Chris Eagle “THE IDA PRO BOOK” 2nd edition
перевод 19-й главы (ПРОЦЕССОРНЫЕ МОДУЛИ)

Процессорные модули IDA

Последним внешним компонентом IDA (*в предыдущих главах рассматривались подключаемые модули или плагины - plug-ins, и модули загрузки или загрузчики - loaders - от перев.*), который возможно создать, используя SDK, являются процессорные модули, они гораздо сложнее из всех существующих типов модулей IDA. Процессорные модули (*для краткости буду обозначать их как ПМ - от перев.*) ответственны за все операции дизассемблирования, которые имеют место быть внутри IDA. Помимо обычного преобразования с языка машинных опкодов в их соответствующее представление на языке ассемблера, ПМ также отвечают за такие задачи, как создание функций, генерацию перекрестных ссылок, и отслеживание поведения стекового указателя. Как было показано в предыдущих главах, компания Hex-Rays предоставляет возможность (начиная с IDA 5.7) создавать ПМ с помощью одного из скриптовых языков IDA (*имеется в виду поддержка скриптов на языке Python, с помощью них возможно написать любой внешний компонент - будь то плагин, загрузчик или ПМ - от перев.*).

Обычная причина, по которой требуется разработка ПМ - это реверс бинарного кода, для которого не существует ПМ. Среди прочего, таким бинарным кодом могут быть прошивки для встроенных микроконтроллеров или исполняемые образы вытащенные из карманных устройств. Не очевидный случай использования ПМ - дизассемблирование инструкций пользовательских виртуальных машин, встроенных для обфускации выполняемого кода. В таких случаях существующий ПМ IDA такой как pc-модуль x86 поможет Вам понять только саму виртуальную машину; он не предложит помощь в том, как дизассемблировать байт-код, выполняемый виртуальной машиной. Рольф Роллз (Rolf Rolles) продемонстрировал именно такое применение ПМ в своей статье, опубликованной на OpenRCE.org (*смотри “Defeating HyperUnpackMe2 With an IDA Processor Module”* www.openrce.org/articles//full_view/28) В приложении В своей статьи, Рольф делится своими мыслями о создании ПМ IDA; это один из немногих документов, имеющихся по данному вопросу.

В мире модулей IDA, есть бесконечное множество мыслимых и немыслимых применений плагинов, и после скриптов (*те самые, что IDC command и Python command - от перев.*), плагины безусловно наиболее широко доступные сторонние дополнения для IDA. Необходимость в пользовательских модулях загрузки значительно меньше, чем в плагинах. В этом нет ничего странного, так как бинарных файловых форматов (и следовательно необходимых загрузчиков) существует намного меньше, чем число разнообразных применений плагинов. Естественным результатом, благодаря которому внешние модуля распространяются бесплатно с IDA, является то, что существует, как правило, сравнительно мало опубликованных сторонних модулей загрузки. Еще меньше нужда в ПМ, т.к. число наборов инструкций, требуемых для декодирования, меньше, чем количество файловых форматов, которые используют эти наборы инструкций. И в итоге получаем неутешительный результат почти полного отсутствия сторонних ПМ, кроме тех немногих, которые идут в поставке с IDA и с SDK. Но, судя по темам постов на форумах Hex-Rays, станет ясно, что люди работают над ПМ; просто эти ПМ не выходят в паблик (не для широкой публики).

В этой главе, мы надеемся пролить дополнительный свет на тему создания ПМ для IDA и развеять мифы (по крайней мере отчасти) последнего модульного компонента IDA. В качестве рабочего примера мы разработаем ПМ для дизассемблирования байт-кода языка Python. Поскольку компоненты ПМ могут быть довольно большими, то нет возможности включить в книгу полные листинги каждого куска модуля. Полный исходный код ПМ Python доступен на сайте книги (*www.idabook.com*). Важно понимать, что без модуля загрузки Python невозможно полное автоматическое дизассемблирование откомпилированных *.pyc* файлов. В отсутствии такого загрузчика, Вам необходимо загрузить *.pyc* файл вручную в бинарном режиме, выбрать ПМ Python, выбрать подходящий на Ваш взгляд стартовый адрес для начала функции и

преобразовать байты на экране в инструкции Python, использую команду “Edit” -> “Code”.

БАЙТ-КОД PYTHON

Python (*www.python.org*) объектно-ориентированный, интерпретируемый язык программирования. Python часто используют для решения скриптовых задач в схожем стиле языка Perl. Исходные модули Python - это текстовые файлы в основном имеющие расширение *.py*. Всякий раз, когда скрипт Python выполняется, интерпретатор компилирует исходный код во внутреннее представление, известное как *байт-код Python (docs.python.org/library/dis.html#bytecodes - здесь лежит полный список кодов инструкций, также смотрите opcode.h в исходниках поставки Python, в котором указана мнемоника кода и соответствующий ему опкод)*. Этот опкод в конце концов интерпретируется виртуальной машиной. В целом процесс отчасти аналогичен тому, как исходный код Java компилируется в байт-код Java, который в конце концов выполняется на виртуальной машине Java. Основным отличием является то, что Java-пользователи должны явно компилировать свои исходники на Java в байт-код, в то время как в Python исходный код неявно преобразуется в байт-код каждый раз, когда пользователь собирается выполнить Python скрипт.

Для того, чтобы избегать повторной трансляции исходника в байт-код, интерпретатор Python может сохранить представление исходного кода в *.pyc* файле, который дает возможность загрузки напрямую с последующим выполнением, экономя время на трансляции. Обычно пользователи явно не создают *.pyc* файлы. Взамен, Python интерпретатор автоматически создает *.pyc* файлы для любого исходного модуля, который импортируется другим исходником. Таким образом, если модули используются повторно очень часто, то Вы можете сэкономить время, если байт-код будет легко доступен. Байт-код файлы (*.pyc*) грубо эквивалентны Java .class файлам.

Учитывая, что интерпретатору Python не требуется исходный код, когда соответствующий файл байт-кода доступен, то можно распространять некоторые части Python проекта в виде байт-кода, а не в качестве исходника. В таких случаях был бы полезен реверс файлов байт-кода, чтобы понять, что они делают, так же, как мы могли бы сделать реверс с любым другим программным обеспечением, распространяемым в бинарном коде. Эту цель и преследует наш пример ПМ Python - предоставить инструмент для реверса байт-кода Python.

ИНТЕРПРЕТАТОР PYTHON

Немного общей информации об интерпретаторе Python может быть полезна тем, как мы разрабатывали ПМ Python. Интерпретатор Python реализует стековую виртуальную машину, способную выполнять Python байт-код. Под *стековой*, мы имеем в виду то, что виртуальная машина не имеет регистров, кроме указателя инструкций и указателя стека. Большинство инструкций байт-кода Python управляют стеком с помощью операций чтения, записи, или проверки содержимого стека. Например, инструкция байт-кода **BINARY_ADD**, удаляет два элемента из стека интерпретатора, складывает их, и результат размещает обратно на вершину стека интерпретатора.

С точки зрения устройства набора инструкций, байт-код Python относительно прост для понимания. Все инструкции Python состоят из одно байтового опкода и либо нулевого или двухбайтового операнда. Пример ПМ, представленного в этой главе не требует от Вас каких-либо предварительных знаний о байт-коде Python. В нескольких случаях, когда потребуется специфическое знание, у нас будет время, чтобы достаточно объяснить необходимый байт-код. Главная цель этой главы дать общее представление об ПМ IDA и некоторые соображения при его создании. И байт-код Python здесь всего лишь простой способ достижения этой цели.

П И Ш Е М П Р О Ц Е С С О Р Н Ы Й М О Д У Л Ь С П О М О Щ Ь Ю S D K

Было бы правильно начать обсуждение создания ПМ не учитывая стандартную отговорку о том, что документации, касающейся ПМ очень мало. Кроме прочтения SDK, включая файлы и исходный код процессорных модулей, идущих в поставке с SDK, Вы заметите, что файл *readme.txt* единственный файл, который проливает какой-то свет на процесс создания ПМ, а это всего лишь несколько комментариев под заголовком “DESCRIPTION OF PROCESSOR MODULES”.

Стоит уточнить, что README файл ссылается на конкретные имена файлов внутри ПМ так, как если бы эти имена были жестко заданы, хотя на самом деле они таковыми не являются. Однако, как правило, эти имена файлов встречаются повсюду в примерах SDK, и именно они указаны в сценариях сборки для этих примеров. Вы вправе создать свой ПМ с использованием любых имен файлов, которые Вам нравятся, но необходимо будет обновить имена в сценариях сборки соответственно.

Основное значение ссылок на конкретные имена файлов ПМ - передать идею о том, что ПМ состоит из трех логических частей: анализатора, эмулятора инструкций, и генератора печати. Мы рассмотрим назначение каждого компонента по ходу создания нашего ПМ Python.

Несколько примеров процессоров Вы найдете в *<SDKDIR>/module.* Один из простейших процессоров - это процессор z8. Другие процессорные модули различаются по сложности в зависимости от их наборов инструкций и от обязанностей, которые они берут на себя при загрузке. Если Вы думаете о написании собственных ПМ, один из подходов для начала работы (рекомендуемый Ильфаком в файле README) является копирование существующего ПМ и изменения его в соответствии с вашими потребностями. В таком случае, Вам нужно найти именно тот ПМ, который больше всего напоминает логическую структуру (не обязательно близок по архитектуре процессора) Вашего предполагаемого модуля.

Структура processor_t

ПМ также как и плагины, и загрузчики экспортирует ровно одну вещь. Для ПМ - это структура **processor_t** с обязательным именем **LPH**. Она экспортируется автоматически, если Вы добавите включаемый файл *<SDKDIR>/module/idaidp.hpp*, который, в свою очередь, включает в себя множество других заголовочных файлов SDK обычно необходимых для процессорных модулей. Одна из причин, почему написание ПМ является настолько сложной задачей заключается в том, что структура содержит 56 полей, которые должны быть инициализированы, и 26 из них должны содержать указатели на функции, в то время как одно из полей есть указатель на массив из одного или нескольких указателей на структуру, в которой каждое поле указывает на другой тип структуры (asm_t), содержащей 59 полей, требующих инициализации. Не правда ли просто? Одно из принципиальных неудобств при сборке процессорных модулей крутится вокруг инициализации всех необходимых статических данных, процесс, подверженный ошибкам из-за большого количества полей в каждой структуре данных. Это одна из причин, почему Ильфак рекомендует использовать уже существующие ПМ в качестве шаблона для новых разрабатываемых Вами процессоров.

Из-за сложности этих структур данных, мы не будем пытаться перечислить все возможные поля и их применения. Взамен, мы выделим основные поля, а за более подробной информацией об этих или других полях в рамках каждой структуры смотрите *idp.hpp*. Порядок, в котором мы рассмотрим различные поля структуры **processor_t** совсем не похож на порядок, в котором те поля объявлены в **processor_t**.

Основные моменты инициализации структуры LPH

Перед тем как разбираться с аспектами работы Вашего ПМ, есть несколько обязательных статических данных, о которых надо позаботится. Так как Вы пишите модуль, связанный с дизассемблированием, то Вам нужно создать список команд-мнемоник на языке ассемблера, обрабатываемых Вашим процессором. Список оформлен в форме массива структур **instruc_t** (определен в *idp.hpp*) и обычно располагается в файле именуемым *ins.cpp*. Как показано ниже, **instruc_t -** это простая структура с двойным предназначением. Во-первых, она предоставляет поиск по таблице мнемоник. Во-вторых, она описывает некоторые базовые характеристики каждой команды.

```
struct instruc_t {
        const char *name; //мнемоника инструкции
        ulong feature;      //битовое поле CF_xxx флажков, определенных в idp.hpp
};
```

Поле **feature** используется для обозначения поведения инструкции, например, отвечает ли инструкция за запись или за чтение любого из его операндов и как продолжится выполнение кода, когда инструкция выполнится (default, jump, call). Аббревиатура CF в CF_xxx обозначает *canonical feature*. Поле **feature** используется в основном для общего представления потока управления и перекрестных ссылок. Некоторые из наиболее интересных флажков *canonical feature* описаны здесь:

CF_STOP Инструкция не передает управление следующей инструкции. Возможные примеры - абсолютный переход (jump) или инструкция возврата из функции (return).

CF_CHGn Инструкция изменяет операнд **n**, где **n** от 1 до 6.

CF_USEn Инструкция “использует” операнд **n**, где **n** от 1 до 6 и “использует” означает “читает” или “ссылается на” (но не модифицирует; смотри CF_CHGn) ячейку памяти

CF_CALL Команда вызова функции

Инструкции в списке не обязательно перечислять в каком-либо порядке. В частности, нет необходимости располагать инструкции в порядке равным их двоичным опкодам, и нет никаких требований на однозначное соответствие между инструкциями в этом списке (массиве) и действительными двоичными опкодами. Первые и последние несколько строк нашего массива инструкций показаны ниже:

```
instruc_t Instructions[] = {
        {"STOP_CODE", CF_STOP}, /* 0 */
        {"POP_TOP", 0}, /* 1 */
        {"ROT_TWO", 0}, /* 2 */
        {"ROT_THREE", 0}, /* 3 */
        {"DUP_TOP", 0}, /* 4 */
        {"ROT_FOUR", 0}, /* 5 */
        {NULL, 0}, /* 6 */
        {NULL, 0}, /* 7 */
        {NULL, 0}, /* 8 */
        {NOP, 0}, /* 9 */
        …
        {"CALL_FUNCTION_VAR_KW", CF_CALL}, /* 142 */
        {"SETUP_WITH", 0}, /* 143 */
        {"EXTENDED_ARG", 0}, /* 145 */
```

```
        {"SET_ADD", 0}, /* 146 */
        {"MAP_ADD", 0} /* 147 */
};
```

В нашем примере из-за того, что байт-код Python так прост мы будем придерживаться однозначного соответствия между инструкцией и ее байт-кодом. Обратите внимание, что для того, чтобы сделать это, некоторые инструкции записываем таким образом: **{NULL, 0}** (на случай, когда байт-код не определен).

Перечисление, в котором хранятся именованные константы инструкций (как правило, оно определено в *ins.hpp*) обеспечивает проецирование чисел на инструкции, как показано ниже:

```
enum python_opcodes {
        STOP_CODE = 0,
        POP_TOP = 1,            //удалить верхний элемент со стека
        ROT_TWO = 2,            //поменять местами два верхних элемента на стеке
        ROT_THREE = 3,          //сдвинуть верхний элемент ниже 2-го и 3-го элементов
        DUP_TOP = 4,            //дублировать верхний элемент на стеке
        ROT_FOUR = 5,                   //сдвинуть верхний элемент ниже 2-го, 3-го и 4-го элементов
        NOP = 9,                //нет операции
        …
        CALL_FUNCTION_VAR_KW = 142,
        SETUP_WITH = 143,
        EXTENDED_ARG = 145,
        SET_ADD = 146,
        MAP_ADD = 147,
        PYTHON_LAST = 148
};
```

Здесь мы предпочли точное назначение числа каждому значению перечисления, как для ясности, так и потому, что есть пробелы в нашей последовательности и потому, что мы взяли только актуальные опкоды Python в роли индексов наших инструкций. Была добавлена дополнительная константа (PYTHON_LAST) для быстрого поиска конца списка. Имея список инструкций и связанное с ним перечисление, нам достаточно информации проинициализировать три поля структуры **LPH** (наш глобальный **processor_t**):

```
        int instruc_start;      //целочисленный код 1-й инструкции              (STOP_CODE)
        int instruc_end;        //целочисленный код последней инструкции + 1    (PYTHON_LAST)
        instruc_t *instruc;     //список инструкций                             (Instructions)
```

Мы должны эти поля установить в STOP_CODE, PYTHON_LAST, Instructions соответственно. Вместе эти поля позволяют ПМ быстро находить мнемонику любой инструкции при дизассемблировании (*например, Instructions[STOP_CODE] -> {“STOP_CODE”,CF_STOP}*).

Также для большинства ПМ мы должны определить массив имен регистров и перечисление, в котором хранятся константы регистров. Если бы мы писали x86 ПМ, то мы начали бы с чем-то вроде следующего, где для краткости мы ограничимся основным набором x86 регистров:

```
static char *RegNames[] = {
        "eax", "ebx", "ecx", "edx", "edi", "esi", "ebp", "esp",
        "ax", "bx", "cx", "dx", "di", "si", "bp", "sp",
```

```
        "al", "ah", "bl", "bh", "cl", "ch", "dl", "dh",
        "cs", "ds", "es", "fs", "gs"
};
```

Массив RegNames, как правило, объявлен в reg.cpp. В этом файле, в котором также объявлен LPH, переменная RegNames должна быть статическая (*статические переменные объявляются директивой static. Они инициализируются только один раз - при первом вызове функции и сохраняют свое значение даже после выхода из функции - от перев.*). Перечисление, в котором хранятся константы регистров, должно быть объявлено в заголовочном файле, обычно его название совпадает с именем процессора (в этом случае *x86.hpp*):

```
enum x86_regs {
        r_eax, r_ebx, r_ecx, r_edx, r_edi, r_esi, r_ebp, r_esp,
        r_ax, r_bx, r_cx, r_dx, r_di, r_si, r_bp, r_sp,
        r_al, r_ah, r_bl, r_bh, r_cl, r_ch, r_dl, r_dh,
        r_cs, r_ds, r_es, r_fs, r_gs
};
```

Убедитесь в правильности соответствия между массивом имен регистров и значениями перечисления (*например, RegNames[r_eax] должно вовозвращать "eax", а не "ebx"*). Все вместе массив и перечисление разрешают ПМ быстро искать имя регистра при выводе операндов инструкции. Эти данные инициализируют два дополнительных поля в **LPH**:

```
        int regsNum;            // общее число регистров        (qnumber(RegNames))
        char **regNames;        // массив имен регистров        (RegNames)
```

Эти поля часто устанавливают равными qnumber(RegNames) и RegNames соответственно, где qnumber - это макрос, определенный в *pro.h*, который вычисляет количество элементов в статическом выделенном массиве.

Процессорному модулю IDA необходимо всегда указывать информацию о сегментных регистрах независимо от того, а есть ли эти самые регистры у процессора или нет. Так как x86 использует сегментные регистры, то в нем с настройкой все довольно просто. Сегментные регистры устанавливаются в следующих полях структуры **processor_t**:

```
① // Информация о сегментных регистрах (используйте виртуальные CS и DS регистры, если
// Ваш процессор не имеет сегментных регистров):
int regFirstSreg; // номер 1-го сегментного регистра
int regLastSreg; // номер последнего сегментного регистра
int segreg_size; // размер сегментного регистра в байтах

② // Если Ваш процессор не использует сегментные регистры, то Вы должны определить
// 2 виртуальных сегментных регистра для CS и DS.
// Допустим назовем их rVcs и rVds.
int regCodeSreg; // номер CS регистра
int regDataSreg; // номер DS регистра
```

Инициализируя наш гипотетический x86 ПМ, мы заполним вышеописанные 5 полей таким образом:

```
r_cs, r_gs, 2, r_cs, r_ds
```

Обратите внимание на комментарии ① и ② о сегментных регистрах. IDA всегда ожидает информацию об этих регистрах, даже если процессор их не поддерживает. Возвращаясь к нашему ПМ Python, заметим, что в нашем случае нет надобности создавать регистры, т.к. интерпретатор Python имеет стековую архитектуру и в нем нет таковых, но мы должны решить вопрос с сегментным регистром. Типичный подход к решению этого вопроса - создать минимальный набор сегментных регистров (для кода и данных). В принципе, мы создаем фиктивные сегментные регистры из-за того, что IDA ожидает их наличие. Тем не менее, даже если IDA ожидает их, мы отнюдь не обязаны их использовать, так что мы просто игнорируем их в нашем ПМ. Для нашего ПМ Python сделаем следующее:

```
//в reg.cpp
static char *RegNames = { "cs", "ds" };

//в python.hpp
enum py_registers { rVcs, rVds };
```

А теперь инициализируем поля в LPH:

```
rVcs, rVds, 0, rVcs, rVds
```

Перед тем как перейти к реализации какого-либо поведения ПМ Python, мы уделим немного времени оставшимся мелочам, которые устанавливаются в LPH. Первые пять полей структуры processor_t описаны ниже:

```
int version; // должно быть IDP_INTERFACE_VERSION
int id; // IDP id, PLFM_xxx значение или самоназначенное значение > 0x8000
ulong flag; // Возможности процессора, логическое поле флажков PR_xxx
int cnbits; // Число бит в байте для кодового сегмента (обычно 8)
int dnbits; // Число бит в байте для сегмента данных (обычно 8)
```

Поле version выглядит знакомым, так как оно обязательное и для плагинов и для загрузчиков, и уже встречалось в предыдущих главах. Для пользовательских модулей процессора, поле id должно быть самоназначенным значением большим, чем 0x8000. Поле flag описывает различные характеристики ПМ в виде комбинации PR_xxx флажков, определенных в *idp.hpp*. Для ПМ Python, мы решили указать только PR_RNAMESOK, которая разрешает использовать имена регистров в качестве имен локаций (и это нормально, у нас нет регистров), и PRN_DEC, которая устанавливает по умолчанию формат чисел в десятичные. Остальные два поля, cnbits и dnbits, устанавливаем равным 8.

А Н А Л И З А Т О Р

На данный момент мы заполнили структуру LPH достаточно, чтобы начать думать о первой части ПМ - анализаторе. В существующих примерах ПМ, анализатор обычно выполнен в виде функции с именем **ana** (Вы можете задавать абсолютно любое имя), которая лежит в файле *ana.cpp*. Прототип этой функции очень простой:

```
int idaapi ana(void); //анализируем одну инструкцию и возвращаем ее длину
```

Вы должны поле **u_ana** объекта LPH установить равным указателю на Вашу функцию анализа. Работа анализатора - проанализировать одну инструкцию с заполнением глобальной переменной **cmd** информацией о ней, и вернуть длину инструкции. Анализатор не должен вносить изменения в базу данных.

Переменная **cmd** является глобальным экземпляром объекта **insn_t**. Класс **insn_t**, определенный в *ua.hpp*, используется для описания одной инструкции в базе данных. Ее объявление выглядит так:

```
class insn_t {
public:
        ea_t cs; // Базовый адрес сегмента в параграфах. Устанавливается ядром IDA
        ea_t ip; // Виртуальный адрес инструкции (внутри сегмента). Устанавливается ядром IDA
        ea_t ea; // Линейный адрес инструкции. Устанавливается ядром IDA
        ① uint16 itype; // значение из перечисления по инструкциям (не опкод!). Устан. функцией ana()
        ② uint16 size; // размер инструкции в байтах. Устан. функцией ana()
        union { // поле зависит от процессора. Устан. функцией ana()
                uint16 auxpref;
                struct {
                        uchar low;
                        uchar high;
                } auxpref_chars;
        };
        char segpref; // поле зависит от процессора. Устан. функцией ana()
        char insnpref; // поле зависит от процессора. Устан. функцией ana()
        ③ op_t Operands[6]; // информация об операндах инструкции. Устан. функцией ana()
        char flags; // флажки инструкции. Устан. функцией ana()
};
```

До вызова Вашей функции анализатора, ядро IDA заполняет первые три поля объекта **cmd** сегментым и линейным адресом анализируемой инструкции. Далее анализатор в процессе своей работы должен заполнить остальные поля, где обязательными полями являются - **itype** ①, **size** ② и **Operands** ③. Поле **itype** должно принимать одно из значений перечисления по инструкциям (*в нашем примере - это enum python_opcodes*), обсуждаемого выше. Поле **size** равно общему размеру инструкции (в байтах) и должно использоваться как возвратное значение из функции *ana()*. Если инструкцию невозможно разобрать (*распарсить*), то анализатор должен вернуть **size** равным нулю. И наконец, инструкция может иметь до 6 операндов, и анализатор должен заполнить информацию о каждом операнде, используемом в инструкции.

Функцию анализатора часто реализуют с помощью оператора switch. Первым шагом анализатор, как правило, запрашивает один или более (зависит от процессора) байт из потока команд и использует их как значение оператора switch. SDK предлагает ряд специальных функций для доступа к потоку команд

(инструкций):

```
//прочитать один байт текущей инструкции
uchar ua_next_byte(void);
//прочитать два байта текущей инструкции
ushort ua_next_word(void);
//прочитать 4 байта текущей инструкции
ulong ua_next_long(void);
//прочитать 8 байт текущей инструкции
ulonglong ua_next_qword(void);
```

Положение текущей инструкции первоначально имеет то же самое значение, что установлено в **cmd.ip**. Каждый вызов одной из **ua_next_xxx** функций имеет побочный эффект, выраженный в увеличении **cmd.size** согласно числу прочитанных байт с помощью вызова **ua_next_xxx** функции (1,2,4 или 8). Прочитанные байты должны быть декодированы: для этого достаточно назначить соответствующее значение перечисления типу инструкции (полю **itype**), определить число и тип любых операндов, требуемых для инструкции, и определить общую длину команды. Поскольку процесс декодирования может быть многоступенчатым, то дополнительные байты, требуемые для получения полноценной инструкции, считываются из потока команд. До тех пор, пока Вы используете **ua_next_xxx** функции, параметр **cmd.size** обновляется автоматически, исключая необходимость отслеживать какое количество байт мы запросили по ходу восстановления инструкции. Глядя с высокоуровневой перспективы, можно сказать, что анализатор отчасти подражает фазам работы реального процессора: выборке команд и декодированию инструкций.

Как и в реальной жизни, процесс декодирования инструкций, как правило, легче для процессоров с фиксированными размерами инструкции, как это часто бывает с RISC-архитектурой, и, как правило, тяжелее для процессоров, которые используют переменную длину инструкции, таких как x86.

Используя прочитанные байты, анализатор должен установить один элемент в массиве **cmd.Operands** для каждого операнда в выбранной инструкции. Операнды инструкций представлены с использованием экземпляров класса **op_t**, который определен в *ua.hpp* и кратко показан здесь:

```
class op_t {
public:
        char n; // номер операнда (0,1,2). Ядро устан. это поле. Не изменять!
        optype_t type; // тип операнда. Устан. в ana, смотри ua.hpp для значений

        // смещение операнда относительно начала инструкции
        char offb; //Устан. в функции в ana, устан. в 0 если не известно

        // смещение 2-й части операнда (если есть) относительно начала инструкции
        char offo; //Устан. в функции в ana, устан. в 0 если не известно
        uchar flags; //Устан. в функции в ana. Смотри ua.hpp для возможных значений

        char dtyp; // Задает тип данных операнда. Смотри в ana. Смотри ua.hpp для значений

        // Следующие объединения unions содержат дополнительную полезную информацию об операнде
        union {                 //для операнда с типом o_reg
                uint16 reg;     //номер регистра для типа o_reg
```

```
        uint16 phrase;      //номер регистровой фразы для типов o_phrase и o_displ
                            //определите номер фразы на Ваше усмотрение
    };
    union {                 //для операнда с типом o_imm (непосредственное значение) или
        uval_t value;       //смещения (o_displ+OF_OUTER_DISP)
        struct {            //Для удобства доступа к старшей и младшей частям значения
            uint16 low;
            uint16 high;
        } value_shorts;
    };
    union {                 //виртуальный адрес (смещение внутри сегмента)
        ea_t addr;          //типы операнда (o_mem,o_displ,o_far,o_near)
        struct {            //Для удобства доступа к старшей и младшей частям адреса
            uint16 low;
            uint16 high;
        } addr_shorts;
    };
    //Поля зависят от процессора, используйте их на свое усмотрение. Устан. в ana
    union {
        ea_t specval;
        struct {
            uint16 low;
            uint16 high;
        } specval_shorts;
    };
    char specflag1, specflag2, specflag3, specflag4;
};
```

Настройка операнда начинается с определения типа операнда (поле **type**), которое равно одному из значений константы **optype_t**, определенной в *ua.hpp*. Поле **type** описывает либо операнд-источник, либо операнд-приемник. Другими словами, поле **type** грубо описывает режим адресации, используемый для доступа к операнду. Например, тип **o_reg** означает, что операнд является регистром; **o_mem** означает, что операнд - это адрес памяти известный на стадии компиляции, а **o_imm** означает, что операнд - это непосредственные данные, содержащиеся в инструкции.

Поле **dtype** определяет размер данных операнда. Это поле должно содержать одно из значений dt_*xxx,* описанных в *ua.hpp*. Например, dt_byte для 8-миразрядных данных, dt_word для 16-разрядных и dt_dword для 32-разрядных.

На примере x86 инструкций показаны основные типы данных с наиболее часто встречающимися операндами:

```
mov eax, 0x31337                ; o_reg(dt_dword), o_imm(dt_dword)
push word ptr [ebp - 12]    ; o_displ(dt_word)
mov [0x08049130], bl        ; o_mem(dt_byte), o_reg(dt_byte)
movzx eax, ax               ; o_reg(dt_dword), o_reg(dt_word)
ret                         ; o_void(dt_void)
```

Способ, которым определяется какие unions в **op_t** мы используем, диктуется значением поля **type**. Например, когда операнд имеет тип **o_imm**, то непосредственные данные должны храниться в поле **value**, и в случае, когда тип **o_reg**, то номер регистра (значение из перечисления, в котором хранятся константы регистров) в поле **reg** должно быть заполнено. Более полную информацию о том, где хранить каждую часть инструкции содержатся в *ua.hpp*.

Обратите внимание, что ни одно из полей в op_t не описывает в каком качестве выступает операнд - то ли он источник данных, то ли он приемник. На самом деле, это не работа анализатора определять такие вещи. Флажки *canonical feature* (напомню, флажки CF_xxx), указанные в массиве инструкций используются в более поздней стадии процессора для того, чтобы выяснить, как именно применять операнд.

Некоторые из полей как в insn_t классе, так и в op_t классе описываются как *процессоро-зависимые*, т.е. Вы можете использовать эти поля для любых целей на Ваше усмотрение. Такие поля часто применяют для хранения информации, которая не вписывается ни в одно из других полей этих классов. К тому же процессоро-зависимые поля - удобный механизм для передачи информации между различными этапами ПМ и поэтому на последующих этапах нет надобности копировать работу анализатора.

Итак, все основные правила для работы анализатора рассмотрены, теперь мы можем разработать минимальный анализатор для байт-кода Python. Байт-код Python очень прост. Длина опкодов Python - 1 байт. Инструкции, у которых значение опкодов менее 90, не имеют операндов, а значения от 90 (включительно) и выше имеют 2-байтовый операнд. Наш базовый вариант анализатора показан здесь:

```
#define HAVE_ARGUMENT 90
int idaapi py_ana(void) {
        cmd.itype = ua_next_byte(); //для нас опкоды РАВНЫ itypes (обновляется cmd.size)
        if (cmd.itype >= PYTHON_LAST) return 0; //неправильная инструкция
        if (Instructions[cmd.itype].name == NULL) return 0; //неправильная инструкция
        if (cmd.itype < HAVE_ARGUMENT) { //нет операндов
                cmd.Op1.type = o_void; //Op1 - это макрос для Operand[0] (смотри ua.hpp)
                cmd.Op1.dtyp = dt_void;
        }
        else { //инструкция должна иметь 2-хбайтовый операнд
                if (flags[cmd.itype] & (HAS_JREL | HAS_JABS)) {
                        cmd.Op1.type = o_near; //операнд  - ссылка на код
                }
                else {
                        cmd.Op1.type = o_mem; //операнд - ссылка на память (sort of)
                }
                cmd.Op1.offb = 1; //смещение на операнд равно 1 байту внутри инструкции
                cmd.Op1.dtyp = dt_dword; //В python нет размеров, так что мы просто выбираем что-нибудь
                cmd.Op1.value = ua_next_word(); //выборка 2-х байт значения операнда (обновляется
                cmd.size)
                cmd.auxpref = flags[cmd.itype]; //сохраним флажки для последующих стадий ПМ
                if (flags[cmd.itype] & HAS_JREL) {
                        //вычислим адрес относительного перехода (jump)
                        cmd.Op1.addr = cmd.ea + cmd.size + cmd.Op1.value;
                }
                else if (flags[cmd.itype] & HAS_JABS) {
                        cmd.Op1.addr = cmd.Op1.value; //сохраним абсолютный адрес
```

```
            }
            else if (flags[cmd.itype] & HAS_CALL) {
                //вызов функции, операнд указывает сколько аргументов в стеке
                //сохраним эти значения для последующих этапов ПМ
                cmd.Op1.specflag1 = cmd.Op1.value & 0xFF; //позиционный параметр
                cmd.Op1.specflag2 = (cmd.Op1.value >> 8) & 0xFF; //ключевой параметр
            }
        }
        return cmd.size;
}
```

Для процессорного модуля Python, мы решили создать дополнительный массив флажков **flags**, по одному на команду для дополнения (а в некоторых случаях для репликации) флажков *canonical feature* каждой инструкции. Мы их определили как флажки HAS_JREL, HAS_JABS и HAS_CALL. Они говорят о том, что операнд инструкции является либо относительным адресом перехода, либо абсолютным адресом, либо адресом функции, соответственно. Объяснить каждую деталь фазы анализатора трудно не вдаваясь в работу интерпретатора Python, поэтому мы обобщаем информацию о нашем анализаторе чуть ниже, а также через комментарии в предыдущем коде, не забывая, что работа анализатора заключается в том, чтобы анализировать по одной команде:

1. Анализатор получает следующий байт инструкции из потока команд и принимает решение о правильности опкода Python.
2. Если инструкция не содержит операндов, то **cmd.Operand[0]** (cmd.Op1) устанавливается в **o_void**.
3. Если у команды есть операнд, то **cmd.Operand[0]** принимает значение согласно типа операнда. Несколько процессоро-зависимых полей используются для передачи информации далее к последующим этапам ПМ, т.е. устанавливаем эти поля здесь в анализаторе, а разбираемся с ними на следующих стадиях ПМ (например, в эмуляторе).

Более сложные наборы команд почти наверняка потребуют более сложный анализ. В целом, однако, поведение любого анализатора можно обобщить следующим образом:

1. Читаем необходимое количество байт из потока команд, определяем есть ли такая инструкция, получаем значение из перечисление, в котором хранятся именованные константы инструкций, затем найденное значение сохраняем в **cmd.itype**. Эта операция часто реализуется как большой оператор ветвления switch, который разбирает каждое значение опкода.
2. Читаем необходимое количество дополнительных байт для того, чтобы определить число операндов инструкции, режимы адресации, используемые этими операндами, а также узнать отдельные компоненты каждого операнда (регистры и непосредственные данные). В результате заполняем элементы массива **cmd.Operands**. Эта операция может быть разложена на отдельные подфункции анализа операндов.
3. Возвращаем общую длину инструкции и ее операндов.

Строго говоря, после того как мы полностью разложили инструкцию на составляющие, IDA достаточно информации для создании представления инструкции на языке ассемблера. Чтобы создать перекрестные ссылки, облегчить процесс разбора рекурсии и отследить как изменяется стековый указатель программы IDA должна получить дополнительные сведения о поведении каждой инструкции. Этой работой

занимается эмулятор ПМ IDA.

Э М У Л Я Т О Р

В то время как анализатор изучает структуру одиночной инструкции, этап эмулятора сконцентрирован на поведении одиночной инструкции. В примерах ПМ IDA, как правило, эмулятор реализован функцией с именем **emu** (вы вправе назвать ее как угодно) и находится она в файле *emu.cpp*. Также как и функция **ana**, прототип этой функции очень простой:

```
int idaapi emu(void); //эмуляция одной инструкции
```

Согласно *idp.hpp*, функция **emu** должна возвращать длину инструкции, которая была эмулирована; однако большинство примеров эмуляторов, кажется, возвращает значение равное 1.

Вы должны поле **u_emu** объекта LPH установить равным указателю на Вашу функцию эмулятора. К тому моменту, когда вызывается **emu()** объект **cmd** уже заполнен анализатором. Основная цель эмулятора - создание перекрестных ссылок на основе поведения инструкции, описанной в **cmd**. Также в эмуляторе отслеживаются любые изменения указателя стека и здесь же создаются локальные переменные, исходя из наблюдений за функциями доступа к стековому фрейму. В отличие от анализатора, эмулятор может изменить базу данных IDA.

Определение того факта, что надо создать перекрестную ссылку, как правило, делается на основании изучения флажков *canonical features* совместно с полем **type** операндов инструкции. Очень простой вариант эмулятора для набора команд, где инструкция может иметь до двух операндов. И такой эмулятор, который можно встретить во многих примерах из SDK, показан здесь:

```
void TouchArg(op_t &op, int isRead); //Реализуется автором ПМ

int idaapi emu() {
        ulong feature = cmd.get_canon_feature(); //получить флажки команды CF_xxx

        if (feature & CF_USE1) TouchArg(cmd.Op1, 1);
        if (feature & CF_USE2) TouchArg(cmd.Op2, 1);

        if (feature & CF_CHG1) TouchArg(cmd.Op1, 0);
        if (feature & CF_CHG2) TouchArg(cmd.Op2, 0);
        if ((feature & CF_STOP) == 0) {
                //нормальное исполнение программы
                //добавить “ссылку на следующую команду”(ordinary flow)
                ua_add_cref(0, cmd.ea + cmd.size, fl_F);
        }
        return 1;
}
```

Кстати, про ссылку ordinary flow хорошо разъяснено в книге Криса Касперски “Образ мышления - дизассемблер IDA” глава “Перекрестные ссылки” - от перев.

Для каждого операнда инструкции, предыдущая функция проверяет флажки CF_xxx и определяет какого вида перекрестная ссылка должна быть сформирована. В приведенном примере, функция по имени

TouchArg проверяет один операнд, чтобы принять решение какой тип назначить перекрестной ссылке и обрабатывает подробности для создания корректной ссылки. Когда Ваш эмулятор создает перекрестные ссылки,то Вы должны использовать для этого специальные функции описанные в *ua.hpp*, а не в *xref.hpp*. Следующие несколько строк приблизительно рассказывают как определяется тип перекрестной ссылки.

- Если тип операнда **o_imm**, операция - чтение (isRead = true) и значение операнда - указатель, то создать смещение на ссылку (offset reference). Определить, является ли операнд указателем, вызывая функцию isOff (*объявлена в bytes.hpp, isOff - is offset? - это смещение?*), например так, isOff(uFlag, op.n). Добавить смещение на перекрестную ссылку с помощью **ua_add_off_drefs**, например так, **ua_add_off_drefs**(op, dr_O);
- Если тип операнда **o_displ** и значение операнда - указатель, то создать смещение на перекрестную ссылку с типом взависимости от операции - запись или чтение, например, **ua_add_off_drefs**(op, isRead ? dr_R : dr_W);
- Если тип операнда **o_mem**, добавить перекрестную ссылку на адрес данных с типом взависимости от операции - запись или чтение с помощью **ua_add_dref**, например, **ua_add_dref**(op.offb, op.addr, isRead ? dr_R : dr_W);
- Если тип операнда **o_near**, добавить перекрестную ссылку на код с типом взависимости от операции - jump (условный или безусловный переход) или call (вызов функции) с помощью **ua_add_cref**, например, ua_add_cref(op.offb, op.addr, feature & CF_CALL ? fl_CN : fl_JN);

Эмулятор отвечает также за то, как ведет себя регистр указателя стека. Эмулятор должен вызывать фукнцию **add_auto_stkpnt2** для информирования IDA о том, что значение указателя стека изменилось. Прототип add_auto_stkpnt2 выглядит так:

```
bool add_auto_stkpnt2(func_t *pfn, ea_t ea, sval_t delta);
```

Указатель **pfn** указывает на функцию, которая содержит адрес, эмулированный в данный момент. Если **pfn** равен NULL, то он автоматически определяется IDA. Параметр **ea** должен равняться концу адреса инструкции (как правило, **cmd.ea + cmd.size**), которая изменяет указатель стека. Параметр **delta** используется для указания количества байт, на которое он увеличивается или уменьшается. Отрицательное значение означает, что стек растет (например, после инструкции **push**), а положительное - стек уменьшается (например, после инструкции **pop**). Простая 4-байтовая корректировка указателя стека операцией **push** можно эмулировать следующим образом:

```
if (cmd.itype == X86_push) {
        add_auto_stkpnt2(NULL, cmd.ea + cmd.size, -4);
}
```

В целях сохранения точного отчета о том, как ведет себя указатель стека, эмулятор должен распознавать и эмулировать все инструкции, касающиеся указателя, а не только простые **push** и **pop** случаи. Более сложный вариант возникает тогда, когда функция выделяет локальные переменные методом вычитания некой константы из указателя стека. Этот случай проиллюстрирован ниже:

```
//обрабатываем случай вида: sub esp, 48h
```

```
if (cmd.itype == X86_sub && cmd.Op1.type == o_reg && cmd.Op1.reg == r_esp && cmd.Op2.type == o_imm) {
                add_auto_stkpnt2(NULL, cmd.ea + cmd.size, -cmd.Op2.value);
}
```

Так как архитектуры процессоров значительно отличаются друг от друга, то не возможно для IDA (или любой другой программе, если на то пошло) рассчитать все возможные способы, какими операнд может быть сформирован или все пути, какими инструкция может ссылаться на другие инструкции или данные . В результате, нет точных рецептов изготовления своего модуля-эмулятора. Просматривая исходники существующих ПМ, понимаешь, что потребуется преодолеть множество проб и ошибок прежде, чем эмулятор будет делать все, что вы от него хотите.

Эмулятор для нашего ПМ Python показан ниже:

```
int idaapi py_emu(void) {
        //Мы только определяем адреса назначения для относительных переходов (relative jumps)
        if (cmd.auxpref & HAS_JREL) { //проверяем флажки, установленные в анализаторе
                ua_add_cref(cmd.Op1.offb, cmd.Op1.addr, fl_JN);
                }
        //Добавить “ссылку на следующую команду” (ordinary flow) до тех пор, пока не встретим команду с
        //установленным флажком CF_STOP
        if((cmd.get_canon_feature() & CF_STOP) == 0) {
                //cmd.ea + cmd.size вычисляем адрес следующей инструкции
                ua_add_cref(0, cmd.ea + cmd.size, fl_F);
        }
        return 1;
}
```

Опять же, из-за архитектуры интерпретатора Python, мы сильно ограничены в типах создаваемых перекрестных ссылок. В байт-коде Python нет концепции адрес памяти для элементов данных, а абсолютный адрес каждой команды может быть определен только путем разбора метаданных, содержащихся в скомпилированном Python (*.pyc*) файле. Сами данные, либо хранятся в таблицах и ссылка к ним идет по индексу в этих таблицах, либо они хранятся на стеке программы и они не имеют прямых ссылок. И снова, в то время как мы можем напрямую считывать элемент данных по индексу из операнда инструкции, мы не можем знать структуру таблиц, где лежат эти данные, если мы заранее не проанализировали дополнительные метаданные, содержащиеся в *.pyc* файле. В нашем процессоре, мы вычисляем только относительный переход на целевую инструкцию и адрес следующей команды, потому что он легко вычисляется относительно текущего адреса инструкции. Тот факт, что наш ПМ обеспечивает лучшее дизассемблирование, только в том случае, если он имеет более детальное понимание структуры файла - это ограничение, которое обсудим далее в разделе "**Архитектура процессорного модуля**".

По схожим причинам, мы точно не отслеживаем поведение стекового указателя в нашем ПМ Python. Прежде всего потому, что IDA обрабатывает изменения указателя стека только тогда, когда изменения идут в пределах функции, а у нас в настоящее время нет способов обнаруживать границы функции внутри Python кода. Если бы мы реализовывали отслеживание стекового указателя, то было бы мудро не забывать, что, благодаря стековой архитектуре, практически каждая инструкция Python изменяет стек в некотором роде. В этом случае можно упростить процесс определения того, на сколько байт указатель стека изменяется

каждой командой - для этого проще определить массив значений, по одному на каждую инструкцию Python, который содержит значение равное количеству байт, на которое изменяется стек. Затем эти значения будем использовать при обращении к функции **add_auto_stkpnt2** каждый раз при эмуляции инструкции.

Как только эмулятор добавил все перекрестные ссылки, какие смог и сделал все изменения в базу данных, которые он посчитал необходимыми, Вы готовы вывести результат на экран. В следующем разделе мы обсудим роль генератора печати (*outputter)* для формирования IDA дизассемблированного листинга на экран.

O U T P U T T E R (ГЕНЕРАТОР ПЕЧАТИ)

Outputter - выводит одну дизассемблированную инструкцию, на основании данных указанных в глобальной переменной **cmd**, на экран IDA. В ПМ IDA, генератор печати, как правило, реализован функцией с именем **out** (вы можете назвать ее как угодно) в файле с именем *out.cpp*. Как и **ana**, и **emu** функции, прототип **out** функции очень прост:

```
void idaapi out(void); //выводим одну дизассемблированную инструкцию
```

Вы должны поле **u_out** объекта LPH установить равным указателю на Вашу функцию вывода. К моменту, когда происходит вызов **out** функции, **cmd** уже проинициализирован анализатором. Ваша функция вывода не должна вносить изменения в базу данных. Вы также должны создать вспомогательную функцию, единственной целью которой является форматирование и вывод на экран одного операнда инструкции. Как правило, эта функция называется **outop** и на нее указывает элемент **u_outop** объекта LPH. Ваша **out** функция не должна вызывать **outop** напрямую. Вместо этого, Вам следует обращаться к **out_one_operand** каждый раз, когда надо распечатать операнд как часть Ваших дизассемблированных строк. Операции вывода данных обрабатываются отдельной функцией, обычно называемой ***cpu*_data** и на нее указывает элемент **d_out** объекта LPH. В нашем ПМ Python эта функцией называется **python_data**.

Каждая строка в листинге дизассемблирования состоит из нескольких компонент, таких как префикс, имя метки, мнемоники, операндов и, возможно, комментария. Ядро IDA несет ответственность за отображение некоторых компонент (такие как, префиксы, комментарии и перекрестные ссылки), а за вывод остальных отвечает ПМ. Несколько полезных функций при создании различных частей строки-результата описаны в *ua.hpp* под следующим подзаголовком (“IDP вспомогательные функции - Вывод результата”):

```
//-------------------------------------------------------------------------------------------------------
//      I D P    H E L P E R    F U N C T I O N S    -    O U T P U T
//-------------------------------------------------------------------------------------------------------
```

Цветовое раскрашивание частей результирующей строки возможно благодаря функциям, которые вставляют специальные тэги цвета в Ваш буфер вывода. Дополнительные функции для создания результирующих строк можно найти в *lines.hpp*.

Вместо того, чтобы использовать модель вывода результата в стиле консоли, где Вы пишете содержимое строки прямо на экран IDA, IDA предлагает схему вывода на основе буфера, где строку пишем сначала в специальный символьный буфер, а затем просим IDA отобразить буфер. Основные этапы для создания результирующей строки выглядит следующим образом:

1. Вызов **init_output_buffer**(char *buf, size_t bufsize) (описанной в *ua.hpp*) - инициализируем наш буфер вывода.
2. Используем функции из *ua.hpp* для создания одной строки вывода, где содержимое строки укладываем в наш буфер. Большинство этих функций автоматически пишут в буфер назначения, созданный на предыдущем шаге, так что нет необходимости каждый раз явно передавать буфер в эти функции. Имена этих функций, как правило, выглядят как **out_xxx** или **OutXxx**.
3. Вызываем **term_output_buffer**() - фиксируем буфер вывода, теперь он готов к печати на экран IDA.
4. Посылаем буфер вывода ядру IDA, используя функции либо **MakeLine** или **printf_line** (обе объявлены в *lines.hpp*).

Обратите внимание, что вызов **init_output_buffer**, **term_output_buffer** и **MakeLine** обычно происходит из Вашей **out** функции. Ваша **outop** функция обычно использует текущий буфер вывода, созданный в **out** и, как правило, нет необходимости в ней инициализировать свои собственные буферы вывода.

Собственно говоря, Вы можете пропустить вообще все манипуляции с буфером, описанные выше в первых четырех шагах, и перейти прямо к вызову **MakeLine**, если не хотите напрягать мозг, чтобы следить за буфером, и отказываясь от вспомогательных функций, предлагаемых в *ua.hpp*. Кроме того, что буфер назначения доступен в функциях по-умолчанию (тот самый, который создаем в **init_out_buffer**), многие из них могут автоматически работать с текущим содержимым переменной **cmd**. Некоторые из наиболее полезных вспомогательных функций из *ua.hpp* описаны здесь:

OutMnem(int width, char *suffix)

Вывод мнемоники команды, которая соответствует **cmd.itype**, в поле шириной не менее **width** символов, с добавлением указанного суффикса. По крайней мере, после мнемоники печатаем один пробел. Ширина по умолчанию составляет 8 символов, а суффикс равен NULL. Суффиксом может быть, например, модификатор размера операнда, как показано в следующих x86 командах: **movsb, movsw, movsd**.

out_one_operand(int n)

Вызывает Вашу функцию **outop** для печати n-го операнда **cmd.Operands[n]**

out_snprintf(const char *format, ...)

Добавляет форматированный текст к текущему буферу вывода

OutValue(op_t &op, int outflags)

Вывод неизменяемых полей операнда. Эта функция выводит **op.value** или **op.addr**, в зависимости от значения **outflags**. Смотрите *ua.hpp* для понимания значений флажков поля **outflags**, которое по умолчанию равно 0. Эта функция предназначена для вызова из **outop**.

out_symbol(char c)

Выводим указанный символ, используя текущий цвет для пунктуации (COLOR_SYMBOL, определенный в *lines.hpp*). Эта функция используется в основном для вывода синтаксических элементов внутри операндов (и, значит, она вызывается из **outop**), например, как запятые и скобки.

out_line(char *str, color_t color)

Добавляет указанную строку, в указанном цвете, в текущий буфера вывода. Цвета определены в *lines.hpp*. Обратите внимание, что эта функция не выводит никаких линий (т.е. не рисует ничего на экран). Лучшее бы имя для этой функции могло быть **out_str**.

OutLine(char *str)
То же самое, что и **out_line**, только не используем цвет

out_register(char *str)
Выводим заданную строку, используя текущий цвет для регистров (COLOR_REG).

out_tagon(color_t tag)
Вставить тег включения заданного цвета в буфер вывода. Последующий вывод в буфер будет отображаться указанным цветом до тех пор, пока не встретим тег выключения цвета.

out_tagoff(color_t tag)
Вставить тег выключения цвета в буфер вывода.

Смотрите в *ua.hpp* дополнительные функции вывода, которые будут полезны в построении вашего outputter.

Кстати, есть возможность, отсутствующая в *ua.hpp,* легко выводить имя регистра. Во время фазы анализа, номера регистров записываются в поля операндов **reg** или **phrase** в зависимости от режима адресации, используемого для этого операнда. Поскольку многие операнды используют регистры, было бы неплохо иметь функцию, которая быстро выводит имя регистра на основании номера регистра. Следующая функция предоставляет минимальные возможности для этого:

```
//способ применения: OutReg(op.reg);
void OutReg(int regnum) {
        out_register(ph.regNames[regnum]); //используем regnum как индекс в массиве имен регистров
}
```

IDA вызывает вашу функцию **out** только по мере необходимости, когда адрес попадает в область просмотра окна “IDA View” (*основное окно, в котором отображается результат дизассемблирования и с которым работает пользователь - от перев.*) или когда часть строки изменяем. Каждый раз, когда вызывается **out**, то ожидается вывод такого количества строк, сколько необходимо для представления одной инструкции, описанной в глобальной переменной **cmd**. Для этого **out**, как правило, вызывает одну или несколько функций **MakeLine** (или **printf_line**). В большинстве случаев одной строки (и, следовательно, одного вызова **MakeLine**) будет достаточно. В случае, если на одну инструкцию надо вывести несколько строк, то Вы никогда не должны добавлять перенос строки в ваш буфер вывода в попытке создать несколько строк сразу. Вместо этого, Вы должны сделать несколько вызовов **MakeLine** для вывода каждой отдельной строки. Прототип функции **MakeLine** показан ниже:

```
bool MakeLine(const char *contents, int indent = -1);
```

Значение отступа **indent** равное -1 означает, что берется значение по умолчанию, которое соответствует текущему значению **inf.indent** (поле ввода *Instructions idention*), указанному на вкладке “Disassembly” **Options -> General** диалога. Параметр **ident** имеет дополнительный смысл, когда инструкция (или данные) охватывает несколько строк в дизассемблированном виде. При многострочной инструкции, отступ -1 выделяет ту строку, которая наиболее важная для этой команды. Пожалуйста, ознакомьтесь с комментариями к **printf_line** функции в *lines.hpp* для получения дополнительной информации об использовании отступов в подобной манере.

До этого момента, мы избегали обсуждения комментариев. Также как и имена, и перекрестные ссылки, комментарии обрабатываются ядром IDA. Тем не менее, мы можем позволить себе выбрать строку в многострочной инструкции, где будет отображаться комментарий. Вывод комментариев контролируется в некоторой степени глобальной переменной под именем **gl_comm**, объявленной в *lines.hpp*. Самое главное, что надо знать о **gl_comm**: комментарии не будут отображаться вообще, если **gl_comm** не равен 1. Если **gl_comm**=0, то комментарии не будут отображаться в конце Вашего вывода, даже если пользователь ввел его вручную и флажок “Comments” в диалоге **Options -> General** включен. Беда в том, что **gl_comm** по умолчанию равен 0, так что Вы должны убедиться, что Вы установите его в 1 в какой-то момент, если хотите, чтобы пользователи видели комментарии в Вашем процессорном модуле. Когда **out** функция печатает многострочную инструкцию, Вам нужно контролировать **gl_comm**, если Вы хотите, чтобы комментарии пользователей отображались и на других строках, кроме первой строки вывода.

Итак, функция **out** для нашего ПМ Python:

```
void py_out(void) {
        char str[MAXSTR]; //MAXSTR в IDA определена в pro.h (это 1024 символа)
        init_output_buffer(str, sizeof(str));
        OutMnem(12); //сначала мы выводим мнемонику команды
        if(cmd.Op1.type != o_void) { //если есть операнд, печатаем и его
                out_one_operand(0);
        }
        term_output_buffer();
        gl_comm = 1; //мы хотим комментариев!
        MakeLine(str); //вывод строки со значением отступа по умолчанию
}
```

Сама функция прокладывает свой путь через компоненты дизассемблированной строки весьма простым способом. Если инструкции Python имеют два операнда, мы могли бы обратиться к **out_symbol** для вывода запятой, а затем вызвать второй раз **out_one_operand** для второго операнда. В большинстве случаев, Ваша **outop** функция в некоторой степени будет более сложной, чем Ваша **out** функция, т.к. структура операндов, как правило, более сложна, чем структура высокоуровневой инструкции. Обычно реализация **outop** функции выглядит как один оператор ветвления switch, который проверяет значение типа операнда **type** и его формат соответственно.

В нашем примере ПМ Python, например, мы вынуждены использовать очень простую **outop** функцию, поскольку в большинстве случаев нам не хватает информации, необходимой для перевода целочисленного значения операндов во что-нибудь более понятное. Нашу реализацию этой функции смотрите ниже, в ней сделано лишь специальная обработка сравнений и относительных переходов:

```
//Символьное представление возможных сравнений
```

```
char *compare_ops[] = {
        "<", "<=", "==", "!=", ">", ">=",
        "in", "not      in", "is", "is not", "exception match"
};
bool idaapi py_outop(op_t& x) {
        if (cmd.itype == COMPARE_OP) {
                //Анализ сравнений: аргумент задает тип выполняемого сравнения. Выводим символьное //
                представление сравнения, а не число
                if (x.value < qnumber(compare_ops)) {
                        OutLine(compare_ops[x.value]);
                }
                else {
                        OutLine("BAD OPERAND");
                }
        }
        else if (cmd.auxpref & HAS_JREL) {
                //здесь мы не проверяем x.type == o_near, потому что мы должны различать относительный
                и //абсолютный переход. В нашем случае, HAS_JREL подразумевает o_near
                out_name_expr(x, x.addr, x.addr);
        }
        else { //в любом другом случае выводим только значение операнда
                OutValue(x);
        }
        return true;
}
```

В дополнение к дизассемблированным инструкциям, листинг обычно содержит байты, которые должны быть представлены в виде данных. На этапе вывода, отображением данных занимается элемент **d_out** объекта **LPH**. Ядро вызывает функцию **d_out** для отображения любых байт, которые не являются частью инструкций, будь то тип данных этих байт неизвестен или байты были отформатированы как данные самим пользователем или эмулятором. Прототип **d_out** показан здесь:

```
void idaapi d_out(ea_t ea); //форматируем данные по указанному адресу
```

Функция **d_out** должна изучить флаги, связанные с адресом, указанным в параметре **ea** и создать соответствующие представления данных в стиле генерируемого языка ассемблера. Эта функция должна быть указана для всех процессорных модулей. Минимальная реализация, предлагаемая SDK в виде функции **intel_data**, вряд ли будет соответствовать Вашим конкретным потребностям.

В нашем примере ПМ Python ситуация такова, что мы на самом деле имеем очень мало информации для форматирования статических данных, поскольку у нас нет средств, чтобы найти их. Ради примера мы используем такую функцию:

```
void idaapi python_data(ea_t ea) {
        char obuf[256];
```

```
        init_output_buffer(obuf, sizeof(obuf));
        flags_t flags = get_flags_novalue(ea); //получить флажки для указанного адреса ea
        if (isWord(flags)) { //вывод данных, объявленных как тип word (слово)
                out_snprintf("%s %xh", ash.a_word ? ash.a_word : "", get_word(ea));
        }
        else if (isDwrd(flags)) { //вывод данных, объявленных как тип dword (двойное слово)
                out_snprintf("%s %xh", ash.a_dword ? ash.a_dword : "", get_long(ea));
        }
        else { //по умолчанию, мы во всех других случаях выводим данные с типом byte (байт)
                int val = get_byte(ea);
                char ch = ' ';
                if (val >= 0x20 && val <= 0x7E) {
                        ch = val;
                }
                out_snprintf("%s %02xh ; %c", ash.a_byte ? ash.a_byte : "", val, ch);
        }
        term_output_buffer();
        gl_comm = 1;
        MakeLine(obuf);
}
```

Функции для доступа и проверки флажков, связанные с любым адресом в базе данных IDA найдете в *bytes.hpp*. В приведенном примере, тестируем флажки на наличие по указанному адресу данных с типом word (слово) или dword (двойное слово), и выводим данные на экран с соответствующим ключевым словом объявления типа из текущего модуля ассемблера. Глобальная переменная **ash** - это экземпляр **asm_t** структуры, которая описывает характеристики синтаксиса ассемблера, используемого в данный момент при дизассемблировании. В целях отображения более сложных данных, таких как массивы, логика в приведенном примере должна быть более многозначительней, чем сейчас.

УВЕДОМЛЕНИЯ ПРОЦЕССОРА

В главе 17 мы обсуждали возможность перехвата плагинами различных сообщений-уведомлений с помощью функции **hook_to_notification_point**. Перехватив уведомления, плагины таким образом информируются о различных мероприятиях, происходящих в базе данных. Концепция уведомлений существует и для процессорных модулей, но уведомления процессора осуществляются несколько иным образом, чем у плагинов.

Все ПМ должны установить указатель на функцию уведомления - это поле **notify** объекта **LPH**. Прототип **notify** функции показан здесь:

```
int idaapi notify(idp_notify msgid, ...); //уведомление процессора сообщением msgid
```

Функция **notify** - это функция с переменным количеством параметров, которая получает на входе код уведомления и изменяемый список аргументов специально для этого кода. Полный список доступных

кодов уведомлений процессора можно найти в *idp.hpp*. Уведомления существуют как для простых действий, таких как загрузка (**init**) и выгрузка (**term**) ПМ, так и для более сложных ситуаций, когда код или данные создаются, добавление или удаление функций, добавление или удаление сегментов. Список параметров, предоставляемых с каждым кодом уведомления также указан в *idp.hpp*. Прежде чем взглянуть на пример функции уведомления, стоит отметить следующие замечания, встречаемые только в некоторых SDK примерах ПМ:

```
//По хорошему, ПМ должен вызывать invoke_callbacks() функцию в своей notify() функции.
//Если invoke_callbacks() возвращает код 0, то ПМ должен обработать уведомление сам.
//В противном случае код должен быть возвращен вызывающему процессу.
```

Для того, чтобы гарантировать, что все модули, которые перехватывают уведомления процессора, оповещены надлежащим образом, должна быть вызвана **invoke_callbacks** функция. Она обращается к ядру для распространения данного уведомления для всех зарегистрированных функций обратного вызова. Функция **notify**, используемая в нашем ПМ Python показана ниже:

```
static int idaapi notify(processor_t::idp_notify msgid, ...) {
        va_list va;
        va_start(va, msgid); //установить список аргументов
        int result = invoke_callbacks(HT_IDP, msgid, va);
        if (result == 0) {
                result = 1; //по умолчанию - успешно обработано уведомление
                switch(msgid) {
                        case processor_t::init:
                                inf.mf = 0; //обеспечить прямой порядок байтов!
                                break;
                        case processor_t::make_data: {
                                ea_t ea = va_arg(va, ea_t);
                                flags_t flags = va_arg(va, flags_t);
                                tid_t tid = va_arg(va, tid_t);
                                asize_t len = va_arg(va, asize_t);
                                //наша функция d_out может обрабатывать только типы byte, word, dword
                                if (len > 4) {
                                        result = 0; //запретить обработку данных большей длины
                                }
                                break;
                        }
                }
        }
        va_end(va);
        return result;
}
```

Наша **notify** функция обрабатывает только два кода уведомления: **init** и **make_data**. Код **init** необходим для того, чтобы заставить ядро трактовать данные как “остороконечные” (*дословный перевод little-endian: т.е. порядок байт данных идет от младшего байта к старшему*). Флаг inf.mf (*mf аббревиатура most first - самый первый*) указывает ядру IDA какой порядок байт назначить (0 - от младшего к старшему, 1 - от старшего к младшему). Код **make_data** посылается всякий раз, как происходит попытка конвертации байт в данные. В нашем случае, функция d_out способна понять байт, слово, двойное слово, поэтому она проверяет размер данных и запрещает создавать элементы более, чем 4 байта.

Другие поля структуры processor_t

Для того, чтобы закончить обсуждение темы создания процессорных модулей, мы, по крайней мере, должны затронуть несколько дополнительных полей в объекте **LPH**. Как упоминалось ранее, в этой структуре есть огромное количество указателей на функции. Если посмотреть на определение структуры **processor_t** в *idp.hpp*, станет ясно, что в некоторых случаях можно спокойно установить некоторые указатели на функции в NULL, и тогда ядро IDA не будет вызывать их. Для всех других функций, требуемых **processor_t**, кажется разумно предположить, что Вы обязаны предоставить их реализации. Но здесь работает общее правило: когда Вы находитесь в растерянности относительно того, что должны делать эти функции, то часто можно обойти проблему, реализовав пустые функции-заглушки. В нашем ПМ Python, где было не ясно, что NULL есть допустимое значение, мы установили указатели на функции следующим образом (за информацией о том, что делает каждая конкретная функция смотри в *idp.hpp*):

header = Указателю на функцию-заглушку в нашем примере.
footer = Указателю на функцию-заглушку в нашем примере.
segstart = Указателю на функцию-заглушку в нашем примере.
segend = Указателю на функцию-заглушку в нашем примере.
is_far_jump = NULL в нашем примере.
translate = NULL в нашем примере.
realcvt = Указателю на **ieee_realcvt** из *ieee.h*.
is_switch = NULL в нашем примере.
extract_address = Указателю на функцию, которая возвращает (BADADDR–1) в нашем примере.
is_sp_based = NULL в нашем примере.
create_func_frame = NULL в нашем примере.
get_frame_retsize = NULL в нашем примере.
u_outspec = NULL в нашем примере.
set_idp_options = NULL в нашем примере.

Кроме этих функцией заслуживают упоминания следующие три поля:

shnames = NULL-terminated массив символьных указателей, которые указывают на короткие имена (менее чем девять символов), описывающие процессор (например, *python*). Массив заканчивается NULL-указателем.
lnames = NULL-terminated массив символьных указателей, которые указывают на полные имена, описывающие процессор (например, *Python 2.4 byte code*). Массив должен содержать такое же число элементов как в **shnames** массиве.

asms = NULL-terminated массив указателей на структуры текущего ассемблера (**asm_t**).

shnames и **lnames** массивы определяют имена всех типов процессоров, которые могут быть обработаны текущим модулем процессора. Пользователи могут выбрать альтернативные процессоры на вкладке Analysis диалога Options -> General, как показано на рисунке 19-1.

Процессорные модули с поддержкой нескольких процессоров должны обрабатывать код уведомления **processor_t.newprc** для того, чтобы получать информацию о смене процессора.

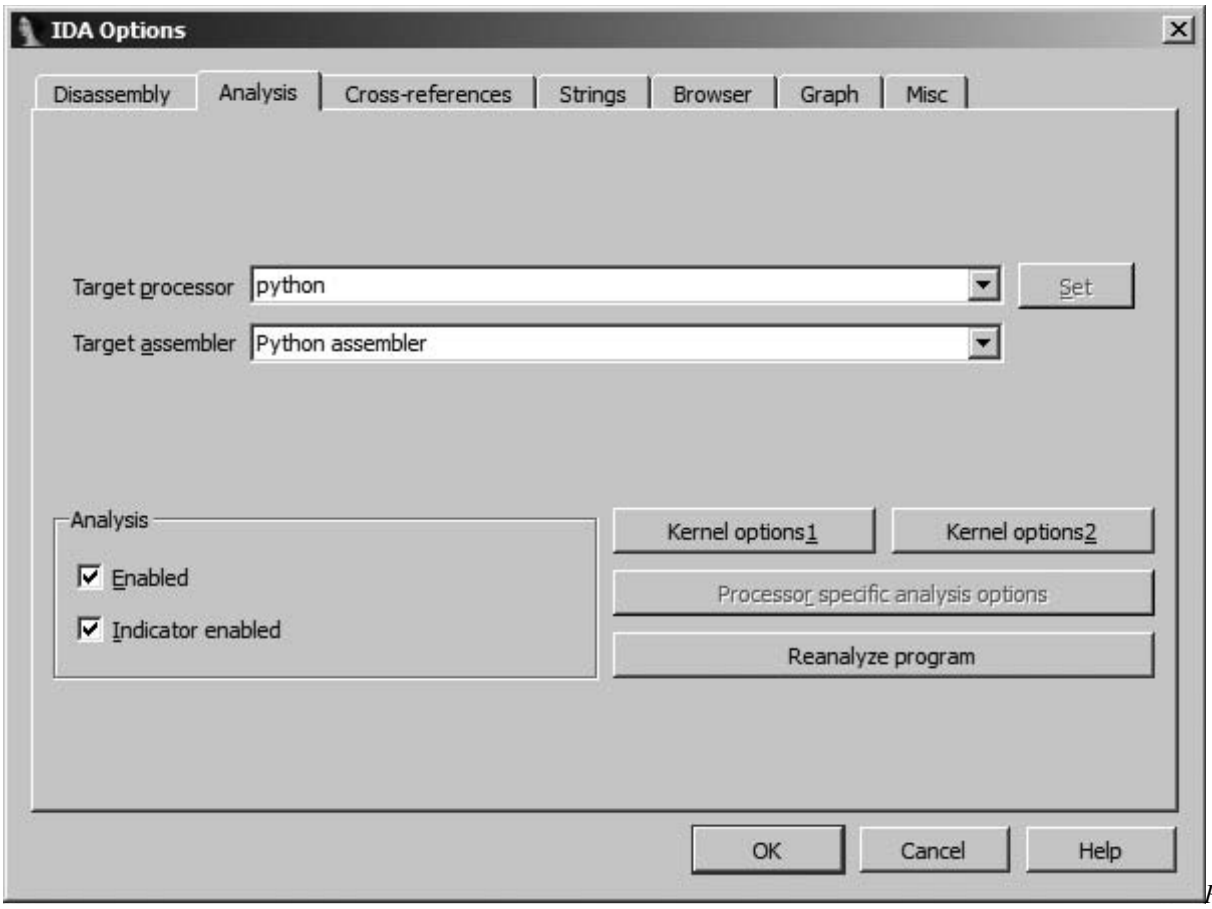

Рис. 19-1

Структура **asm_t** используется для описания некоторых синтаксических элементов языка ассемблера, такие как формат шестнадцатеричных чисел, строк и символьных разделителей, а также различные ключевые слова, обычно используемые в этом языке. Смысл поля **asms** (напомним, это массив указателей) в том, чтобы разрешить несколько различных стилей-ассемблеров, создаваемые одним процессорным модулем. Процессорные модули, которые поддерживают несколько ассемблеров должны обрабатывать код уведомления **processor_t.newasm**, чтобы информировать процессор об изменении языка.

В конечном счете, полная версия нашего простого ПМ Python способна генерировать код, такой, как нижеследующий:

```
ROM:00156       LOAD_CONST 12
ROM:00159       COMPARE_OP ==
```

```
ROM:00162               JUMP_IF_FALSE loc_182
ROM:00165               POP_TOP
ROM:00166               LOAD_NAME 4
ROM:00169               LOAD_ATTR 10
ROM:00172               LOAD_NAME 5
ROM:00175               CALL_FUNCTION 1
ROM:00178               POP_TOP
ROM:00179               JUMP_FORWARD loc_183
ROM:00182 # ------------------------------------------------------------
ROM:00182 loc_182:                                  # CODE XREF: ROM:00162j
ROM:00182               POP_TOP
ROM:00183
ROM:00183 loc_183:                                  # CODE XREF: ROM:00179j
ROM:00183               LOAD_CONST 0
ROM:00186               RETURN_VALUE
```

Хотя вполне возможно, что для создания дизассемблированного кода Python с более полной информацией, чем приведенный выше, потребуется гораздо больше знаний формата файла *.pyc*, чем предполагалось для этого примера. Немного более полнофункциональный модуль ПМ Python доступен на веб-сайте книги.

СБОРКА ПРОЦЕССОРНОГО МОДУЛЯ

Процесс сборки и установки процессорного модуля IDA очень похож на процесс сборки плагинов и загрузчиков, с одним главным отличием, что если что-то не выполнилось, то IDA не сможет использовать ваш процессор. Некоторые незначительные различия в процессе сборки включают в себя:

1. Расширения файлов для процессорных модулей: .w32/.w64 на Windows, .ilx/.ilx64 на Linux, .imc/.imc64 на OS X
2. Скрипты для сборки примеров ПМ SDK (также как и наших собственных ПМ) хранят вновь созданные исполняемые файлы процессоров в *<SDKDIR>/bin/procs*
3. Установка ПМ проще некуда. Надо скопировать скомпилированные файлы процессоров в папку *<IDADIR>/procs*
4. Для процессорных модулей на платформе Windows необходимо использовать индивидуальную MS-DOS заглушку (*эта заглушка включает в себя MS-DOS заголовок файла, а также код, смысл которого предупредить пользователей, что программа Windows не может быть выполнена в режиме MS-DOS)*, поставляемую с SDK.
5. Для процессорных модулей на платформе Windows необходим индивидуальный этап постобработки, который не нужен для плагинов и загрузчиков. Его цель заключается во вставке строки описания процессора в определенное место скомпилированного файла процессорного модуля. Строка описания отображается в выпадающем списке процессоров в диалоговом окне загрузки файла.

При создании ПМ на платформе Windows Вы должны использовать индивидуальную MS-DOS заглушку, поставляемую с SDK (*<SDKDIR>/module/stub*). Для этого Вы должны сообщить компоновщику

(линкеру), что надо использовать именно свою заглушку, а не ту, что идет по умолчанию. При использовании Windows-ориентированных компиляторов есть возможность указать альтернативную заглушку с помощью файлов определения модуля (*.def*). Компиляторы Borland (используемые самими авторами IDA) поддерживают спецификацию альтернативных заглушек с помощью *.def* файлов. SDK включает в себя файл *<SDKDIR>/module/idp.def* на случай, если Вы будете использовать Borland инструменты. И GNU, и Microsoft линкеры поддерживают файлы *.def* (хотя и с немного другим синтаксисом); однако, ни один из них не поддерживает спецификацию альтернативной MS-DOS заглушки, из-за чего возникает проблема, когда Вы используете один из этих компиляторов.

Допустим на минутку, что Вам удалось собрать свой процессорный модуль с MS-DOS заглушкой, поставляемой с SDK, далее Вам все равно необходимо вставить строку-описание процессора в выполняемый образ ПМ. Этой цели служит утилита *<SDKDIR>/bin/mkidp.exe*. Вы можете добавить описание процессора, используя следующий синтаксис для вызова утилиты **mkidp**:

```
$ mkidp module description
```

Где *module* - это путь до Вашего процессорного модуля, а *description* - это текстовое описание Вашего модуля в следующем шаблоне:

```
Полное имя модуля:короткое имя модуля
```

Добавить описание к нашему ПМ Python мы можем с помощью следующей команды:

```
$ ./mkidp procs/python.w32 "Python Bytecode:python"
```

Утилита mkidp пытается вставить строку в указанный модуль по смещению в 128 байт от начала файла, в пространство, которое лежит между MS-DOS заглушкой и PE-заголовком, предполагая, что такое место существует. Если PE-заголовок находится слишком близко к концу MSDOS заглушки, и, соответственно, просто не хватит места, то Вы получите следующее сообщение об ошибке:

```
mkidp: too long processor description
```

На данном этапе здесь всё зависит от Ваших инструментов, потому что у процессоров, собранных с помощью компоновщика от Microsoft, будет достаточно места, чтобы вставить описание, в то время как у процессоров, собранных с помощью компоновщика GNU, места не хватит.

Для того, чтобы для себя прояснить ситуацию и использовать либо Microsoft, либо GNU компиляторы, мы разработали утилиту, которую назвали **fix_proc**, доступную в секции “Глава 19” веб-сайта книги. Утилита **fix_proc** использует тот же синтаксис командной строки, что и **mkidp**, но обеспечивает дополнительные возможности, позволяющие ей вставить описание процессора в процессорный модуль, собранный в большинстве компиляторов. Когда **fix_proc** выполняется, он заменяет существующую MS-DOS заглушку процессора заглушкой, поставляемой с SDK (тем самым устраняя необходимость в *.def* файлах в процессе сборки). В то же самое время, **fix_proc** выполняет необходимые действия о перемещении PE-заголовка процессора так, чтобы было достаточно места для хранения строки описания процессора, а затем и добавляет эту строку по нужному смещению в двоичном файле. Мы используем **fix_proc** как замену **mkidp** в выполнении необходимых шагов постобработки для процессорных модулей.

Замечание.

Строго говоря, использовать MS-DOS заглушку SDK для процессорных модулей не обязательно. IDA устроит такой ПМ, в котором она найдет 128 байт строки описания процессора. В fix_proc мы заменяем существующую MS-DOS заглушку заглушкой SDK просто для того, чтобы избежать возможных конфликтов из-за пространства, выделенного для строки-описания.

Таблица 19-1 описывает какие особенности вносят компиляторы при сборке процессорных модулей.

Только процессоры с правильными форматами строк-описаний будут перечислены в диалоговом окне загрузки файла. Другими словами, без корректного поля описания не возможно выбрать процессорный модуль.

	Первоначальная сборка		После mkidp		После fix_proc	
Компилятор	Исп. .def ?	Своя MS-DOS заглушка?	Своя MS-DOS заглушка?	Есть поле описание?	Своя MS-DOS заглушка?	Есть поле описание?
Borland	Да	Да	Да	Да	Да	Да
Microsoft	Нет	Нет	Нет	Да	Да	Да
GNU	Нет	Нет	Нет	Нет	Да	Да

Все эти различия в процессе сборки требуют добавить еще несколько изменений в Makefile, на котором был собран загрузчик (*из предшествующей главы 18 - от перев.*). В листинге 19-1 представлен Makefile, измененный для создания нашего примера ПМ Python.

```
#Установить значение этой переменной, указывающей на Ваш каталог SDK
IDA_SDK=../../
PLATFORM=$(shell uname | cut -f 1 -d _)
ifneq "$(PLATFORM)" "MINGW32"
IDA=$(HOME)/ida
endif
#Установите значение этой переменной желаемому имени Вашего процессора
PROC=python
#Укажите строку-описание для Вашего процессора
#согласно синтаксиса <полное имя>:<короткое имя>
❶ DESCRIPTION=Python Bytecode:python
ifeq "$(PLATFORM)" "MINGW32"
PLATFORM_CFLAGS=-D__NT__ -D__IDP__ -DWIN32 -Os -fno-rtti
PLATFORM_LDFLAGS=-shared -s
LIBDIR=$(shell find ../../ -type d | grep -E "(lib|lib/)gcc.w32")
ifeq ($(strip $(LIBDIR)),)
LIBDIR=../../lib/x86_win_gcc_32
```

```
endif
else ifeq "$(PLATFORM)" "Linux"
PLATFORM_CFLAGS=-D__LINUX__
PLATFORM_LDFLAGS=-shared -s
IDALIB=-lida
IDADIR=-L$(IDA)
PROC_EXT=.ilx

else ifeq "$(PLATFORM)" "Darwin"
PLATFORM_CFLAGS=-D__MAC__
PLATFORM_LDFLAGS=-dynamiclib
IDALIB=-lida
IDADIR=-L$(IDA)/idaq.app/Contents/MacOs
PROC_EXT=.imc
endif

#Платформо-зависимые аргументы компилятора
CFLAGS=-Wextra $(PLATFORM_CFLAGS)

#Платформо-зависимые ld флажки
LDFLAGS=$(PLATFORM_LDFLAGS)

#укажите по необходимости любые дополнительные библиотеки
EXTRALIBS=

# Каталог назначения, где будут размещены откомпилированные образы
OUTDIR=$(IDA_SDK)bin/procs/

# Инструмент постобработки для добавления комментария процессора
❷ MKIDP=$(IDA_SDK)bin/fix_proc
#MKIDP=$(IDA)bin/mkidp

#список объектных файлов Вашего проекта
OBJS=ana.o emu.o ins.o out.o reg.o

BINARY=$(OUTDIR)$(PROC)$(PROC_EXT)

all: $(OUTDIR) $(BINARY)
clean:
        -@rm *.o
        -@rm $(BINARY)

$(OUTDIR):
        -@mkdir -p $(OUTDIR)
CC=g++
INC=-I$(IDA_SDK)include/
```

```
%.o: %.cpp
	$(CC) -c $(CFLAGS) $(INC) $< -o $@
LD=g++
ifeq "$(PLATFORM)" "MINGW32"
#процессоры под Windows требуют постобработки
$(BINARY): $(OBJS)
	$(LD) $(LDFLAGS) -o $@ $(OBJS) $(IDALIB) $(EXTRALIBS)
❸	$(MKIDP) $(BINARY) "$(DESCRIPTION)"
else
$(BINARY): $(OBJS)
	$(LD) $(LDFLAGS) -o $@ $(OBJS) $(IDALIB) $(EXTRALIBS)
endif

#измените python (см. ниже) на имя Вашего процессора, не забудьте добавить любые
#дополнительные файлы, от которых зависит Ваш процессор
python.o: python.cpp
ana.o: ana.cpp
emu.o: emu.cpp
ins.o: ins.cpp
out.o: out.cpp
reg.o: reg.cpp
IDALIB=$(LIBDIR)/ida.a
PROC_EXT=.w32
```

Листинг 19-1: makefile нашего процессорного модуля Python

Кроме незначительных поправок, как изменения суффиксов и папки расположения файлов для ПМ по умолчанию, основные различия - определение строки описания ❶, определение утилиты для вставки строки описания ❷, и добавляем этап сборки для вставки строки описания в процессорные модуля на платформе Windows ❸.

НАСТРОЙКА СУЩЕСТВУЮЩИХ ПРОЦЕССОРОВ

Возможно, Вы собираетесь разрабатывать процессорный модуль, но тут замечаете, что существующий модуль делает почти все, что Вам нужно. Если у Вас есть исходный код процессорного модуля, то Вы можете легко изменить его в соответствии с Вашими потребностями. С другой стороны, если у Вас нет исходного кода, то возникает чувство, что Вам не повезло. К счастью, IDA предлагает механизм по настройке существующих процессоров за счет использования плагинов. Перехватив соответствующие коды уведомления процессора, плагин будет перехватывать вызовы одного или нескольких этапов существующих процессоров - анализаторов, эмуляторов, и outputter. Потенциальные области применения настройки существующих процессоров:

- Расширение возможностей существующих процессоров для распознавания дополнительных инструкций.

- Исправление ошибок в существующих процессорных модулях (хотя, возможно, быстрее будет сообщить Ильфаку о том, что вы нашли ошибку).
- Подстройка вывода на экран существующих процессорных модулей в соответствии с Вашими особыми потребностями

Следующие коды уведомления, объявленные в **processor_t** и описанные в *idp.hpp*, могут быть обработаны плагинами для перехвата вызовов на различных этапах работы процессора:

- **custom_ana** Ведет себя также как **u_ana**, однако, значение **cmd.itype** у любых новых инструкций должно быть равно 0x8000 и выше.
- **custom_emu** Для эмуляции пользовательских типов инструкций. Если Вы хотите передать управление существующему эмулятору процессора, то вызовите **(* ph.u_emu) ()**.
- **custom_out** Формирует вывод на экран пользовательских инструкций или предоставляет свой вариант вывода на экран для существующих инструкций. Если Вы хотите передать управление функции **out** процессора, то вызовите **(* ph.u_out) ()**.
- **custom_outop** Вывод одного пользовательского операнда. Если Вы хотите передать управление функции **outop** существующего процессора, то вызовите **(* ph.u_outop) (op)**.
- **custom_mnem** Генерируем мнемонику пользовательских инструкций.

Следующий отрывок кода из плагина, который изменяет вывод на экран процессорного модуля x86, а именно, заменяет команду **leave** командой **cya** и меняет местами операнды (только при выводе на экран) у тех инструкций, которые имеют два операнда (по аналогии с AT&T-синтаксисом):

```
int idaapi init(void) {
(1)     if (ph.id != PLFM_386) return PLUGIN_SKIP;
(2)             hook_to_notification_point(HT_IDP, hook, NULL);
        return PLUGIN_KEEP;
}

int idaapi hook(void *user_data, int notification_code, va_list va) {
        switch (notification_code) {
                case processor_t::custom_out: {
(3)                     if (cmd.itype == NN_leave) { //перехват инструкции leave
(4)                             MakeLine(SCOLOR_ON SCOLOR_INSN "cya" SCOLOR_OFF);
                                return 2;
                        }
                        else if (cmd.Op2.type != o_void) {
                                //перехват инструкций с 2-мя инструкциями
                                op_t op1 = cmd.Op1;
                                op_t op2 = cmd.Op2;
                                cmd.Op1 = op2;
                                cmd.Op2 = op1;
(5)                             (*ph.u_out)();
                                cmd.Op1 = op1;
                                cmd.Op2 = op2;
                                return 2;
```

```
                        }
                    }
            }
            return 0;
}

plugin_t PLUGIN = {
            IDP_INTERFACE_VERSION,
(6)         PLUGIN_PROC | PLUGIN_HIDE | PLUGIN_MOD, // флажки плагина
            init,               // функция инициализации
            term,               // прервать работу. этот указатель может быть установлен в NULL.
            run,                // invoke plugin
            comment,            // подробный комментарий о плагине
            help,               // многострочный help о плагине
            wanted_name,        // предпочительное краткое имя плагина
            wanted_hotkey       // предпочительная горячая клавиша для запуска плагина
};
```

Функция **init** проверяет, что текущий процессор - это x86-процессор (1) и тогда ставим функцию перехвата уведомления процессора (2). В функции обратного вызова **hook** плагин обрабатывает уведомление **custom_out**, чтобы определить инструкцию **leave** (3) и заменяет ее альтернативной **cya** (4) при выводе на экран. Для команд с 2-мя операндами, функция **hook** временно сохраняет операнды текущей инструкции, затем меняем их местами перед непосредственным вызовом функции **u_out** x86-процессора (5), которая “рисует” инструкцию на экране. По возвращении из **u_out** операнды команды меняются местами обратно, восстанавливая оригинальный порядок. Наконец, флажки (6) обозначают, что плагин должен быть загружен в тоже время, что и процессор, плагин не должен быть виден в меню Edit -> Plugins и он изменяет базу данных. Следующий листинг показывает какой эффект вызывает плагин при выполнении:

```
            .text:00401350  push    ebp
(7)         .text:00401351  mov     400000h, edx
            .text:00401356  mov     esp, ebp
(7)         .text:00401358  mov     offset unk_402060, eax
(7)         .text:0040135D  sub     0Ch, esp
            .text:00401360  mov     edx, [esp+8]
            .text:00401364  mov     eax, [esp+4]
(7)         .text:00401368  mov     offset unk_402060, [esp]
            .text:0040136F  call    sub_401320
(8)         .text:00401374  cya
            .text:00401375  retn
```

В листинге можно наблюдать результат работы плагина, заметив, что константы появляются в качестве первого операнда в четырех инструкциях (7) и инструкция **cya** используется вместо инструкции **leave** (8).

В главе 21 рассмотрим как нам поможет плагин настройки процессора в анализе отдельных видов обфускации бинарных файлов.

А Р Х И Т Е К Т У Р А П Р О Ц Е С С О Р Н О Г О М О Д У Л Я

Как только Вы приступили к проектированию процессорных модулей, есть одна вещь, которую Вы должны рассмотреть, а именно, будет ли процессор тесно связан с конкретным загрузчиком или будет независим от любого загрузчика. Например, рассмотрим модуль x86-процессора. Этот модуль не делает никаких предположений о типе файла, который в настоящее время дизассемблируется. Поэтому его легко комбинировать и использовать в сочетании с широким набором загрузчиков, таких как PE, ELF и Mach-O.

Аналогично, загрузчики показывают свою универсальность при обработке формата файла независимо от используемого процессора с этим файлом. Например, PE-загрузчик работает одинаково хорошо с файлом, который содержит будь то x86 или ARM код; ELF-загрузчик одинаково хорошо работает с файлом, который содержит будь то x86, MIPS или SPARC код; а Mach-O-загрузчик прекрасно работает с файлом, где есть PPC или x86 код.

Реальные процессоры идеально подходят для создания процессорных модулей, которые не полагаются на определенный формат входного файла. С другой стороны, языки виртуальной машины представляют собой гораздо серьезную задачу. В то время как разнообразные загрузчики (например, ELF, a.out, и PE) могут использовать загруженный код для выполнения на родном аппаратном обеспечении, виртуальная машина обычно выступает одновременно и загрузчиком, и процессором. В результате для виртуальной машины и формат файла и находящийся в нем байт-код тесно связаны между собой. Одно не может существовать без другого. Мы сталкивались с этим ограничением несколько раз при разработке ПМ Python. Во многих случаях просто не было возможности для создания более читабельного дизассемблированного кода без глубокого понимания структуры файла.

Для того, чтобы процессор Python имел доступ к необходимой дополнительной информации, мы могли бы сделать Python-загрузчик, который настраивает базу данных специальным образом так, что процессор Python точно знает, где искать необходимую информацию . В этом случае значительный объем данных - состояния загрузки - должны перейти от загрузчика к процессору. Один из подходов заключается в хранении подобных данных в базе данных netnodes (*низкоуровневой механизм хранения данных в IDA описан в главе 16 - от перев.*), откуда они могут быть извлечены в дальнейшем с помощью процессорного модуля.

Альтернативный подход - создать загрузчик, который не делает ничего другого, кроме распознавания *.pyc* файлов, сообщая процессорному модулю, что он должен обрабатывать все другие задачи загрузки; в этом случае процессор будет знать наверняка, как найти всю информацию, необходимую для дизассемблирования *.pyc* файла.

IDA упрощает создание тесно связанных загрузчиков и процессорных модулей, позволяя загрузчику переложить все операции загрузки на связанный с ним модуль процессора. Вот как Java загрузчик и Java процессор, включенные в SDK, построены. Для того, чтобы отложить загрузку модуля процессора, загрузчик должен сначала принять файл, возвращая тип файла **f_LOADER** (определенный в *ida.hpp*). Если загрузчик выбран пользователем, то функция **load_file** загрузчика в случае необходимости должна указать соответствующий тип процессора с помощью вызова **set_processor_type** (*idp.hpp*), прежде чем отправить сообщение-уведомление загрузки процессору. Чтобы собрать такую комбинацию загрузчик/процессор Python, мы могли бы написать загрузчик со следующей функцией **load_file**:

```
void idaapi load_file(linput_t *li, ushort neflag, const char *) {
        if (ph.id != PLFM_PYTHON) { //общий ID процессора
                set_processor_type("python", SETPROC_ALL|SETPROC_FATAL);
```

```
        }
        //говорим процессору python выполнить загрузку для нас
        //посылая код уведомления processor_t::loader
        if (ph.notify(processor_t::loader, li, neflag)) {
                error("Python processor/loader failed");
        }
}
```

Когда процессорный модуль получает уведомление **loader**, он берет на себя ответственность за отображение входного файла на базу данных и проверяет, доступна ли ему любая информации, необходимая для этапов **ana**, **emu**, и **out**. Связку загрузчика и процессора Python, которая работает таким образом можно получить на сайте книги.

П И Ш Е М П Р О Ц Е С С О Р Н Ы Й М О Д У Л Ь Н А С К Р И П Т Е

Представленная в IDA 5.7 возможность создания процессорных модулей с помощью одного из скриптовых языков IDA весьма упрощает процесс создания. Как минимум при создании полностью исключается фаза компиляции модуля. Elias Bachaalany из Hex-Rays представил статью “Пишем ПМ на скрипте” в блоге Hex Blog (http://www.hexblog.com/?p=116) и байт-код ПМ EFI в IDA реализован в виде скрипта на языке Python (см. *<IDADIR>/procs/ebc.py*). Обратите внимание, что в то время как пост Hex Blog задает новый уровень работы с IDA, фактически API, используемый в ПМ, написанном на скриптовом языке, похоже, еще в развитии. Лучший способ разрабатывать собственный скриптовый ПМ - это начать с шаблона модуля, поставляемого вместе с SDK (см. *<SDKDIR>/module/script/proctemplate.py*). Среди прочего, шаблон перечисляет все поля, обязательные в ПМ Python.

В скриптовом ПМ присутствуют почти все элементы описанные выше. Понимание этих элементов облегчит Вам переход к написанию скриптовых модулей. Кроме того, три скриптовых процессорных модуля, поставляемые в настоящее время вместе с IDA (согласно IDA 6.1: *ebc.py, msp430.py, spu.py*) послужат отличным примером для разработки Ваших собственных модулей. Структуру этих модулей чуть легче понять, чем примеры на C++, поставляемые вместе с SDK, которые охватывают несколько файлов и требуют правильной настройки среды сборки.

По большому счету две вещи необходимо для создания ПМ, написанного на языке Python:

- Определить подкласс **idaapi.processor_t**, обеспечивающий реализацию для всех необходимых функций процессорного модуля, таких как **emu**, **ana**, **out**, и **outop**.
- Определить функцию **PROCESSOR_ENTRY** (не член Вашего подкласса), которая возвращает экземпляр класса процессора.

Следующий листинг показывает, как реализованы необходимые элементы:

```
from idaapi import *

class demo_processor_t(idaapi.processor_t):
        # Инициализация небходимых полей процессора, включая id и
        # assembler и многие другие. Поле assembler - это словарь (ассоциативный массив для хранения пар
```

```
        # ключ-значение, где значение однозначно определяется ключом), который содержит ключи для
        # всех полей asm_t. Список инструкций, именуемый instruc также необходим. Каждый элемент
        # этого списка - двухэлементный словарь, содержащий ключи name и feature.
        # Здесь же определены функции, необходимые для processor_t. Они перечислены ниже.
        def ana(self):
                # анализатор
        def emu(self):
                # эмулятор
        def out(self):
                # генератор печати
        def outop(self):
                # функция вывода операндов outop

# определить точку входа ПМ, которая создает и возвращает экземпляр processor_t
def PROCESSOR_ENTRY():
        return demo_processor_t()
```

На самом деле скриптовый ПМ содержит гораздо больше полей и функций, чем указано выше, по существу его поля - это зеркальное отображение полей, необходимых в любом процессорном модуле, реализованном на C++. После того как скрипт написан, его установка происходит простым копированием Вашего скрипта в каталог *<IDADIR>/procs*.

К Р А Т К О

Как наиболее сложное модульное расширение IDA, процессорные модули потребует от Вас времени для обучения и еще больше времени для их создания, хотя создание процессоров с помощью скриптов на языке Python уменьшит количество телодвижений в некоторой степени. Однако, если у Вас сфера деятельности связана с рынком реверс-инжиниринга, или Вы просто хотите быть на переднем крае в сообществе реверс-инжиниринга, то почти наверняка Вы найдете для себя необходимость разработки процессорного модуля в некоторый момент. Мы не можем достаточно оценить какую роль играет терпение или метод проб и ошибок при разработки любого процессора. Но тяжелая работа более чем окупится, когда вы сможете повторно использовать модуль процессора с каждым новым бинарником, встречаемый Вами.

В конце этой главы, мы закончим обсуждение модулей расширения IDA. В последующих главах мы обсудим многие способы применения IDA, используемые в реальных сценариях, и посмотрим, как пользователи, прибегая к расширениям IDA, выполняют множество интересных задач анализа.