расширением. Версия под win32 поддерживает длинные имена файлов, включая пробелы. Дозагрузка фалов не влияет на результат работы этой функции.

Пример использования:

```
Message ("%s \n ",  
GetInputFile ()  
);
```

My File.exe

Return	Пояснения
	Имя дизассемблируемого файла вместе с расширением

## СТРОКИ

К сожалению, встроенный язык IDA не поддерживает даже основных конструкций Си для работы со строками. Так, например, невозможно получить посимвольный доступ к строке или указатель на нее же.

Зато IDA поддерживает инициализацию и контекцию, (слияние) строк, что демонстрирует следующий пример:

```
auto a,b;  
a="Hello";  
b="IDA! \n";  
a=a+", "+b;  
Message ("%s \n", a);
```

Hello, IDA!

Таким образом, строки в IDA представляют собой закрытые объекты, доступные лишь посредством набора, манипулирующих с ними функций. Их всего три.

Это определение длины строки (strlen), взятие подстроки (substr) и поиск подстроки (strsrch). Возможность модификации строки отсутствует, и в том случае, когда возникает потребность изменить хотя бы один символ, приходится перестраивать всю строку целиком.

Для этого может пригодиться две следующие функции, которые рекомендуется включить в idc.idc, что бы сделать их доступными для всех пользовательских скриптов.

```
static setstr(str, pos, ch)  
{  
auto s0;
```

```

s0=substr(str,0,pos);
s0=s0+ch;
s0=s0+substr(str,pos+strlen(ch), strlen(str));
return s0;
}

```

```

static setstr(str, pos, ch)
{
auto s0;
s0=substr(str,0,pos);
s0=s0+ch;
s0=s0+substr(str,pos, strlen(str));
return s0;
}

```

Первая из них позволяет в строке `str` заместить любую подстроку `ch` с позиции `pos`, а вторая осуществляет вставку с «раздвижкой»

Примеры использования даны ниже, но для повседневного использования обе функции рекомендуется дополнить проверками корректности передаваемых параметров.

```

Message("%s \n",
setstr("Hello World!",5," "))
);

```

```

Hello, World!

```

```

Message("%s \n",
insstr("Hello, World!",7,"my "))
);

```

```

Hello, my World!

```

Таким образом, достаточно лишь одной функции взятия подстроки, что бы обеспечить **неограниченный** доступ к объекту строка. Для этого достаточно лишь скопировать ее в другой объект, доступный нам для чтения и записи, например, виртуальную память, массив и так далее. К сожалению, это медленно работает, но зачастую является возможным единственным выходом.

Другую группу строковых операций представляют функции всевозможного преобразования форматов. Например, перевод символьной строки в двоичное (шестнадцатеричное) число и наоборот. Однако, поскольку IDA поддерживает аналог функции `sprintf`, то чаще всего пользуются одним единственным вызовом `fmt`. Это гораздо удобнее, чем хранить в голове имена множества функций.

**char substr (char str, long x1,long x2);**

Функция осуществляет взятие подстроки. IDA не поддерживает стандартную для Си конструкцию `str[a]`, поэтому для любого посимвольного разбора строки приходится вызывать 'substr'

Функция принимает следующие операнды:

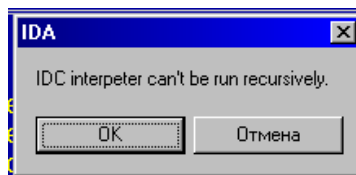
операнд	пояснение
---------	-----------

x1	индекс начала подстроки
x2	индекс конца подстроки
	если x2 == -1, то возвращается весь остаток строки целиком
Return	Пояснение
char	Подстрока

В версии 3.84 и более ранних, эта функция не имела никакого контроля над границами индексов, и если оказывалось, что  $x2 < x1$ , то Windows закрывала приложение IDA, как совершившее недопустимую операцию. Так же наблюдалась непредсказуемая работа приложения при выходе индексов за границы строки.

В версии 4.0 этот недостаток уже устранен. В случае  $x2 < x1$  функция возвращает, пустую строку, а при нарушении границ доступа (начальный индекс за границами строки) хоть и выводит диалоговое окно, сообщающие о нарушении границ доступа, но не выходит из дизассемблера, позволяя продолжить работу.

(Правда при этом попытка исполнения любого скрипта заканчивается следующей ошибкой, вплоть до перезапуска IDA)



Если конечный индекс лежит за пределами строки, то IDA просто возвращает остаток строки и аварийной ситуации не возникает.

Пример использования этой функции для построения простейшего синтаксического анализатора:

```

auto a,temp,c;
a="key -Hello";
for (temp=0;temp<strlen(a);temp++)
{
c=substr(a,temp,temp+1);
if (c=="-")
{
Message (substr(a,temp+1,-1));
break;
}
}

```

Hello

**long strstr (char str, char substr);**

Функция осуществляет поиск подстроки в строке, принимая следующие операнды:

Операнд	пояснения	
'str'	строка	
'substr'	подстрока	
Return	==return	пояснения
	!=BADADDR	Индекс подстроки в строке
	==BADADDR	Ошибка

При успешном завершении возвращается индекс начала подстроки в строке. Если

подстрока не найдена, то функция возвращает BADADDR.

Недокументированной особенностью этой функции, является возможность сравнения тождественности числовых выражений. Так, например, strstr(0x1234,0x1234) возвратит '0', а strstr(0x1234,0x123) - BADADDR, т. е. ошибку.

Поиск пустой подстроки всегда успешен; поиск в пустой строке всегда возвращает ошибку.

Пример:

```
auto a,temp;
a="key -Hello";
temp=strstr(a,"-");
if (temp!=-1) Message (substr(a,temp+1,-1));

hello
```

### long strlen (char str);

Функция возвращает длину строки. Длина ненулевой строки равна индексу последнего символа плюс единица.

операнд	Пояснение
str	Строка, заканчивающаяся нулем
Return	пояснения
	Длина строки

Перенос строки считается за один символ. Однако, при выводе строки в текстовый файл он представляется последовательностью символов 0xD 0xA, занимая, таким образом, **два** байта.

Это необходимо учитывать при позиционировании внутри файла, а так же оценке необходимого свободного пространства на диске.

Пример:

```
Message("%d \n", strlen("hello, word! \n"));

14
```

### char form (char format,...);

Ближайший аналог известной функции sprintf. Возвращает форматированную строку и преобразует длинные целые и числа с плавающей запятой в строковые значения.

Например:

```
auto a,s0;
a=0x123;
s0=form("%x \n",a);
Message(s0);

123
```

'form' не единственное средство для преобразования типов, к тому же достаточно медленно работающие. В большинстве случаев рекомендуется использовать другие

функции, преобразующие длинные целые в строку. 'form' выгодно применять только для чисел с плавающей запятой ('float').

Управляющие символы стандартные, и частично совместимые с 'printf' и полностью совместимы со спецификаторами функции Message встроенного языка IDA.

Сф	пояснение
%d	десятичное длинное знаковое целое Пример: <code>Message ("%d", 0xF);</code> 15
%x	шестнадцатеричное длинное целое строчечными символами Пример: <code>Message ("%x", 10);</code> a
%X	шестнадцатеричное длинное целое заглавными символами Пример: <code>Message ("%X", 10);</code> A
%o	восьмеричное длинное знаковое целое Пример: <code>Message ("%o", 11);</code> 13
%u	десятичное длинное беззнаковое целое Пример: <code>Message ("%u", -1);</code> 4294967295
%f	десятичное с плавающей точкой Пример: <code>Message ("%f", 1000000);</code> 1.e6
%c	символьное значение Пример: <code>Message ("%c", 33);</code> !
%s	строковое значение Пример: <code>Message ("%s", "Hello, Word! \n");</code> Hello, Word!
%e	вывод чисел в экспоненциальной форме Пример: Пример: <code>Message ("%e", 1000000);</code> 1.e6
%g	вывод чисел в экспоненциальной форме <b>ЗАМЕЧАНИЕ:</b> В оригинале спецификатор '%g' заставляет функцию саму решать, в какой форме выводить число - с десятичной точкой или в экспоненциальной форме, из соображений здравомыслия и удобочитаемости. IDA всегда при задании этого спецификатора представляет числа в экспоненциальной форме.

	<p>вывод указателя (не поддерживается)</p> <p><b>ЗАМЕЧАНИЕ:</b> вместо спецификатора '%p' IDA использует '%a', преобразующее линейный адрес в строковый сегментный, и автоматически подставляет имя сегмента.</p> <p>Так, например, 'Message("%a \n", 0x10002)' выдаст 'seg000:2'. Обратите внимание, что таким способом <b>нельзя</b> узнать адрес переменной.</p> <p>Пример:</p> <pre>auto a; a="Hello!\n"; Message("%a \n", a); 0</pre> <p>Возвращается <b>ноль</b>, а не указатель на переменную.</p>
%p	вывод десятичного целого всегда со знаком, не опуская плюс.
%+d	в оригинале - вывод шестнадцатеричного целого всегда со знаком, но ida воспринимает эту конструкцию точно так же как и 'x'.
%+x	<p>'n' длина выводимого десятичного числа, при необходимости дополняемая слева пробелами.</p> <p>Например:</p> <pre>Message("Число-%3d \n", 1); Число- 1</pre> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Например:</p> <pre>Message("число-%3d \n", 10000); Число-10000</pre>
%nd	<p>'n' длина выводимого шестнадцатеричного числа, при необходимости дополняемая слева пробелами.</p> <p>Например:</p> <pre>Message("Число-%3x \n", 1); Число- 1</pre> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Например:</p> <pre>Message("Число-%3x \n", 0x1234); Число-1234</pre>
%nd	<p>'n' длина выводимого десятичного числа, при необходимости дополняемая слева незначащими нулями.</p> <p>Пример:</p> <pre>Message("Число-%03d", 1); Число-001</pre> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Пример</p> <pre>Message("Число-%03d", 1000) Число-1000</pre>

%0nx	<p>'n' длина выводимого шестнадцатеричного числа, при необходимости дополняемая слева незначащими нулями.</p> <p>Пример:</p> <pre>Message ("Число-%03x", 0x1);</pre> <p>Число-001</p> <p>Если выводимое число не укладывается в 'n' позиций, то оно выводится целиком.</p> <p>Пример:</p> <pre>Message ("число-%03x", 0x1234);</pre> <p>Число-1234</p>
%#x	<p>Вывод префикса '0x' перед шестнадцатеричными числами</p> <p>Пример:</p> <pre>Message (" %#x", 123);</pre> <p>0x123</p>
%#o	<p>Вывод префикса '0' перед восьмеричными числами</p> <p>Пример:</p> <pre>Message (" %#o", 1);</pre> <p>01</p>
%n	Количество выведенных символов (не поддерживается)

### long xtol (char str);

Функция преобразовывает строковое шестнадцатеричное значение str в длинное целое.

операнд	Пояснение	
str	Строковое шестнадцатеричное число	
Return	==return	пояснения
		длинное целое
	==0	Ошибка

Разрешается использовать спецификаторы 'x' и 'h' для указания системы исчисления, в противном случае будет использоваться система исчисления по умолчанию (как правило, шестнадцатеричная).

Функция понимает одиночный знак '+' или '-', но "спотыкается" на '+-1' или на префиксе шестнадцатеричных чисел '\$', используемых языком Pascal.

Числом считается все до первого нецифрового символа (кроме "A-F"). Так, например:

```
Message ("0x%X \n",
        xtol ("123-2"))
);
```

0x123

В случае ошибки возвращается ноль. Так, например.

```
Message ("0x%X \n",
        xtol ("*1"))
);
```



0x0

### char atoa (long ea);

Функция преобразовывает линейный адрес в строку, используя символьные имена сегментов, выбирая сегмент с наибольшим из возможных адресов.

операнд	Пояснение	
ea	32-разрядный линейный адрес	
Return	==return	пояснения
	!=	Сегментный адрес в строковом представлении
	==	Ошибка

Так, например:

```

+-[_]----- Program Segmentation -----3-[ ]--+
|   Name      Start      End   Align Base Type Cls  32es  Start EA End EA  |
|  seg000     00000000 000032EA byte  1000 pub CODE  N FFFF 00010000 000132EA  0_ |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|
+1/1      [ ]-----|-----|-----|-----|-----|-----|-----|-----|-----+

```

```

Message("%s \n",
        atoa(0x10000)
        );

```

seg000:0

Если указанный адрес не принадлежит ни одному сегменту, то функция преобразует его к виду segment : offset, по стандартной формуле преобразования:

$$\text{segment} = \text{ea} \gg 4;$$

$$\text{offset} = \text{ea} - (\text{ea} \gg 4).$$

Например:

```

Message("%s \n",
        atoa(0x18)
        );

```

1:00000008

### char ltoa (long n,long radix);

Функция преобразовывает длинное целое в символьное с произвольной системой исчисления.

Функция принимает следующие операнды:

операнд	назначение
n	Задаёт операнд
	==n   Операнд

	0	первый слева операнд
	1	Второй, третий и остальные
	-1	все операнды
radix	требуемая система исчисления. <b>ЗАМЕЧАНИЕ:</b> В контекстной помощи IDA сообщается, что 'radix' может принимать значения 2, 8, 10, 16. Однако это стандартная Си-функция, и она может принимать и другие значения, например, 3 или 11. Точнее, все кроме 0 и 1, а так же не более 24, при которых, независимо от аргумента, функция возвращает пустую строку.	
Return	==return	пояснения
	!=""	Сегментный адрес в строковом представлении
	=="	Ошибка

Пример:

```
auto a;
for (a=0;a<50;a++)
Message ("%x, %s \n",
a,
ltoa(a,a));
```

```
0, 1, 2, 10 3, 10 4, 10 5, 10 6, 10 7, 10 8, 10 9, 10 a, 10
b, 10 c, 10 d, 10 e, 10 f, 10 10, 10 11, 10 12, 10 13, 10 14, 10 15, 10
16, 10 17, 10 18, 10 19, 10 1a, 10 1b, 10 1c, 10 1d, 10 1e, 10 1f, 10 20, 10
21, 10 22, 10 23, 10 24, 10 25, 26, 27, 28, 29, 2a, 2b,
2c, 2d, 2e, 2f, 30, 31,
```

### long atol (char str);

Функция преобразует строковое десятичное значение в длинное целое. Постфикс 'h' игнорирует, а, встретив префикс 'x', возвращает нулевое значение.

операнд	Пояснение	
srt	Строка, заканчивающаяся нулем	
Return	==return	пояснения
		длинное целое
	==0	Ошибка

Пример:

```
Message ("%x \n",
atol ("16")
);
```

10

### РАЗНОЕ

### **char Compile (char filename);**

Очень полезная функция, компилирующая в память указанные файлы (скрипты IDA). Все функции, объявленные в них, будут глобальными и доступными всем остальным скриптам вплоть до завершения сеанса работы с IDA.

Обычно так главный модуль скрипта подключает все остальные необходимые ему функции, находящиеся во внешних файлах. Обратите внимание, что 'Compile' *только* компилирует, но не запускает функцию 'main()'. Если модуль требует инициализации, то 'main' придется вызвать вручную. Однако в компилируемом файле не должно содержаться функции, совпадающей по имени с текущей (то есть вызвавшей 'Compile'). Иначе поведение дизассемблера окажется непредсказуемым. В остальных случаях совпадение имен откомпилированных функций приводит лишь к их замещению.

Заметим, что системные функции таким образом перекрыть не удастся и выдастся сообщение об ошибке. Любопытно, что 'Compile' возвращает не код ошибки, а осмысленную символьную строку. В случае успеха операции - пустую.

Пример:

demo.idc:

```
static MyFunc()
{
    Message("hello, sailor! \n");
}
```

compl.idc

```
static main()
{
    Compile("demo.idc");
    MyFunc();
}
```

Загрузим и запустим на выполнение 'compl.idc' На экране появиться 'Hello, Sailor!'. Нетрудно убедиться, что функция 'MyFunc' доступна и с консоли и с остальных скриптов.

Заметим, что IDA сложным образом манипулирует с понятием текущего каталога, поэтому если приведенный пример не сработает, попробуйте указать полный путь к файлу 'demo.idc'

### **void Exit (long code);**

Функция осуществляет выход из IDA в операционную систему, сохраняя все последние изменения и закрывая базу.

Выполнение скрипта при этом прерывается. Закрывание активных файлов не гарантируется.

Операнд	назначение
code'	код завершения процесса

## long Exec (char command);

Функция позволяет выполнить команду операционной системы, не выходя из IDA.

Операнд	Пояснение
command	Команда операционной системы (как правило командного интерпретатора command.com)

Эта команда расширяет возможности скриптов. Можно, например, удалять временные файлы, или склеивать их командой 'copy' и делать многие другие вещи, в том числе писать вирусы и троянские компоненты. (Поэтому с присланными вам скриптами сомнительного происхождения следует вести себя крайне осторожно)

Для запуска параллельного процесса предусмотрена команда 'start', например `Exec("start command.com")`. В Windows приложение запустится в соседнем окне. В этом случае функция возвращает не код завершения процесса (поскольку таковой еще не известен), а результат исполнения программы, использующейся операционной системой для запуска приложения. Для консольных программ, это как правило 'copagent'

Крайне рекомендуется проверять все скрипты, полученные из чужих рук на наличие этой команды.

Пример использования:

```
Exec ("command.com /c del *.tmp");
```

## ПРИЛОЖЕНИЯ

### КРАТКО О ЗАГРУЗКЕ ФАЙЛОВ

#### ОПЦИИ КОМАНДНОЙ СТРОКИ

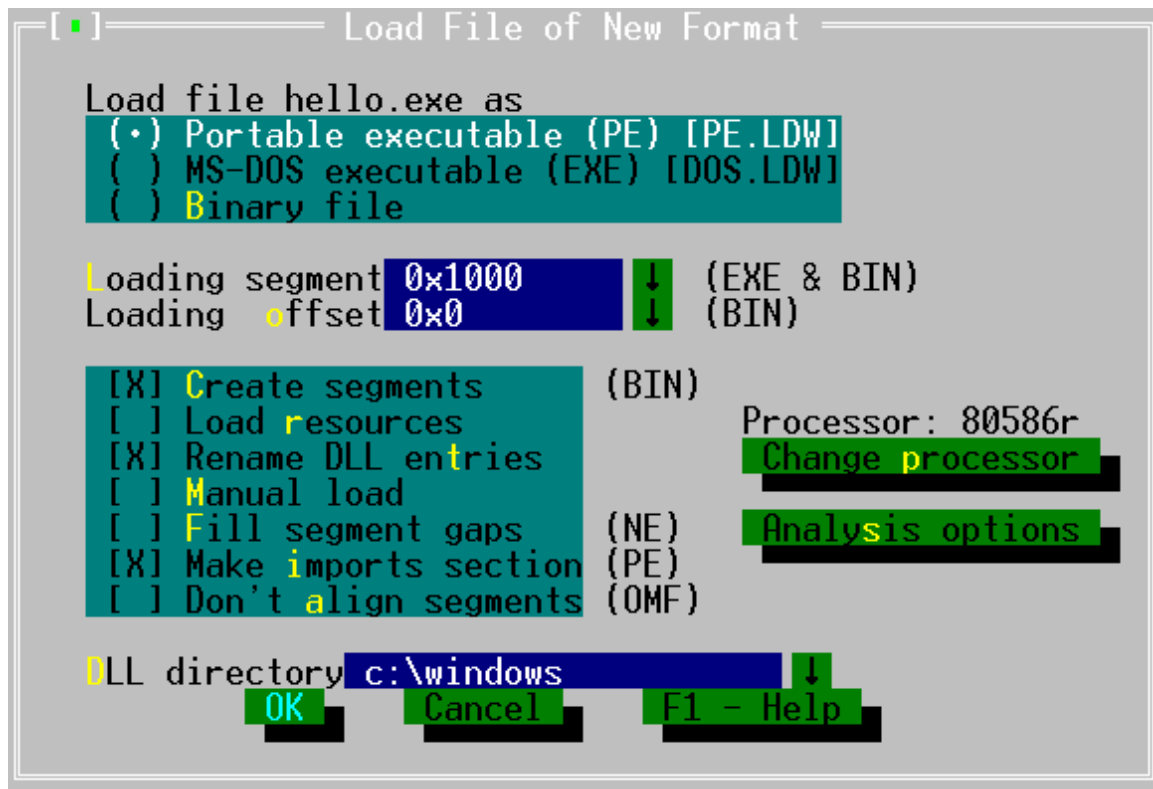
Любопытной особенностью IDA является то, что она поддерживает zip формат и позволяет работать непосредственно с упакованными файлами. Если имя файла не указано в командной строке, то IDA сама запросит его при запуске. В процессе дизассемблирования IDA не работает с выбранным файлом (и его безболезненно можно удалить), а создает набор файлов ID? - собственной базы данных - с которой и взаимодействует.

Поэтому при повторных загрузках файла грузиться не он сам, а созданная ранее база. Понимание этого крайне важно, т.к. при этом не отслеживаются никакие модификации файла, как уже упоминалось исследуемый файл однократно загружается в базу и в дальнейшем не принимает никакого участия в работе.

Для особых ситуаций предусмотрен ключ командной строки '-c', удаляющий перед запуском ранее созданные дизассемблером базы. Разумеется вместе со всеми заботливо созданными вами комментариями, именами, метками... уничтожая всю проделанную работу. Конечно, это не может служить решением проблемы - необходимо

перегрузить только модифицированный фрагмент, а не уничтожить всю базу! И хотя штатно такая возможность не была предусмотрена, встроенный язык IDA позволяет написать подобный загрузчик самостоятельно.

При первой же загрузке файла появится следующий диалог:



```
( ) Portable executable (PE) [PE.LDW]
( ) MS-DOS executable (EXE) [DOS.LDW]
( ) Binary file
```

В большинстве случаев IDA автоматически определяет тип загружаемого файла, но все же оставляет конечное решение за Вами. Это действительно часто бывает полезно во многих случаях - например вы всегда можете загрузить файл как бинарный и работать с его структурами вручную. Например текущая версия IDA не поддерживает само-загружающиеся модули и завершает работу при попытке загрузки онных.



Однако, хоть и редко, но с такими файлами исследователь рано или поздно сталкивается. Единственный выход загружать файл как бинарный и дизассемблировать его с помощью собственных скриптов. Не очень вдохновляющая перспектива конечно, но можно утешить себя по крайней мере тем, что другие дизассемблеры не умеют и этого.

Остается только надеяться, что рано или поздно разработчики разрешат эту проблему, но и сегодня IDA обгоняет остальные дизассемблеры числом и качеством поддерживаемых форматов файлов.

EXE	Исполняемый файл	MS-DOS
COM	Исполняемый файл	MS-DOS, CP\M
SYS	Устанавливаемый драйвер	MS-DOS
NE	New Executable Format	Windows 3.x; OS/2
LX	Linear Executable Format	OS/2 2.x and OS/2 Warp
LE	Linear Executable Format	Windows VxD
PE	Portable Executable Format	Windows 95; Windows NT
OMF	Intel Object Module Format	MS DOS; Windows 3.x; OS/2
LIB	Library of Object Files	MS DOS, MS Windows 3.x; OS/2.
AR	Library of Object Files	UNIX, MS Windows95; MS Windows NT
COFF	Common Object File Format	UNIX
NLM	Novell Netware Loadable Modules	
ZIP	Archive files.	
JAVA	Java classes	

В действительности этот список не исчерпывает возможности IDA, гибкая архитектура которой позволяет работать с произвольными типами файлов, кроме того с каждой версией число поддерживаемых форматов все увеличивается.

```

Loading segment 0x1000    (EXE & BIN)
Loading offset 0x0       (BIN)

```

Сегмент и смещение загрузки на первом этапе освоения IDA лучше не изменять. Смещение актуально только для BIN файлов. Например для дизассемблирования дампа MBR сектора его необходимо загрузить со смещением 0x7C00 иначе все смещения будут указывать в "космос". Для типотизированных файлов IDA игнорирует установленное смещение загрузки, извлекая эту информацию из соответствующих полей заголовка.

Базовый адрес сегмента на самом деле не имеет никакого отношения к исследуемому файлу, а к виртуальной памяти IDA. Его значение необходимо только при написании собственных скриптов, а в остальных случаях оно никак не отразится процессе дизассемблирования.

```
[X] Create segments
```

Создавать или не создавать сегмент для бинарных файлов. По умолчанию всегда создается по крайней мере один сегмент. Эту можно выключить когда расставить сегменты вы хотите по своему усмотрению (например, при анализе само-загружаемых модулей). Но создать хотя бы один сегмент необходимо в любом случае. Внутренняя архитектура IDA такова, что большинство команд работает только с сегментами. Попытка дизассемблировать фрагмент, не принадлежащий ни к одному из сегментов вызовет протест со стороны IDA:

```
This command can't be applied to the addresses without a segment.  
Create a  
segment first.
```

Однако, другие команды, такие например, чтение\запись памяти будут успешно работать и могут быть использованы, например, в скрипте расшифровки файла. В этом случае, разумеется в создании сегмента никакой необходимости нет.

#### [ ] Load resources

Указывает на необходимость загрузки ресурсов. Актуально только для PE и NE файлов. По умолчанию выключено, однако довольно часто в ресурсах расположены текстовые строки или даже некоторые данные, и в этих случаях ресурсы необходимо загрузить. В остальных же случаях это только лишний расход времени и памяти.

#### [X] Rename DLL entries

IDA умеет распознавать большинство библиотечных функций (подробнее об этой уникальной возможности в главе посвященной технологии FLIRT). При этом она может заменять импортируемые по ординалу функции их непосредственными именами. Если же по каким-то причинам для вас это не приемлемо, то данную опцию необходимо отключить. При этом символьные имена IDA добавит в повторяемые комментарии. Сравните:

```
; Imports from MFC42.DLL  
?DoMessageBox@CWinApp@@UAENPBDII@Z dd ?  
?SaveAllModified@CWinApp@@UAENXZ dd ?  
?InitApplication@CWinApp@@UAENXZ dd ?
```

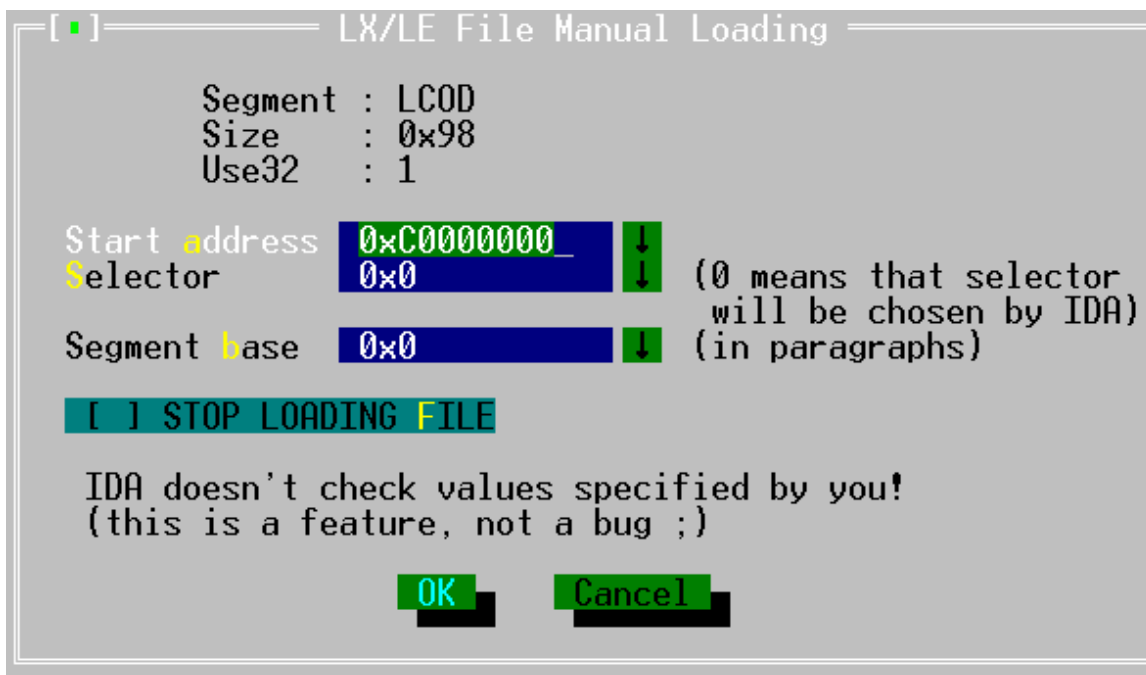
и

```
; Imports from MFC42.DLL  
MFC42_2512 dd ? ; DATA XREF: j_MFC42_2512□r  
;  
?DoMessageBox@CWinApp@@UAENPBDII@Z:  
MFC42_5731 dd ? ; DATA XREF: j_MFC42_5731□r  
;  
?SaveAllModified@CWinApp@@UAENXZ:  
MFC42_3922 dd ? ; DATA XREF: j_MFC42_3922□r  
;  
?InitApplication@CWinApp@@UAENXZ:  
MFC42_1089 dd ? ; DATA XREF: j_MFC42_1089□r
```

Разумеется, гораздо удобнее когда IDA ординалы заменяет осмысленными именами и найдется немного ситуаций, в которых эту опцию приходится отключать.

[ ] Manual load

"Ручная" загрузка некоторых типов файлов. Главным образом полезна для NE\LX\LE форматов. При этом пользователь получает возможность для каждого из объектов файла задать селектор и базовый адрес загрузки. Появится диалог следующего вида:



'Start Address' указывает по какому адресу будет расположен загружаемый объект. Это значение вычисляется дизассемблером автоматически и обычно нет причин менять его. Подробнее об этом будет рассказано в главе, посвященной анализу vxd файлов.

Селектор, разумеется не имеет никакого отношения к исследуемому файлу, а непосредственно к дизассемблеру, а точнее организации внутренней (виртуальной) памяти IDA. Сейчас мы не будем задерживать на этом внимание, отметим только, что последний необходим для доступа к сегментам, написанных вами скриптов, а в остальных случаях его значение будет "прозрачно" для пользователя.

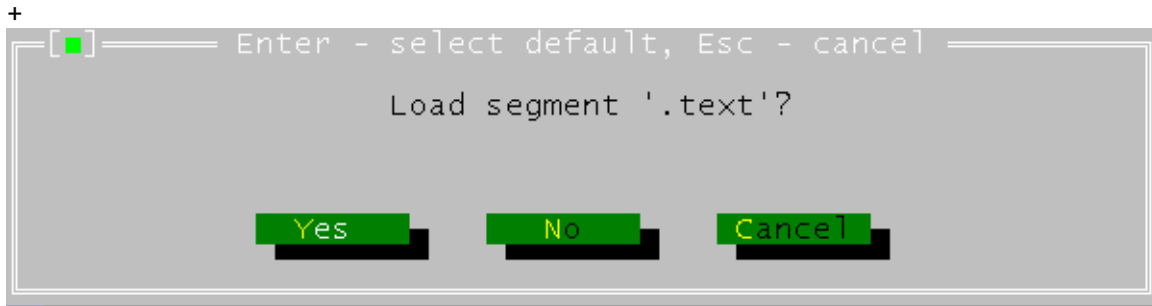
Базовый адрес связан с виртуальным адресом следующей формулой:

$$\text{VirtualAddress} = \text{LinearAddress} - (\text{SegmentBase} \ll 4)$$

Т.е. одному и тому же виртуальному адресу могут соответствовать различные пары SegmentBase:LinearAddress. Подробности в главе, посвященной организации виртуальной памяти IDA.

'Stop loading file' разумеется означает прекращение загрузки остальных объектов (сегментов) файла. Может быть использовано для экономии времени - если остальные объекты/секции вас не интересуют, то к чему тратить время\ресурсы на их загрузку?





PE файлы имеют более простую организацию, в первом приближении представляя собой просто образ памяти процесса. Ручная загрузка сводится только к произвольному выбору загружаемых секций. Ни базовый адрес, ни адрес загрузки изменить невозможно.

[X] `Make imports section`

По умолчанию IDA преобразует секцию импорта `.idata` PE файлов в набор директив `'extern'` и отсекает ее. Обычно это работает нормально, но никто не гарантирует, что в секции импорта не окажется размещенными некоторые данные. Так, например, поступают некоторые вирусы, размещая свое тело в таблице адресов. Разумеется, при этом IDA их "не увидит", в таких случаях `'Make imports section'` следует отключить.

На этом обзор описания загрузки файлов будем считать законченным. Интерактивной работе с IDA посвящен второй том этой серии.