# Расширенный ассемблер: NASM

Перевод: AsmOS group, © 2001

## *Глава 1. Введение*

### 1.1 Что такое NASM?

Расширенный ассемблер NASM – это 80x86 ассемблер, разработанный исходя из принципов переносимости и модульности. Он поддерживает широкий диапазон форматов объектных файлов, включая форматы Linux a.out и ELF, NetBSD/FreeBSD, COFF, Microsoft 16-bit OBJ и Win32. Он способен также создавать простые бинарные файлы. Синтакс NASM максимально упрощен для понимания и похож на синакс Intel, но слегка посложнее. Он поддерживает инструкции Pentium, P6 и MMX, а также имеет макро-расширения.

#### *1.1.1 Зачем еще один ассемблер?*

Расширенный ассемблер вырос из идеи, поданной на comp.lang.asm.x86 (или возможно на alt.lang.asm - я забыл уже, где), когда по существу не было хорошего свободно доступного ассемблера серии x86 и нужно было, чтобы кто-то написал его.

- a86 — хороший ассемблер, но не бесплатный, и если вы не заплатите, то 32-битный код писать не сможете — только DOS.
- gas свободно доступен и портирован под DOS и Unix, но разработан для обратной совместимости с gcc. Поэтому проверка ошибок минимальна, к тому-же, с точки зрения любого, кто попробовал что-либо *написать* в нем — синтаксис ужасен. Плюс ко всему вы не можете в нем написать 16-разрядный код (по крайней мере, правильно).
- as86 — только под Linux и (по крайней мере моя версия) кажется не имеет практически никакой документации.
- MASM очень хорош, очень дорог и работает только под DOS.
- TASM лучше, но все еще борется с MASM за совместимость, что означает миллионы директив и куча волокиты. Его синтаксис — по существу MASM-овский, но с противоречиями и причудами (в некоторой степени удаляемыми посредством режима Ideal). Он также дорогой и тоже — только ДОС.

Таким образом, представляем на ваше рассмотрение NASM. В сегодняшнем виде он все еще находится в стадии прототипа — мы не обещаем, что он будет превосходить по быстродействию любой из упомянутых выше ассемблеров. Но пожалуйста, *пожалуйста* шлите нам замечания о замеченных ошибках, исправления, полезную информацию, да все, что угодно, что вы можете передать нам (и спасибо огромное многим людям, кто уже сделал это!), и мы будем улучшать его (в смысле, NASM) снова и снова.

#### *1.1.2 Условия лицензирования*

Чтобы ознакомиться с условиями лицензирования, при которых вы можете пользоваться NASM, пожалуйста, прочитайте файл Licence, являющийся неотъемлимой частью любого дистрибутивного архива NASM.

### 1.2 Контакты

Текущая версия NASM (0.98) поддерживается H. Peter Anvin, hpa@zytor.com. Если вы захотите сообщить об обнаруженных ошибках, прочитайте сначала параграф 10.2.

Страничка NASM в интернете WWW – http://www.cryogen.com/Nasm.

Связаться с авторами можно по следующим адресам: jules@earthcorp.com и anakin@pobox.com.

Новые релизы NASM доступны на ftp.kernel.org, sunsite.unc.edu, ftp.simtel.net и ftp.coast.net. Уведомления и объявления смотрите на comp.lang.asm.x86, alt.lang.asm, comp.os.linux.announce и comp.archives.msdos.announce (последнее из уведомлений автоматически закачивается на ftp.simtel.net).

Если вы не имеете доступ к Usenet или предпочитаете получать информацию о выпусках новых версий по электронной почте, вы можете подписаться на лист рассылки nasm-announce, послав email по адресу majordomo@linux.kernel.org, содержащий строку subscribe nasm-announce.

Если вы хотите также получать информацию о выходе бета-релизов NASM, пошлите по тому-же адресу (см.выше) письмо, содержащее строку nasm-beta.

## 1.3 Инсталляция

### *1.3.1 Инсталляция NASM под MS-DOS или Windows*

При получении DOS-архива NASM, nasmXXX.zip (где XXX означает номер версии NASM, содержащегося в архиве), распакуйте его в отдельный каталог (например, c:\nasm).

Архив содержит четыре исполняемых файла: NASM-исполняемые файлы nasm.exe и nasmw.exe, и NDISASM-исполняемые файлы ndisasm.exe и ndisasmw.exe. Файлы, имеющие в окончании имени w, работают под Win9x/ME/NT, а те, которые без w — работают под DOS-ом.

Для запуска NASM требуются только эти файлы, так что скопируйте их в каталог, указанный в вашей переменной PATH , либо отредактируйте autoexec.bat для добавления пути к каталогу с исполнимыми файлами NASM в переменную PATH. (если вы устанавливаете только версию под Win32, можете смело переименовать nasmw.exe в nasm.exe.)

И это все!!! NASM установлен! Для запуска NASM не обязательно иметь отдельный каталог (если, конечно вы не добавили его к переменной PATH), поэтому можете удалить его (каталог), если у вас мало места на диске.

Если вы загрузили DOS-архив с исходниками nasmXXXs.zip, он будет также содержать полный исходный текст NASM и набор Make-файлов, которые вы можете (будем надеяться) использовать для перестроения вашей копии NASM "с нуля". В файле Readme перечислены все Make-файлы и указано, с какими компиляторами они работают.

Обратите внимание, что исходники insnsa.c, insnsd.c, insnsi.h и insnsn.c автоматически генерируются из главной таблицы инструкций insns.dat Perl-скриптом; файл macros.c генерируется из standard.mac другим Perl-скриптом. Хотя дистрибутив NASM 0.98 и включает эти автогенерируемые файлы, вам может потребоваться перестроить их (и следовательно, вам будет необходим интерпретатор Perl), если вы захотите изменить insns.dat, standard.mac или документацию. Возможно в будущем в исходниках не будет этих файлов вовсе. Версии Perl для ряда платформ, включая DOS и Windows, доступны на www.cpan.org.

### *1.3.2 Инсталляция NASM под Unix*

При получении Unix-архива исходников NASM, nasm-X.XX.tar.gz (где X.XX означает номер версии NASM в архиве) распакуйте его в каталог типа /usr/local/src. Архив при распаковке создаст собственный подкаталог nasm-X.XX.

NASM — автоконфигурируемый пакет: как только вы распакуете его, перейдите к каталогу, куда он был распакован и введитее ./configure. Данный шелл-скрипт найдет самый подходящий компилятор C для сборки NASM и, соответственно, настройки Make-файлов.

Как только NASM сконфигурируется, вы можете ввести make для сборки бинарных файлов nasm и ndisasm, а затем ввести make install для установки их в /usr/local/bin и установки man-страниц nasm.1 и ndisasm.1 в /usr/local/man/man1. В качестве альтернативы вы можете указать опции типа --prefix к команде configure скрипта (подробности см. в файле INSTALL) или установить программы самостоятельно.

NASM также имеет набор утилит для обработки заказного формата объектных файлов RDOFF, находящихся в подкаталоге rdoff архива NASM. Вы можете собрать их при помощи make rdf и установить при помощи make rdf_install, если конечно они вам нужны.

Если NASM будет не в состоянии автоматически выбрать конфигурацию, вы все-же сможете скомпилировать его при помощи make-файла Makefile.unx. Скопируйте или переименуйте этот файл в Makefile и попробуйте ввести make. Имеется также файл Makefile.unx в подкаталоге rdoff.

## *Глава 2: Запуск NASM*



### 2.1 Синтаксис командной строки NASM

Для ассемблирования файла вы должны ввести следующую команду:

```
nasm -f <format> <filename> [-o    <output>]
```

Например,

```
nasm -f elf myfile.asm
```

будет ассемблировать myfile.asm в ELF-объектный файл myfile.o. А строка

```
nasm -f bin myfile.asm -o myfile.com
```

будет ассемблировать myfile.asm в обычный бинарный файл myfile.com. Для получения файла-листинга, содержащего слева от оригинального исходного текста шестнадцатиричные коды, генерируемые NASM, используйте ключ -l, обозначающий имя файла-листинга, например:

```
nasm -f coff myfile.asm -l myfile.lst
```

Для получения справки по командной строке NASM, укажите следующий ключ:

```
nasm -h
```

При этом вы получите также список доступных форматов выходных файлов и что они означают. Если вы используете Linux, но не уверены, какая ваша система — a.out или ELF, введите

```
file nasm
```

в каталоге, где находятся бинарные файлы NASM. В ответ вы получите что-то вроде

```
nasm: ELF 32-bit LSB executable i386 (386 and up) Version    1
```

Это означает, что ваша система — ELF и вы должны при ассемблировании использовать ключ -f elf. Если же вы увидите

```
nasm: Linux/i386 demand-paged executable (QMAGIC)
```

или что-то наподобие этого, ваша система — a.out, и при ассемблировании нужно будет указать ключ -f aout. (Linux системы a.out считаются устаревшими и на сегодняшний день встречаются редко). Подобно Unix компиляторам и ассемблерам, NASM "бесшумен": вы не будете видеть никаких сообщений вообще, если только это не сообщения об ошибках.

#### *2.1.1 Ключ -o: Указание имени выходного файла*

Обычно NASM выбирает имя выходного файла самостоятельно; так как это зависит от формата объектного файла. Если формат объектного файла — Microsoft (obj и win32), он удалит расширение .asm (или любое другое, какое вам нравится использовать — NASMу все равно) из имени исходного файла и заменит его на .obj. У объектных файлов Unix-формата (aout, coff, elf и as86) он будет заменять расширение на .o. Для формата rdf он будет использовать расширение .rdf, а в случае формата bin он просто удалит расширение, например из myfile.asm получится файл myfile.

Если выходной файл уже существует, NASM перезапишет его, если только его имя не совпадает с именем входного файла — в этом случае появится предупреждение и в качестве выходного файла будет использовано имя nasm.out.

В случаях, когда имя по умолчанию недопустимо, используйте ключ -o командной строки, позволяющий определить необходимое вам имя выходного файла. Имя выходного файла должно следовать за ключем -o, неважно с пробелом между ними или без. Например:

```
nasm -f bin program.asm -o program.com
nasm -f bin driver.asm -odriver.sys
```

### 2.1.2 Ключ -f: Указание формата выходного файла

Если ключ -f в командной строке отсутствует, NASM будет выбирать формат выходного файла самостоятельно. В распространяемой версии NASM формат по умолчанию всегда bin; если вы создаете собственную копию NASM, то при компиляции можете переопределить значение OF_DEFAULT на то, которое вам нужно по умолчанию.

Как и для ключа -o, разделительный пробел между -f и форматом выходного файла необязателен: -f elf и -felf для NASM идентичны.

Полный список доступных выходных форматов может быть получен при помощи команды `nasm -h`.

### 2.1.3 Ключ -l: Генерация файла-листинга

Если в командной строке вы укажете ключ -l и имя файла (как обычно, пробел необязателен), NASM будет генерировать из исходника файл-листинг, где адреса и генерируемый код будут расположены слева, а исходный код с развернутыми многострочными макросами (за исключением тех, которые специально требуют обратное: см. параграф 4.2.9) — справа. Например:

```
nasm -f elf myfile.asm -l myfile.lst
```

### 2.1.4 Ключ -E: Перенаправление ошибок в файл

Под MS-DOS перенаправление стандартного потока ошибок в файл может быть сопряжено с трудностями (хотя способы имеются). Так как NASM обычно направляет предупреждения и сообщения об ошибках в поток stderr, перехват этих сообщений ( например для последующего просмотра в редакторе) может вызвать сложности.

В связи с этим имеется специальный ключ -E, после которого указывается имя файла (пробел необязателен). Данный ключ перенаправляет стандартный поток ошибок в указанный файл. Таким образом, вы можете запустить NASM, например следующим образом:

```
nasm -E myfile.err -f obj myfile.asm
```

### 2.1.5 Ключ -s: Перенаправление ошибок в stdout

Ключ -s перенаправляет стандартный поток ошибок stderr в выходной поток stdout (естественно, в MS-DOS). Например, для ассемблирования файла myfile.asm и передачи ошибок программе more, вы можете ввести следующее:

```
nasm -s -f obj myfile.asm | more
```

См. также ключ -E, параграф 2.1.4.

### 2.1.6 Ключ -i: Каталоги поиска включаемых файлов

Когда NASM встречает в исходнике директиву %include (см. параграф 4.5), он будет искать указанный в ней файл не только в текущем каталоге, но и во всех каталогах, указанных в командной строке при помощи ключа -i. Следовательно, вы можете включить файлы из, например, библиотеки макросов, введя следующую команду:

```
nasm -ic:\macrolib\ -f obj myfile.asm
```

(Как обычно, пробел между -i и строкой поиска допустим, но не обязателен)

С целью переносимости кода в NASMe не "зашито" соглашение об именовании файлов той или иной ОС, под которой он запущен; строка, которую вы укажете как аргумент ключа, будет обработана в точности так, как есть. Заключительный обратный слэш в приведенном выше примере под Unix необходим, т.к. в этой ОС заключительные слэши обычно требуются.

(Вы можете извлечь из этого обстоятельства выгоду, использовав данный ключ "не по назначению" — например ключ -ifoo будет заставлять директиву %include "bar.i" искать файл foobar.i...)

Если вы хотите описать *стандартный* путь поиска включаемых файлов, такой как /usr/include в системе Unix, вы должны поместить одну или более директив -i в переменную окружения NASM (см. параграф 2.1.13).

Для совместимости с make-файлами большинства C компиляторов, данный ключ может быть также задан как -I.

### 2.1.7 Ключ -p: Предварительно включаемые файлы

При помощи ключа -p NASM позволяет вам *предварительно* включить некоторые файлы в ваш исходник. Так, строка запуска

```
nasm myfile.asm -p myinc.inc
```

эквивалентна строке nasm myfile.asm и помещением в начало файла директивы %include "myinc.inc". В целях симметричности с ключами I, -D и -U данный ключ может быть также задан как -P.

### 2.1.8 Ключ -d: Предопределение макроса

Аналогично тому, как ключ -p дает альтернативу помещения в начало исходного файла директивы %include, ключ -d дает альтернативу директиве %define. Таким образом, команда

```
nasm myfile.asm -dFOO=100
```

альтернативна помещению в начало файла директивы

```
%define FOO 100
```

Вы можете также опустить значение константы: ключ -dFOO эквивалентен строке %define FOO. Данная возможность может быть полезна при ассемблировании для включения/выключения опций, проверяемых при помощи директивы %ifdef, например -dDEBUG. Для совместимости с make-файлами большинства C компиляторов, данный ключ может быть также задан как -D.

### 2.1.9 Ключ -u: Отмена определения макроса

Ключ -u отменяет определение ранее определенного макроса. Например, в результате выполнения следующей командной строки:

```
nasm myfile.asm -dFOO=100 -uFOO
```

FOO *не будет* предопределенным макросом для программы. Это полезно для временного отключения опций, заданных в make-файлах. Для совместимости с make-файлами большинства C компиляторов, данный ключ может быть также задан как -U.

### *2.1.10 Ключ -e: Только препроцессирование*

NASM допускает выполнение только препроцессирования входного файла. Использование ключа -e (не требующего параметров) заставит NASM препроцессировать входной файл, развернуть все макро ссылки, удалить все комментарии и директивы препроцессора и вывести результирующий файл в стандартный выходной поток (или сохранить его как отдельный файл, если используется также опция -o).

Данный ключ неприменим для программ, где препроцессор должен вычислить выражения, зависящие от значений адресов: такой код как

```
%assign tablesize ($-tablestart)
```

будет вызывать ошибку в режиме "только препроцессирование".

### *2.1.11 Ключ -a: Отключение препроцессора*

Если NASM используется в качестве выходной части компилятора, то для увеличения скорости компиляции желательно полностью подавить препроцессирование в случае, если компилятор уже делает его. Ключ -a, не требующий никаких параметров, заставляет NASM заменить свой препроцессор ничего не делающей "заглушкой".

### *2.1.12 Ключ -w: Разрешение/Запрещение предупреждений при ассемблировании*

В процессе ассемблирования NASM может замечать множество вещей, заслуживающих внимания пользователя, но не представляющих собой таких серьезных ошибок, чтобы NASM не смог сгенерировать выходной файл. Эти предупреждения выдаются так же, как и ошибки, но перед собственно сообщением добавляется слово "warning". Предупреждения не являются причиной прерывания процесса генерации выходного файла.

Некоторые события менее серьезны, чем другие: они только иногда заслуживают внимания пользователя. В связи с вышесказанным, NASM поддерживает ключ командной строки -w, включающий или выключающий определенные классы предупреждений. Эти классы предупреждений описываются по имени, например orphan-labels; вы можете разрешить предупреждения данного класса при помощи ключа -w+orphan-labels и запретить их ключом -w-orphan-labels.

Ниже перечислены классы подавляемых предупреждений:

- macro-params включает предупреждения о многострочных макросах, которым передается неверное число параметров. Данный класс предупреждений по умолчанию разрешен; см. параграф 4.2.1, где приведен пример отключения этого класса.
- orphan-labels включает предупреждения о строках исходника, не содержащих инструкций, но в которых описываются метки без завершающего двоеточия. NASM по умолчанию не предупреждает об этих вещах; см. пример в параграфе 3.1, где показано, как можно включить их.
- number-overflow включает предупреждения о числовых константах, не вписывающихся в диапазон 32 бита (0x7fffffff). Данный класс предупреждений по умолчанию разрешен.

### *2.1.13 Переменная окружения NASM*

Если вы определите переменную окружения NASM, программа будет интерпретировать ее как список дополнительных ключей командной строки, обрабатывающихся *раньше* "настоящих" параметров командной строки. Вы можете использовать эту возможность, например для описания стандартных каталогов поиска включаемых файлов, поместив в пременную NASM ключ -i.

Значение пременной разделяется пробелами, поэтому значение -s -ic:\nasmlib будет обработано как два отдельных ключа. В то же время значение -dNAME="my name" не будет воспринято так, как вы хотите

(внутри есть пробел) и командный процессор NASMa соответственно не поймет двух бессмысленных параметров -dNAME="my и name".

Для разрешения этого введено следующее правило: если вы начинаете переменную NASM некоторым символом, не являющимся знаком "минус", NASM будет воспринимать этот символ как разделитель опций. Таким образом, значение !-s!-ic:\nasmlib переменной NASM эквивалентно -s -ic:\nasmlib, но зато теперь !-dNAME="my name" будет работать правильно.

### 2.2 Пользователям MASM: Отличия

Если вы использовали для написания программ MASM, или TASM в режиме совместимости с MASM, или a86, прочитайте данный раздел, в котором приводятся основные отличия синтаксиса MASM и NASM. Если же вы вообще не использовали раньше MASM, просто пропустите этот раздел и читайте дальше.

#### *2.2.1 NASM чувствителен к регистру символов*

Самым "простым" отличием является регистро-чувствительность NASM — он различает обращения к таким меткам, как foo, Foo или FOO. Если вы ассемблируете .obj-файлы для DOS или OS/2, то для перевода всех экспортируемых в другие модули символов в верхний регистр можете активизировать директиву UPPERCASE (описана в параграфе 6.2). Однако *в пределах* одного модуля NASM различает метки, отличающиеся друг от друга только регистром.

#### *2.2.2 NASM требует квадратные скобки для ссылок на память*

NASM был разработан, кроме всего прочего, для упрощения запоминания синтаксиса. Одна из целей проекта NASM состоит в том, чтобы везде, где это возможно, было взаимно однозначное соответствие между отдельной строкой кода NASM и генерируемой из нее инструкцией. В MASM вы этого сделать не можете: если определите, например,

```
foo     equ 1
bar     dw 2
```

затем напишете две строки кода

```
mov ax,foo
mov ax,bar
```

то будут сгенерированы две совершенно различных инструкции, несмотря на то, что синтаксис на вид совершенно идентичен.

NASM уходит от этой нежелательной ситуации путем упрощения синтаксиса для ссылок на память. Правило элементарно — *любой* доступ к содержимому памяти требует постановки квадратных скобок вокруг адреса, а при любом доступе к *адресу* переменной квадратные скобки не ставятся. Таким образом, инструкция вида mov ax,foo будет *всегда* ссылаться на константу времени компиляции, неважно EQU ли это или адрес переменной, в то же время для получения доступа к *содержимому* переменной bar вы должны использовать код mov ax,[bar].

Из этого также следует, что NASM не нуждается в ключевом слове MASMa OFFSET, т.к. MASMовский код mov ax,offset bar означает то же самое, что и mov ax,bar для NASM. Если вы накопили достаточно много кода, написанного под MASM и хотите использовать его в NASMe, вы всегда можете вставить строчку вида %idefine offset, указывающую препроцессору, что ключевое слово OFFSET является ничего не делающей инструкцией.

Данное несоответствие еще в большей степени присутствует в a86, где объявление метки с завершающим двоеточием описывает собственно метку, а без двоеточия — переменную. Так, в a86 инструкция mov ax,var будет вести себя по разному, в зависимости от объявления метки var: как var: dw 0 (это метка) или как var dw 0 (а это уже переменная размером в слово). NASM по сравнению с этим очень прост: *все* является метками.

NASM, в целях упрощения, не поддерживает гибридный синтаксис MASMa и его клонов, такой, как например mov ax,table[bx], где ссылка на память обозначена частью внутри квадратных скобок, а частью — вне их. Правильный синтаксис в NASM для указанной выше инструкции будет mov ax,[table+bx]. Соответственно, инструкция вида mov ax,es:[di] не поддерживается, правильная инструкция — mov ax,[es:di].

### 2.2.3 NASM не хранит типы переменных

NASM не запоминает определямые вами типы переменных. Поскольку MASM эти вещи запоминает, то при встрече var dw 0 он запомнит, что вы определили var как пременную размером в слово и затем будет способен разрешить неопределенность при появлении инструкции mov var,2. NASM же преднамеренно не будет помнить ничто относительно символа var за исключением того, где он начинается, поэтому вы должны явно указывать mov word [var],2.

В соответствии с этим, NASM не поддерживает инструкции LODS, MOVS, STOS, SCAS, CMPS, INS или OUTS, поддерживаются только их формы вида LODSB, MOVSW и SCASD, где явно задается размер компонентов обрабатываемой строки.

### 2.2.4 NASM не поддерживает ASSUME

Как часть общей идеологии упрощения, NASM не поддерживает директиву ASSUME. NASM не будет следить за тем, какие значения вы помещаете в сегментные регистры и поэтому никогда *автоматически не будет* генерировать префикс замены сегмента.

### 2.2.5 NASM не поддерживает модели памяти

NASM не имеет никаких директив для поддержки различных 16-битных моделей памяти. Программист должен самостоятельно следить, какие функции предполагается вызывать "дальним вызовом", а какие — ближним, и соответственно помещать правильную форму инструкции RET (RETN или RETF; NASM допускает применение RET в качестве альтернативной формы RETN); кроме того, программист ответственен за кодирование инструкций CALL FAR, где это необходимо при вызове внешних функций, и должен также следить, какие описания внешних переменных являются дальними, а какие — ближними.

### 2.2.6 Различия в обработке чисел с плавающей точкой

NASM, в отличие от MASM, использует другие имена для ссылок на регистры сопроцессора: MASM ссылается на эти регистры как ST(0), ST(1) и т.д., в NASMe для этой цели используются имена st0, st1 и т.д.

Начиная с версии 0.96, NASM обрабатывает инструкции форм "nowait" так же, как и MASM-совместимые ассемблеры. Особая обработка, использованная в версиях 0.95 и младше была основана на неправильном понимании авторами.

### 2.2.7 Прочие различия

По историческим причинам NASM использует ключевое слово TWORD там, где MASM и совместимые с ним ассемблеры используют TBYTE.

NASM объявляет резервируемое пространство (неинициализированные данные) не так, как MASM: там, где MASM-программист может написать stack db 64 dup (?), NASM требует следующее — stack resb 64, что интерпретируется как "резервирование 64 байт". Так как NASM обрабатывает ? как обычный символ, вы можете написать что-то вроде ? equ 0, а затем использовать dw ?, что будет возможно полезно для каких-то целей. DUP однако остается неподдерживаемым синтаксисом.

И, наконец, в дополнение ко всему вышесказанному, макросы и директивы NASM работают совершенно отлично от MASM. Подробности приведены в главе 4 и главе 5.

### 3.1 Обзор ассемблерной строки NASM

Как и в большинстве ассемблеров, каждая строка NASM содержит (если это не макрос, препроцессорная или ассемблерная директива – см. главу 4 и главу 5) комбинацию четырех полей:

```
метка: инструкция операнды ; комментарий
```

Как обычно, большинство этих полей необязательны; допускается присутствие или отсутствие любой комбинации метки, инструкции и комментария. Конечно, необходимость поля операндов определяется инструкцией процессора.

NASM не накладывает ограничений на количество пробелов в строке: метки могут иметь пробелы вначале, а инструкции могут не иметь никаких пробелов и т.п. Двоеточие после метки также необязательно. (Это означает, что если вы хотите поместить в строку инструкцию lodsb, а введете lodab, строка останется корректной, но вместо инструкции будет объявлена метка. Выявить данные опечатки отчасти можно, введя в строке запуска NASM ключ -w+orphan-labels — в этом случае при обнаружении метки без заключительного двоеточия будет выдаваться предупреждение).

Допустимыми символами в метках являются буквы, цифры, знаки _, $, #, @, ~, . и ?. Допустимые символы в начале метки (первый символ метки) — только буквы, точка (.) (со специальным значением, см. параграф 3.8), знак подчеркивания (_) и вопросительный знак (?). В идентификаторе может также присутствовать префикс $ для указания того, что это действительно идентификатор, а не зарезервированное слово; таким образом, если некоторый компонуемый вами модуль описывает символ eax, вы можете в коде NASM (для указания того, что это не регистр) сослаться на него так: $eax.

Поле инструкций может содержать любые процессорные инструкции: поддерживаются инструкции Pentium и P6, FPU, MMX, а также некоторые недокументированные инструкции. Перед инструкциями могут присутствовать префиксы LOCK, REP, REPE/REPZ или REPNE/REPNZ, используемые по их обычному предназначению. Поддерживаются префиксы размера адреса и операнда A16, A32, O16 и O32 — пример их использования приведен в главе 9. В качестве префикса инструкции вы можете использовать также обозначение сегментного регистра: код mov [bx],ax эквивалентен коду mov [es:bx],ax. Мы рекомендуем использовать последний синтаксис, т.к. он согласуется с другими синтаксическими особенностями языка, однако для инструкций, не имеющих операндов (например, LODSB) и требующих в некоторых случаях замены сегмента, на данный момент не существует никакого синтаксического способа обойти конструкцию es lodsb.

Префиксы, такие как CS, A32, LOCK или REPE могут присутствовать в строке самостоятельно и при этом NASM будет генерировать соответствующие префикс-байты.

В дополнение к инструкциям процессора, NASM поддерживает также несколько псевдо-инструкций, описанных в параграфе 3.2.

Операнды инструкций могут принимать несколько форм: они могут быть регистрами (например ax, bp, ebx, cr0: NASM не использует синтаксис стиля a-ля-gas, где имена регистров должны предваряться знаком %), эффективными адресами (см. параграф 3.3), константами (параграф 3.4) или выражениями (параграф 3.5).

Для инструкций сопроцессора NASM допускает различные формы синтаксиса: вы можете использовать двух-операндную форму, поддерживаемую MASMом, а также чисто NASMовскую одно-операндную форму. Подробности о форме каждой поддерживаемой инструкции приведены в приложении A. Например, вы можете написать:

```
fadd st1                ; это значит st0 := st0 + st1
fadd st0,st1            ; это то же самое

fadd st1,st0            ; это значит st1 := st1 + st0
fadd to st1             ; это то же самое
```

Почти любая инструкция сопроцессора, ссылающаяся на содержимое памяти, должна использовать один из префиксов DWORD, QWORD или TWORD для указания на то, операнд какого размера должен участвовать в команде.

### 3.2 Псевдо-инструкции

Псевдо-инструкции не являются реальными инструкциями x86 процессора, но все равно помещаются в поле инструкций, т.к. это наиболее подходящее место для них. Текущими псевдо-инструкциями являются DB, DW, DD, DQ и DT, их копии для работы с неинициализированной памятью RESB, RESW, RESD, RESQ и REST, команды INCBIN, EQU и префикс TIMES.

#### *3.2.1 DB и ее друзья: Объявление инициализированных данных*

Как и в MASM, DB, DW, DD, DQ и DT используются для объявления инициализированных данных в выходном файле. Они могут использоваться достаточно многими способами:

```
db 0x55                ; просто байт 0x55
db 0x55,0x56,0x57      ; последовательно 3 байта
db 'a',0x55            ; символьная константа
db 'hello',13,10,'$'   ; это строковая константа
dw 0x1234              ; 0x34 0x12
dw 'a'                 ; 0x41 0x00 (это просто число)
dw 'ab'                ; 0x41 0x42 (символьная константа)
dw 'abc'               ; 0x41 0x42 0x43 0x00 (строка)
dd 0x12345678          ; 0x78 0x56 0x34 0x12
dd 1.234567e20         ; константа с плавающей точкой
dq 1.234567e20         ; двойной точности
dt 1.234567e20         ; расширенной точности
```

DQ и DT не допускают в качестве операндов числовые или строковые константы.

#### *3.2.2 RESB и ее друзья: Объявление неинициализированных данных*

RESB, RESW, RESD, RESQ и REST разработаны для использования в BSS-секции модуля: они объявляют *не инициализированное* пространство для хранения данных. Каждая принимает один операнд, являющийся числом резервируемых байт, слов, двойных слов и т.д. Как было указано в параграфе 2.2.7, NASM не поддерживает синтаксис резервирования неинициализированного пространства, реализованный в MASM/TASM, где можно делать DW ? или подобные вещи: это заменено полностью. Операнд псевдо-инструкций класса RESB является *критическим выражением*: см. параграф 3.7.

Например:

```
buffer:   resb 64              ; резервирование 64 байт
wordvar:  resw 1               ; резервирование слова
realarray resq 10              ; массив из 10 чисел с плавающей точкой
```

#### *3.2.3 INCBIN: Включение внешних бинарных файлов*

INCBIN заимствована из старого ассемблера DevPac, работавшего на Amige: она включает бинарный файл в выходной файл, оставляя его (бинарный файл) неизменным. Это может быть полезно (например) для включения картинок и музыки непосредственно исполняемый файл игрушки. Эта псевдо-инструкция может быть вызвана тремя разными способами:

```
incbin "file.dat"      ; включение файла целиком
incbin "file.dat",1024 ; пропуск первых 1024 байт
incbin "file.dat",1024,512 ; пропуск первых 1024 и
                       ; включение следующих 512 байт
```

### 3.2.4 EQU: Определение констант

EQU вводит символ для указанного константного значения: если используется EQU, в этой строке кода должна присутствовать метка. Смысл EQU — связать имя метки со значением ее (только) операнда. Данное определение абсолютно и не может быть позднее изменено. Например,

```
message   db 'Привет, фуфел!'
msglen    equ $-message
```

определяет msglen как константу 12. msglen не может быть позднее переопределено. Это не определение препроцессора: значение msglen обрабатывается здесь только *один раз* при помощи значения $ (что такое $ – см. параграф 3.5) в месте определения. Имейте в виду, что операнд EQU также является критическим выражением (параграф 3.7).

### 3.2.5 TIMES: Повторение инструкций или данных

Префикс TIMES заставляет инструкцию ассемблироваться несколько раз. Данная псевдо-инструкция отчасти представляет NASM-эквивалент синтаксиса DUP, поддерживающегося MASM-совместимыми ассемблерами. Вы можете написать, например

```
zerobuf: times 64 db 0
```

или что-то подобное; однако TIMES более разносторонняя инструкция. Аргумент TIMES — не просто числовая константа, а *числовое выражение*, поэтому вы можете писать следующие вещи:

```
buffer:   db 'Привет, фуфел!'
     times 64-$+buffer db ' '
```

При этом будет резервироваться строго определенное пространство, начиная от метки buffer и длиной 64 байта. Наконец, TIMES может использоваться в обычных инструкциях, так что вы можете писать тривиальные развернутые циклы:

```
times 100 movsb
```

Заметим, что нет никакой принципиальной разницы между times 100 resb 1 и resb 100 за исключением того, что последняя инструкция будет обрабатываться примерно в 100 раз быстрее из-за внутренней структуры ассемблера.

Операнд псевдо-инструкции TIMES, подобно EQU и RESB, является критическим выражением (параграф 3.7).

Имейте также в виду, что TIMES не применима в макросах: причиной служит то, что TIMES обрабатывается после макро-фазы, позволяющей аргументу TIMES содержать выражение, подобное 64-$+buffer. Для повторения более одной строки кода или в сложных макросах используйте директиву препроцессора %rep.

## 3.3 Эффективные адреса

Эффективный адрес — это любой операнд инструкции со ссылкой на память. Эффективные адреса в NASM имеют очень простой синтаксис: они содержат выражение (в результате вычислений которого получается нужный адрес), обрамленное квадратными скобками. Например:

```
wordvar   dw 123
     mov ax,[wordvar]
     mov ax,[wordvar+1]
     mov ax,[es:wordvar+bx]
```

Любая другая ссылка, не соответствующая этой простой системе, для NASM недействительна, например es:wordvar[bx].

Более сложные эффективные адреса, когда вовлечено более одного регистра, работают точно также:

```
mov eax,[ebx*2+ecx+offset]
mov ax,[bp+di+8]
```

NASM способен воспринимать алгебру таких выражений, поэтому он правильно транслирует вещи, выглядящие *на первый взгляд* недопустимыми:

```
mov eax,[ebx*5]        ; ассемблируется как [ebx*4+ebx]
mov eax,[label1*2-label2] ; то есть [label1+(label1-label2)]
```

Некоторые варианты эффективных адресов имеют более одной ассемблерной формы; в большинстве таких ситуаций NASM будет генерировать самую короткую из них. Например, у нас имеются простые ассемблерные инструкции [eax*2+0] и [eax+eax]. NASM будет генерировать последнюю из них, т.к. первый вариант требует дополнительно 4 байта для хранения нулевого смещения.

NASM имеет механизм подсказок, позволяющий создавать из [eax+ebx] и [ebx+eax] разные инструкции; это порой полезно, т.к. например [esi+ebp] и [ebp+esi] по умолчанию имеют разные сегментные регистры.

Несмотря на это, вы можете заставить NASM генерировать требуемые формы эффективных адресов при помощи ключевых слов BYTE, WORD, DWORD и NOSPLIT. Если вам нужно, чтобы [eax+3] ассемблировалась со смещением в двойное слово, вместо одного байта по умолчанию, вы можете написать [dword eax+3]. Точно также при помощи [byte eax+offset] вы можете заставить NASM использовать байтовые смещения для небольших значений, не определяемых при первом проходе (см. пример такого кода в параграфе 3.7). В особых случаях, [byte eax] будет кодироваться как [eax+0] с нулевым байтовым смещением, а [dword eax] будет кодироваться с нулевым смещением в двойное слово. Обычная форма, [eax], будет оставлена без смещения.

NASM будет разделять [eax*2] на [eax+eax], т.к. это позволяет избежать использования поля смещения и сэкономить некоторое пространство; соответственно, [eax*2+offset] будет разделено на [eax+eax+offset]. При помощи ключевого слова NOSPLIT вы можете запретить такое поведение NASM: [nosplit eax*2] будет буквально оттранслировано в [eax*2+0].

## 3.4 Константы

NASM знает четыре различных типа констант: числовые, символьные, строковые и с плавающей точкой.

### *3.4.1 Числовые константы*

Числовая константа — это просто число. NASM позволят определять числа в различных системах счисления и различными способами: вы можете использовать суффиксы H, Q и B для шестнадцатеричных, восьмеричных и двоичных чисел соответственно; можете использовать для шестнадцатеричных чисел префикс 0x в стиле C, а также префикс $ в стиле Borland Pascal. Однако имейте в виду, что префикс $ может быть также префиксом идентификаторов (см. параграф 3.1), поэтому первой цифрой шестнадцатеричного числа при использовании этого префикса должна быть обязательно цифра, а не буква.

Некоторые примеры числовых констант:

```
mov ax,100              ; десятичная
mov ax,0a2h             ; шестнадцатеричная
mov ax,$0a2             ; снова hex: нужен 0
mov ax,0xa2             ; опять hex
mov ax,777q             ; восьмеричная
mov ax,10010011b        ; двоичная
```

### *3.4.2 Символьные константы*

Символьная константа содержит от одного до четырех символов, заключенных в одиночные или двойные кавычки. Тип кавычек для NASM несущественен, поэтому если используются одинарные кавычки, двойные могут выступать в роли символа и, соответственно, наоборот.

Символьная константа, содержащая более одного символа, будет загружаться в обратном порядке следования байт: если вы пишете

```
mov eax,'abcd'
```

сгенерированной константой будет не 0x61626364, а 0x64636261, поэтому если сохранить эту константу в память, а затем прочитать, получится снова abcd, но никак не dcba. Это также влияет на инструкцию CPUID Пентиумов (см. section A.29).

### *3.4.3 Строковые константы*

Строковые константы допустимы только в некоторых псевдо-инструкциях, а именно в семействе DB и инструкции INCBIN. Строковые константы похожи на символьные, только длиннее. Они обрабатываются как сцепленные друг с другом символьные константы. Так, например, следующие строки кода эквивалентны.

```
db 'hello'              ; строковая константа
db 'h','e','l','l','o' ; эквивалент из символьных констант
```

Следующие строки также эквивалентны:

```
dd 'ninechars'          ; строковая константа в двойное слово
dd 'nine','char','s'    ; три двойных слова
db 'ninechars',0,0,0    ; и действительно похоже
```

Обратите внимание, что когда используется db, константа типа 'ab' обрабатывается как строковая, хотя и достаточно коротка, чтобы быть символьной, потому что иначе db 'ab' имело бы тот же смысл, какой и db 'a', что глупо. Соответственно, трех- или четырехсимвольные константы, являющиеся операндами инструкции dw, обрабатываются также как строки.

### *3.4.4 Константы с плавающей точкой*

Константы с плавающей точкой допустимы только в качестве аргументов DD, DQ и DT. Выражаются они традиционно: цифры, затем точка, затем возможно цифры после точки, и наконец, необязательная E с последующей степенью. Точка обязательна, т.к. dd 1 NASM воспримет как объявление целой константы, в то время как dd 1.0 будет воспринята им правильно.

Несколько примеров:

```
dd 1.2                  ; "простое" число
dq 1.e10                ; 10,000,000,000
dq 1.e+10               ; синоним 1.e10
dq 1.e-10               ; 0.000 000 000 1
dt 3.1415926535897932384626 ; число pi
```

В процессе компиляции NASM не может проводить вычисления над константами с плавающей точкой (это сделано с целью переносимости). Несмотря на то, что NASM генерирует код для x86 процессоров, сам по себе ассемблер может работать на любой системе с ANCI C компилятором. Само собой, ассемблер не может гарантировать присутствия устройства, обрабатывающего числа с плавающей точкой в формате Intel, поэтому стало бы необходимо включить собственный полный набор подпрограмм для работы с такими числами, что неизбежно привело бы к значительному увеличению размера самого ассемблера, хотя польза от этого была бы минимальна.

## 3.5 Выражения

Синтаксис выражений NASM подобен синтаксису выражений языка C.

NASM не гарантирует размер целых чисел, используемых для вычисления выражений при компиляции: с тех пор как NASM может вполне успешно компилировать и выполняться на 64-разрядных платформах, не будьте так уверены, что выражения вычисляются в 32-битных регистрах и что можно попробовать

умышленно сделать переполнение. Это сработает не всегда. NASM гарантирует только то, что и ANSI C: вы всегда имеете дело *как минимум* с 32-битными регистрами.

В выражениях NASM поддерживает два специальных символа, позволяющих при вычислениях выражений получать текущую позицию (смещение) ассемблирования: это знаки $ и $$. Знак $ вычисляет позицию начала строки, содержащей выражение, т.е. вы можете сделать бесконечный цикл при помощи команды JMP $. Знак $$ определяет начало текущей секции (сегмента), поэтому вы можете узнать, как далеко находитесь от начала секции при помощи выражения ($-$$).

Ниже перечислены арифметические операции NASM в порядке возрастания приоритета.

### *3.5.1 |: Побитовый оператор ИЛИ*

Оператор | производит побитовую операцию ИЛИ, соответствующую процессорной инструкции OR. Побитовое ИЛИ имеет самый низкий приоритет среди арифметических операторов, поддерживаемых NASMом.

### *3.5.2 ^: Побитовый оператор ИСКЛЮЧАЮЩЕЕ ИЛИ*

Оператор ^ обеспечивает выполнение побитовой операции ИСКЛЮЧАЮЩЕЕ ИЛИ.

### *3.5.3 &: Побитовый оператор И*

Оператор & обеспечивает выполнение побитовой операции И.

### *3.5.4 << и >>: Операторы сдвига бит*

<< производит сдвиг бит влево точно так, как это делается в C. Так, 5<<3 обрабатывается как 5 умножить на 8, или 40. >> производит сдвиг бит вправо; в NASM этот сдвиг всегда *беззнаковый*, поэтому биты, освобождаемые слева в результате сдвига, заполняются нулями, а не старшим знаковым разрядом.

### *3.5.5 + и —: Операторы сложения и вычитания*

Операторы + и — выполняют обычное сложение и вычитание.

### *3.5.6 *, /, //, % и %%: Умножение и деление*

* является оператором умножения. Операторы / и // обозначают деление: / соответствует беззнаковому делению, а // — знаковому. Подобно этому, операторы % и %% обеспечивают соответственно беззнаковое и знаковое получение остатка от деления (взятие по модулю).

NASM, также как и ANSI C, не дает никаких гарантий о физическом смысле знакового оператора взятия по модулю.

Так как символ % часто используется макропроцессором, будьте внимательны при применении знакового и беззнакого операторов взятия по модулю — они должны отделяться от других символов строки по крайней мере одним пробелом.

### *3.5.7 Унарные операторы: +, -, ~ и SEG*

Наивысший приоритет в грамматике выражений NASM имеют операторы, применяемые к одному аргументу: оператор "минус" (—) изменяет знак своего операнда, оператор "плюс" (+) ничего не делает (введен для симметричности с минусом), оператор "тильда" (~) вычисляет дополнение операнда, а оператор SEG извлекает сегментный адрес операнда (более подробно описывается в параграфе 3.6).

## 3.6 SEG и WRT

При написании больших 16-битных программ, которые должны быть разделены на несколько сегментов, часто необходимо получить сегментную часть адреса некоторого символа. Для выполнения этой функции в NASM имеется оператор SEG.

Оператор SEG возвращает базу *предопределенного* сегмента символа, относительно которой вычисляется смещение последнего. Так следующий код

```
mov ax,seg symbol
mov es,ax
mov bx,symbol
```

будет загружать в пару ES:BX корректный указатель на символ symbol.

Бывают и более сложные случаи: т.к. 16-битные сегменты и группы способны перекрываться, вы возможно захотите иногда сослаться на некоторый символ при помощи базы сегмента, отличного от предопределенного. NASM позволяет это сделать при помощи ключевого слова WRT (With Reference To). Например, код

```
mov ax,weird_seg       ; weird_seg является базой сегмента
mov es,ax
mov bx,symbol wrt weird_seg
```

загрузит в ES:BX другой, но функционально эквивалентный указатель на символ symbol.

NASM поддерживает дальние (межсегментные) вызовы подпрограмм и передачи управления при помощи синтаксиса call segment:offset, где segment и offset являются непосредственными значениями, поэтому для вызова дальней процедуры вы можете использовать следующий синтаксис:

```
call (seg procedure):procedure
call weird_seg:(procedure wrt weird_seg)
```

(Круглые скобки включены для большей ясности приведенных инструкций. На практике они не нужны).

NASM также поддерживает синтаксис call far, являющийся аналогом первой из выше приведенных инструкций. В этих примерах инструкция JMP будет работать также, как CALL.

Для объявления дальнего указателя на сегмент данных, вы можете писать:

```
dw symbol, seg symbol
```

NASM не поддерживает более удобных аналогов этому объявлению, однако при помощи макропроцессора вы всегда можете их придумать.

### 3.7 Критические выражения

В отличие от TASM и других, NASM является двухпроходным ассемблером; он всегда делает *только* два прохода. Из-за этого он не способен "справиться" со сложными исходными файлами, требующими три и более проходов.

Первый проход используется для определения размера всех ассемблируемых инструкций и данных, поэтому на втором проходе (где генерируется код) известны адреса всех символов, на которые имеются ссылки. Таким образом, NASM не сможет обработать код, в котором размер зависит от значения символа, объявленного позднее, например:

```
          times (label-$) db 0
label:    db 'Где это я?'
```

Аргумент TIMES в этом случае должен точно рассчитываться для всех меток; NASM воспримет этот пример ошибочным, т.к. он не сможет узнать размер строки с TIMES. Для него это будет то же, что и заведомо ошибочный код

```
          times (label-$+1) db 0
label:    db 'А теперь я где?'
```

где *любое* значение аргумента TIMES по определению неверно!

NASM отклоняет такой код при помощи концепции т.н. *критического выражения*, определяемого как выражение, значение которого должно быть рассчитано на первом проходе и которое, следовательно, должно зависеть только от символов, описанных перед ним. Аргумент префикса TIMES является критическим выражением; по некоторым причинам аргументы псевдо-инструкций семейства RESB также являются критическими выражениями.

Критическое выражение может неожиданно возникнуть в следующем контексте:

```
          mov ax,symbol1
symbol1   equ symbol2
symbol2:
```

На первом проходе NASM не может определить значение symbol1, т.к. он объявлен равным symbol2, который, в свою очередь, NASM еще "не видит". Соответственно на втором проходе, при обработке строки mov ax,symbol1 он не способен сгенерировать правильный код, потому что значение symbol1 остается неизвестным. На следующей строке, увидев EQU, NASM сможет определить значение symbol1, однако будет уже поздно.

NASM предотвращает возникновение данных проблем, вводя для критических выражений правосторонний оператор EQU, при котором объявление symbol1 будет отбраковано на первом проходе.

Еще одна похожая проблема, связанная с опережающими ссылками: рассмотрите следующий фрагмент кода.

```
          mov eax,[ebx+offset]
offset    equ 10
```

На первом проходе NASM должен вычислить длину инструкции mov eax,[ebx+offset], не зная значение offset. Он никак не сможет узнать, что смещение offset представляет собой малую величину, вписывающуюся в однобайтное поле смещения и что можно "безбоязненно" сгенерировать более короткую форму эффективного адреса. Однако на первом проходе еще не известно, что такое offset — это может быть символ в сегменте кода и для него возможно нужна полная четырехбайтовая форма инструкции. Таким образом, размер инструкции рассчитывается исходя из четырехбайтовой адресной части. Сделав это предположение, на втором проходе NASM вынужден оставлять длину инструкции как есть, генерируя при этом не совсем оптимальный код. Данная проблема может быть разрешена путем объявления offset *перед* ее первым использованием или явным указанием на байтовый размер смещения: [byte ebx+offset].

### 3.8 Локальные метки

NASM дает специальную трактовку символов, начинающихся с точки. Метка, начинающаяся с точки, обрабатывается как *локальная*. Это означает, что она неразрывно связана с предыдущей нелокальной меткой. Например:

```
label1    ; некоторый код
.loop     ; еще какой-то  код
     jne .loop
     ret
label2    ; некоторый код
.loop     ; еще какой-то  код
     jne .loop
     ret
```

В приведенном фрагменте каждая инструкция JNE переходит на строку непосредственно перед ней, т.к. два определения .loop остаются разделены в силу того, что каждое связано с предшествующей нелокальной меткой.

Данный способ обработки локальных меток позаимствован из ассемблера DevPac (Amiga); однако NASM делает шаг вперед — он позволяет обращаться к локальным меткам из другой части кода. Это достигается путем описания локальной метки на основе предыдущей нелокальной. Описания .loop в примере выше в действительности описывают два разных символа: label1.loop и label2.loop, поэтому если вам это действительно надо, то можете написать:

```
label3    ; некоторый код
     ; и т.д.
     jmp label1.loop
```

Иногда бывает полезно, например, в макросах — определить метку, на которую можно ссылаться отовсюду, но которая не пересекается с обычным механизмом локальных меток. Такая метка не может быть нелокальной, так как существует последующее описание и ссылки на локальные метки; она также не может быть и локальной, вследствие того, что описывающий ее макрос не будет знать полное имя метки. Для разрешения этой проблемы в NASM введен третий тип меток, которые обычно используются только в описаниях макросов: если метка начинается со специального префикса ..@, она ничего не делает по отношению к механизму локальных меток. Таким образом, вы можете написать:

```
label1:   ; нелокальная метка
.local:   ; это label1.local
..@foo:   ; это специальный символ
label2:   ; другая нелокальная метка
.local:   ; это label2.local
     jmp ..@foo             ; переход на три строки вверх
```

NASM имеет возможность определять другие специальные символы, начинающиеся с двух точек: например, ..start используется для указания точки входа в объектном формате obj (см. параграф 6.2.6).

## *Глава 4: Препроцессор NASM*

Перевод: AsmOS group, © 2001

NASM содержит мощный макропроцессор, поддерживающий условное ассемблирование, многоуровневое включение файлов, две формы макро-определений (однострочные и многострочные) и механизм "контекстного стека", расширяющий возможности макро-определений (далее – макросы). Все директивы препроцессора начинаются со знака %.

### 4.1 Однострочные макросы

#### *4.1.1 Обычный способ: %define*

Однострочные макросы описываются при помощи директивы препроцессора %define. Определения работают подобно языку C; т.е. вы можете сделать что-то наподобие

```
%define ctrl 0x1F &
%define param(a,b) ((a)+(a)*(b))
          mov byte [param(2,ebx)], ctrl 'D'
```

что будет развернуто в

```
          mov byte [(2)+(2)*(ebx)], 0x1F & 'D'
```

Если однострочный макрос содержит символы, вызывающие другой макрос, то развертывание первого осуществляется в процессе вызова, а не при определении. Так, код

```
%define a(x) 1+b(x)
%define b(x) 2*x
          mov ax,a(8)
```

будет обработан как и ожидается — mov ax,1+2*8, несмотря на то, что при описании a макрос b еще не определен.

Макросы, описываемые конструкцией %define, регистрочувствительны: после %define foo bar только foo будет развернуто в bar, но никак не Foo или FOO. При использовании вместо %define конструкции %idefine ('i' от слова 'insensitive') вы можете сразу описать все варианты комбинации строчных и прописных букв в имени макроса, поэтому %idefine foo bar будет развертывать в bar не только foo, но и Foo, FOO, fOO и т.п.

Существует механизм, следящий за рекурсивным развертыванием макросов, предотвращающий циклические ссылки и бесконечные циклы. Когда это случается, препроцессор будет развертывать только первое вхождение макроса. Так, в коде

```
%define a(x) 1+a(x)
          mov ax,a(3)
```

макрос a(3) будет развернут единожды, т.е. в конструкцию 1+a(3) и дальнейшее развертывание производиться не будет. Данная возможность может быть полезна: в параграфе 8.1 имеется соответствующий пример.

Вы можете перегружать однострочные макросы: если вы напишете

```
%define foo(x) 1+x
%define foo(x,y) 1+x*y
```

препроцессор, подсчитав передаваемые вами параметры, корректно обработает оба типа вызова макроса: foo(3) будет развернуто в 1+3, в то время как foo(ebx,2) — в 1+ebx*2. В то же время, если вы напишете

```
%define foo bar
```

последующие описания foo будут запрещены: макрос без параметров не допускает описание макроса с тем же именем, но с *параметрами*, и наоборот.

Несмотря на это, *переопределение* однострочных макросов не запрещается: вы можете легко описать макрос как

```
%define foo bar
```

и затем, в том же самом исходном файле, переопределить его как

```
%define foo baz
```

Когда макрос foo будет вызван, он развернется в соответствии с самым поздним своим описанием. Это полезно в основном при описании однострочных макросов с конструкцией %assign (см. параграф 4.1.4).

Вы можете переопределить однострочный макрос при помощи ключа '-d' командной строки NASM: см. параграф 2.1.8.

При описании однострочных макросов вы можете объединять строки при помощи псевдо-оператора %+. Например:

```
%define _myfunc _otherfunc
%define cextern(x) _ %+ x
cextern (myfunc)
```

После первого развертывания третья строка примет вид "_myfunc". При дальнейшей обработке препроцессор развернет эту строку в "_otherunc".

### *4.1.2 Однострочные макросы раннего связыванияs: %xdefine*

Механизм %define, описанный в предыдущем параграфе, обеспечивает описание однострочных макросов с *поздним связыванием*, при котором ссылки на другие макросы разворачиваются при их вызове, поэтому если вы в это время измените внутренний макрос, значение развернутого макроса соответственно изменится. Это свойство полезно, но в некоторых случаях нежелательно. Например:

```
%assign ofs 0
%macro arg 1
        %xdefine %1 dword [esp+ofs]
        %assign ofs ofs+4
%endmacro
        arg  a
        arg  b
```

Если в этом примере вместо %xdefine мы используем %define, оба макроса будут развернуты в одно и то же значение dword [esp+8], что неверно. При использовании конструкции %xdefine макрос разворачивается во время его определения, поэтому a будет развернуто в dword [esp+0], а b — в dword [esp+4].

Макрос %xdefine имеет нечувствительный к регистру эквивалент %ixdefine, работающий так же, как и %idefine по отношению к %define.

### *4.1.3 Отмена определения макроса: %undef*

Команда %undef удаляет однострочные макросы. Например, следующая последовательность:

```
%define foo bar
%undef foo
        mov eax, foo
```

будет развернута в инструкцию mov eax, foo, так как после %undef макрос foo больше не определен.

Отмена определения предопределенных макросов может быть осуществлена при помощи ключа '-u' командной строки NASM: см. параграф 2.1.9.

### *4.1.4 Переменные препроцессора: %assign*

Альтернативным способом определения однострочных макросов является использование директивы %assign (и ее нечувствительного к регистру эквивалента %iassign, отличающегося от %assign тем же самым, чем %idefine отличается от %define).

Данная директива используется для определения однострочного макроса без параметров, но включающего числовое значение. Это значение может быть задано в форме выражения и обрабатывается оно только один раз — при обработке директивы %assign.

Как и в случае %define, макрос, определенный при помощи %assign, может быть позднее переопределен, например следующая строка

```
%assign i i+1
```

увеличивает числовое значение макроса.

%assign полезна для контроля завершения препроцессорных циклов %rep: пример этого см. в параграфе 4.4. Другие примеры применения %assign можно найти в параграфе 7.4 и парарафе 8.1.

Выражение, передаваемое %assign, является критическим (см. параграф 3.7) и должно на выходе давать просто число (не включая в себя различные перемещающиеся ссылки наподобие адресов кода и данных).

## 4.2 Многострочные макросы: %macro

Многострочные макросы в большинстве своем похожи на макросы в MASM и TASM: определение многострочного макроса в NASM выглядит похоже.

```
%macro prologue 1
          push ebp
          mov ebp,esp
          sub esp,%1
%endmacro
```

Здесь определяется как макрос C-подобная функция prologue: теперь вы можете вызвать макрос следующим образом:

```
myfunc:   prologue 12
```

что будет развернуто в три строки кода

```
myfunc:   push ebp
          mov ebp,esp
          sub esp,12
```

Число 1 после имени макроса в строке %macro определяет число параметров, которые ожидает получить макрос prologue. Конструкция %1 внутри тела макроса ссылается на первый передаваемый параметр. В случае макросов, принимающих более одного параметра, ссылка на них осуществляется как %2,%3 и т.д.

Многострочные макросы, как и однострочные, чувствительны к регистру символов, за исключением случая, когда вы примените альтернативную директиву %imacro.

Если вам требуется передать в многострочный макрос запятую в качестве составной части параметра, вы можете сделать это, заключив параметр в фигурные скобки. Например:

```
%macro silly 2
%2:       db %1
%endmacro
          silly 'a', letter_a    ; letter_a:  db 'a'
          silly 'ab', string_ab  ; string_ab: db 'ab'
          silly {13,10}, crlf    ; crlf:      db 13,10
```

### *4.2.1 Перегрузка многострочных макросов*

Многострочные макросы, как и однострочные, могут быть перегружены путем определения одного и того же имени несколько раз с разным числом параметров. В отличие от однострочных макросов, здесь не делается исключений даже для макросов без параметров. Так, вы можете написать

```
%macro prologue 0
          push ebp
          mov ebp,esp
%endmacro
```

для определения альтернативной формы функции-пролога, не выделяющей место в стеке.

Иногда вам может захотеться "перегрузить" процессорные инструкции; например, если вы определите

```
%macro push 2
          push %1
          push %2
%endmacro
```

позже вы сможете написать

```
          push ebx               ; эта строка не вызов макроса!
```

```
        push eax,ecx            ; а это - вызов
```

По умолчанию NASM будет выдавать предупреждение при обработке первой из этих строк, так как push теперь определена как макрос и вызывается с числом параметров, определения для которого нет. При этом будет сгенерирован корректный код, однако ассемблер даст предупреждение. Данное предупреждение можно отключить при помощи ключа командной строки -w-macro-params (см. параграф 2.1.12).

### *4.2.2 Локальные метки в макросах*

NASM позволяет внутри определения многострочного макроса описать метки, которые останутся локальными для каждого вызова макроса: при многократном вызове макроса имена меток каждый раз будут изменяться. Описание такой метки осуществляется при помощи префикса %%. Например, вы можете описать инструкцию, выполняющую RET если флаг Z установлен, следующим образом:

```
%macro retz 0
        jnz %%skip
        ret
%%skip:
%endmacro
```

После этого вы можете вызывать этот макрос столько раз, сколько нужно, и при этом в каждом вызове вместо метки %%skip NASM будет подставлять разные "реальные" имена. NASM создает имена в форме ..@2345.skip, где число 2345 изменяется при каждом вызове макроса. Префикс ..@ предотвращает пересечение имен локальных меток макросов с механизмом обычных локальных меток, описанным в параграфе 3.8. В связи с этим вам нужно избегать определения собственных меток в такой форме (префикс ..@, затем число, затем точка), иначе они могут совпасть с локальными метками макросов.

### *4.2.3 Поглощающие параметры макросов*

Иногда полезно определить макрос, собирающий всю командную строку в один параметр после извлечения одного или двух небольших параметров из начала очереди. Примером здесь может служить макрос, записывающий строку в файл MS-DOS, где вы можете захотеть написать что-то вроде

```
        writefile [filehandle],"Привет, фуфел!",13,10
```

NASM позволяет определить последний параметр макроса в качестве поглощающего. Это означает, что если вы вызовете макрос с большим числом параметров, чем ожидалось, все "лишние" параметры вместе с разделительными запятыми присоединятся к последнему ожидаемому параметру. Так, если вы напишете:

```
%macro writefile 2+
        jmp %%endstr
%%str:  db %2
%%endstr: mov dx,%%str
        mov cx,%%endstr-%%str
        mov bx,%1
        mov ah,0x40
        int 0x21
%endmacro
```

то приведенный выше пример вызова макроса writefile будет работать как нужно: текст перед первой запятой [filehandle] используется в качестве первого параметра и развернется, когда встретится ссылка %1, весь последующий текст объединится в %2 и расположится после db.

Поглощающая природа макроса указывается в NASM при помощи знака (+) после количества параметров в строке %macro.

При определении поглощающего макроса вы тем самым говорите NASM, как он должен разворачивать *любое* число параметров свыше явно указанного; в приведенном случае, например, он будет знать, что должен делать при вызове writefile с 2,3,4 или большим числом параметров. NASM также будет это

учитывать при перегрузке макросов и не позволит вам определить другую форму writefile, принимающую к примеру 4 параметра.

Естественно, приведенный выше макрос может быть реализован и обычным образом (не как поглощающий). В этом случае его вызов должен выглядеть так:

```
writefile [filehandle], {"Привет, фуфел!",13,10}
```

NASM поддерживает оба механизма помещения запятых в параметры макроса и вам самим выбирать, какой предпочтительнее в каждом конкретном случае.

В параграфе 5.2.1 приведен более "продвинутый" способ написания рассмотренного макроса

### *4.2.4 Параметры макросов по умолчанию*

NASM позволяет также определять многострочный макрос с указанием диапазона допустимого количества параметров. Если вы используете эту возможность, то можете задать значения по умолчанию для пропускаемых параметров. Например:

```
%macro die 0-1 "Полный кабздец твоей проге, фуфел!"
        writefile 2,%1
        mov ax,0x4c01
        int 0x21
%endmacro
```

Данный макрос (использующий макрос writefile, описанный в параграфе 4.2.3) может быть вызван как с явно указанным сообщением об ошибке, помещаемым в выходной поток перед закрытием, так и без параметров. В последнем случае в качестве параметра будет подставлено сообщение по умолчанию, введенное при определении макроса.

Обычно для макроса данного типа вы задаете минимальное и максимальное число параметров; минимальное число — это параметры, требующиеся при вызове макроса, а для остальных вы задаете значения по умолчанию. Так, если определение макроса начинается со строки

```
%macro foobar 1-3 eax,[ebx+2]
```

то он может быть вызван с числом параметров от одного до трех; при этом параметр %1 всегда берется из строки вызова макроса, параметр %2 (если не задан) примет значение eax, а параметр %3 (если не задан) — значение [ebx+2].

Вы не обязаны задавать значения по умолчанию при определении макроса — в этом случае они будут пустые. Это может быть полезно для макросов, принимающих различное число параметров, так как число реально передаваемых параметров вы можете указать при помощи конструкции %0 (см. параграф 4.2.5 ниже).

Механизм параметров по умолчанию может комбинироваться с механизмом поглощающих параметров; например, описанный выше макрос die может быть сделан более продвинутым и полезным путем изменения первой строки определения:

```
%macro die 0-1+ "Полный кабздец твоей проге, фуфел!",13,10
```

Максимальное число параметров может быть неограниченным, что обозначается как *. В этом случае, конечно невозможно предусмотреть полный набор параметров по умолчанию. Примеры такого типа макросов см. в параграфе 4.2.6.

### 4.2.5 %0: Счетчик макро-параметров

Параметр %0 возвращает числовую константу, представляющую собой количество передаваемых в макрос параметров. Он может использоваться как аргумент для %rep (см. параграф 4.4) с целью перебора всех параметров макроса. Примеры см. в параграфе 4.2.6 ниже.

### 4.2.6 %rotate: "Вращение" параметров макросов

Unix-программисты знакомы с командой shift оболочки, позволяющей "сдвигать" переданные шелл-скриптом аргументы ($1, $2 и т.д.) так, что аргумент, имевший до этого ссылку $2, получает ссылку $1, а имевший ссылку $1, становится недоступен.

NASM обеспечивает подобный механизм при помощи %rotate. Как видно по имени, эта команда отличается от команды Unix тем, что параметры не теряются: "выталкиваемые" с левого конца списка аргументов, они появляются на правом, и наоборот.

%rotate вызывается с одним числовым аргументом (который может быть выражением). Параметры макроса "вращаются" влево на количество мест, указанных аргументом. Если аргумент отрицателен, параметры вращаются вправо.

Например, пара макросов для сохранения и восстановления набора регистров может работать так:

```
%macro multipush 1-*
%rep %0
          push %1
%rotate 1
%endrep
%endmacro
```

Этот макрос вызывает инструкцию PUSH для каждого аргумента, слева-направо. Начинается это дело с сохранения первого аргумента, %1, затем вызывается %rotate для перемещения всех аргументов на один шаг влево, так что изначально второй аргумент становится доступен как %1. Повторение данной процедуры для всего набора аргументов (достигается это передачей %0 как аргумента для %rep) позволяет сохранить каждый из них.

Обратите внимание на использование звездочки (*) в качестве максимального числа параметров. Это означает, что число передаваемых макросу multipush параметров не ограничено.

Удобно иметь под рукой и обратный макрос, извлекающий регистры из стека, особенно если он не требует передавать ему аргументы в обратном порядке относительно PUSH. В идеале вы будете помещать в текст программы вызов макроса multipush, затем копировать эту строку, вставлять ее туда, где требуется извлечение из стека и заменять имя вызываемого макроса на multipop. И этот макрос будет извлекать регистры со стека в порядке, обратном тому, в каком их туда поместили.

Все это может быть реализовано при помощи следующего определения:

```
%macro multipop 1-*
%rep %0
%rotate -1
          pop %1
%endrep
%endmacro
```

Данный макрос начинает работу с поворота своих аргументов *вправо*, поэтому исходный *последний* аргумент становится аргументом %1. Затем этот аргумент извлекается из стека, список аргументов снова поворачивается вправо и теперь аргументом %1 становится аргумент, бывший в начале предпоследним. Таким образом, аргументы извлекаются из стека в обратном порядке.

### *4.2.7 Объединение параметров макросов*

NASM может объединять параметры макросов с окружающим их текстом. Это позволяет при определении макроса объявлять, например семейства символов. Если вы, например, захотите создать таблицу кодов клавиш вместе со смещениями в этой таблице, вы можете написать:

```
%macro keytab_entry 2
keypos%1 equ $-keytab
         db %2
%endmacro
keytab:
         keytab_entry F1,128+1
         keytab_entry F2,128+2
         keytab_entry Return,13
```

Это будет развернуто следующим образом:

```
keytab:
keyposF1 equ $-keytab
         db 128+1
keyposF2 equ $-keytab
         db 128+2
keyposReturn equ $-keytab
         db 13
```

Точно также можно присоединять текст к другому концу параметра, написав %1foo.

Если вам требуется присоединить к параметру макроса *цифру*, например для определения меток foo1 и foo2 при передаче параметра foo, вы не можете написать %11, так как это будет воспринято как одиннадцатый параметр макроса. Вместо этого вы должны написать %{1}1, где первая единица (определяющая номер параметра макроса) будет отделена от второй (представляющей собой присоединяемый к параметру текст).

Объединение может быть применено и к другим встраиваемым объектам препроцессора, таким как локальные метки макросов (параграф 4.2.2) и контекстно-локальные метки (параграф 4.6.2). В любом случае, неопределенность синтаксиса может быть разрешена путем заключения всего, находящегося после знака % и перед присоединяемым текстом, в фигурные скобки: %{%foo}bar прицепит текст bar к действительному имени локальной метки %%foo. (Это вообще-то излишне, так как форма, используемая NASM для генерации реальных имен локальных макро-меток подразумевает, что и %{%foo}bar, и %%foobar будут развернуты в одно и то же, однако возможность существует).

### *4.2.8 Коды условий в качестве параметров макросов*

NASM может особым образом обрабатывать параметры макросов, содержащие коды условий. Во первых, вы можете ссылаться на макро-параметр %1 при помощи альтернативного синтаксиса %+1, информирующего NASM о том, что этот параметр содержит код условия и заставляющего препроцессор сообщать об ошибке, если макрос вызывается с параметром, *не являющимся* действительным кодом условия.

Однако более полезной возможностью является ссылка на макро-параметр вида %-1, которую NASM будет разворачивать как обратный код условия. Так, макрос retz, описанный в параграфе 4.2.2, может быть заменен макросом условного возврата более общего вида:

```
%macro retc 1
         j%-1 %%skip
         ret
%%skip:
%endmacro
```

Данный макрос может быть теперь вызван как retc ne, что будет развернуто в инструкцию условного перехода JE, или как retc po, что будет преобразовано в переход JPE.

Ссылка %+1 на макро-параметр может вполне спокойно интерпретировать аргументы CXZ и ECXZ как правильные коды условий; однако если передать эту лабуду ссылке %-1, будет сообщено об ошибке, так как обратных кодов условий к таким параметрам не существует.

#### *4.2.9 Подавление развертывания макросов в листинге*

Когда NASM генерирует из программы файл листинга, он обычно разворачивает многострочные макросы посредством записи макро-вызова и последующим перечислением каждой строки, полученной в результате развертывания. Это позволяет увидеть, как генерируются инструкции из макроса в реальный код; однако для некоторых макросов данное загромождение листинга не требуется.

NASM предусматривает для этой цели спецификатор .nolist, который вы можете включить в определение макроса с целью подавления развертывания макроса в файле листинга. Спецификатор .nolist ставится сразу после количества параметров, например так:

```
%macro foo 1.nolist
```

Или так:

```
%macro bar 1-5+.nolist a,b,c,d,e,f,g,h
```

### 4.3 Условное ассемблирование

Как и препроцессор языка C, NASM позволяет ассемблировать отдельные секции исходного файла только тогда, когда выполняются определенные условия. Синтаксис в общем виде выглядит следующим образом:

```
%if<условие>
; некоторый код, ассемблируемый только при выполнении <условия>
%elif<условие2>
; ассемблируется, если <условие> не выполняется, а выполняется <условие2>
%else
; ассемблируется, если и <условие>, и <условие2> не выполняются
%endif
```

Оператор %else необязателен, так же как и оператор %elif. Если нужно, вы можете использовать более одного оператора %elif.

#### *4.3.1 %ifdef: Проверка присутствия однострочного макроса*

Начинающийся со строки %ifdef MACRO условно-ассемблируемый блок будет обрабатываться *только* в том случае, если определен однострочный макрос MACRO. Если он не определен, вместо этого будут обрабатываться блоки %elif и %else (если они есть).

Например, для отладки программы вы можете захотеть ввести следующий код:

```
          ; выполнение некоторой функции
%ifdef DEBUG
          writefile 2,"Функция выполнена полностью.",13,10
%endif
          ; выполнение чего-нибудь еще
```

После этого вы можете использовать ключ -dDEBUG командной строки для создания версии программы, выдающей отладочные сообщения, а удалив этот ключ — создавать окончательный релиз.

Для осуществления обратной проверки (отсутствия определения макроса) вы можете использовать вместо %ifdef оператор %ifndef. Вы можете также тестировать наличие определения макроса в блоках %elif при помощи операторов %elifdef и %elifndef.

### 4.3.2 %ifctx: Проверка контекстного стека

Конструкция условного ассемблирования %ifctx ctxname предполагает обработку идущего вслед за ней кода только тогда, когда на вершине контекстного стека препроцессора находится имя ctxname. Как и в случае с %ifdef, поддерживаются также формы %ifnctx,%elifctx и %elifnctx.

Более подробно о контекстном стеке можно узнать в параграфе 4.6. а пример использования %ifctx приведен в параграфе 4.6.5.

### 4.3.3 %if: Проверка произвольных числовых выражений

Конструкция условного ассемблирования %if expr будет вызывать обработку последующего кода *только* в том случае, если выражение expr не нулевое. Примером использования данной конструкции может служить проверка выхода из препроцессорного цикла %rep: пример смотрите в параграфе 4.4.

Выражения, указываемые для %if, а также его эквивалента %elif, являются критическими (см. параграф 3.7).

%if расширяет обычный синтаксис выражений NASM, предусматривая набор операторов отношения, которые в выражениях обычно запрещены. Операторы =, <, >, <=, >= и <> проверяют на равенство, отношения "меньше чем", "больше чем", "меньше или равно", "больше или равно" и на неравенство соответственно. С-подобные формы == и != также поддерживаются и являются альтернативными формами = и <>. И наконец, имеются операторы низкого приоритета &&, ^^ и ||, производящие логические операции И, ИСКЛЮЧАЮЩЕЕ ИЛИ и ИЛИ. Они работают также, как логические операторы в C (за исключением того, что в C нет логического "ИСКЛЮЧАЮЩЕЕ ИЛИ"), то есть возвращают всегда 0 или 1 и обрабатывают любое ненулевое значение как 1 (например, ^^ возвращает 1 только если одно из его входных значений нулевое, а второе — ненулевое). Операторы отношения также возвращают 1 если условие истинно и 0 — если ложно.

### 4.3.4 %ifidn и %ifidni: Проверка на идентичность текста

Конструкция %ifidn text1,text2 будет вызывать ассемблирование последующего кода *только* в том случае, если текст аргументов text1 и text2 после развертывания однострочных макросов становится идентичным. Отличия в виде пробелов не считаются.

%ifidni подобна %ifidn, но нечувствительна к регистру символов.

Например, следующий макрос помещает регистр или число в стек, позволяя при этом обрабатывать IP как реальный регистр:

```
%macro pushparam 1
%ifidni %1,ip
          call %%label
%%label:
%else
          push %1
%endif
%endmacro
```

Как и большинство других %if-конструкций, %ifidn имеет эквивалент %elifidn и обратные формы %ifnidn и %elifnidn. Соответственно, %ifidni имеет эквивалент %elifidni и обратные формы %ifnidni и %elifnidni.

### 4.3.5 %ifid, %ifnum, %ifstr: Проверка типов символов

Иногда вам может понадобиться, чтобы макросы выполняли различные задачи в зависимости от того, что им передано в качестве аргумента: число, строка или идентификатор. Например, может быть необходимо чтобы макрос смог обрабатывать как строковые константы, так и указатели на уже существующие строки.

Конструкция условного ассемблирования %ifid, принимающая один параметр (который может быть пустым), обрабатывает последующий код *только* в том случае, если первый символ в параметре существует

и является идентификатором. %ifnum работает аналогично, но проверяет символ на соответствие числовой константе; %ifstr тестирует на соответствие символа строке.

Например, описанный в параграфе 4.2.3 макрос writefile может быть расширен для использования преимуществ конструкции %ifstr следующим образом:

```
%macro writefile 2-3+
%ifstr %2
          jmp %%endstr
%if %0 = 3
%%str:    db %2,%3
%else
%%str:    db %2
%endif
%%endstr: mov dx,%%str
          mov cx,%%endstr-%%str
%else
          mov dx,%2
          mov cx,%3
%endif
          mov bx,%1
          mov ah,0x40
          int 0x21
%endmacro
```

После этого макрос writefile может "справиться" со следующими двумя своими вызовами:

```
          writefile [file], strpointer, length
          writefile [file], "Привет!", 13, 10
```

В первом случае strpointer используется в качестве адреса уже объявленной строки, а length — как длина этой строки. Во втором случае макросу передается строка, которую макрос объявляет и получает ее адрес и длину самостоятельно.

Обратите внимание на использование %if внутри %ifstr: это нужно для определения того, передано ли макросу 2 аргумента (в этом случае строка — просто константа и ей достаточно db %2) или больше (в этом случае все аргументы кроме первых двух объединяются в %3 и тогда уже требуется db %2,%3).

Для всех трех конструкций %ifid, %ifnum и %ifstr существуют соответствующие версии %elifXXX, %ifnXXX и %elifnXXX.

### *4.3.6 %error: Сообщения об ошибках, определяемых пользователем*

Директива препроцессора %error заставляет NASM сообщать об ошибках, случающихся на стадии ассемблирования. Так, если ваши исходники будет ассемблировать кто-то еще, вы можете проверить, определен ли нужный макрос при помощи следующего кода:

```
%ifdef SOME_MACRO
; производятся некоторые настройки
%elifdef SOME_OTHER_MACRO
; производятся другие настройки
%else
%error Не определены ни SOME_MACRO, ни SOME_OTHER_MACRO.
%endif
```

Таким образом любой пользователь, не знающий, как правильно ассемблировать ваш код, будет быстро предупрежден об этом несоответствии, вместо того, чтобы увидеть крах программы при ее выполнении и не знать, что этот крах вызвало.

## 4.4 Препроцессорные циклы: %rep

Несмотря на то,что префикс TIMES в NASM весьма удобен, он не может быть использован в многострочных макросах, потому как обрабатывается уже после полного разворачивания последних. Вследствие этого, NASM предусматривает другую форму циклов, работающих на уровне препроцессора, а именно %rep.

Директивы %rep и %endrep (%rep принимает числовой аргумент или выражение; %endrep не принимает никаких аргументов) используются для заключения в них куска кода, который при этом реплицируется столько раз, сколько указано препроцессору:

```
%assign i 0
%rep 64
          inc word [table+2*i]
%assign i i+1
%endrep
```

Этот пример будет генерировать 64 инструкции INC; инкрементируя каждое слово в памяти от [table] до [table+126].

Для образования более сложных условий окончания цикла или его принудительного завершения вы можете использовать директиву %exitrep, например:

```
fibonacci:
%assign i 0
%assign j 1
%rep 100
%if j > 65535
%exitrep
%endif
          dw j
%assign k j+i
%assign i j
%assign j k
%endrep
fib_number equ ($-fibonacci)/2
```

Этот пример создает список всех чисел Фибоначчи, вписывающихся в размер 16 бит. Заметьте, что и в этом случае для %rep должно быть указано максимальное число повторов. Это необходимо для того, чтобы NASM на стадии препроцессирования не впал в бесконечный цикл, что (в многозадачных или многопользовательских системах) приводит обычно к быстрому исчерпанию памяти и невозможности запуска других приложений.

**4.5 Подключение других файлов**

Препроцессор NASMa, используя очень похожий на C синтаксис, позволяет подключать к текщему исходнику другие файлы. Это осуществляется при помощи директивы %include:

```
%include "macros.mac"
```

Эта строка включит файл macros.mac в исходный файл, содержащий директиву %include.

Поиск подключаемых файлов производится в текущем каталоге (каталоге, из которого запускается NASM, а не того, где содержатся его исполнимые файлы или где находится исходный файл) и в любых других каталогах, указанных в командной строке NASM при помощи ключа -i.

Стандартная идиома C, предотвращающая многократное включение одного и того же файла, точно также срабатывает и в NASM: если файл macros.mac имеет форму

```
%ifndef MACROS_MAC
%define MACROS_MAC
; какие-то определения и объявления
```

```
%endif
```

то многократное его включение не будет вызывать ошибок, так как после первого включения символ MACROS_MAC будет уже определен.

При помощи ключа командной строки -p (см. параграф 2.1.7) вы можете подключить файл даже не используя явным образом директиву %include в файле-потребителе.

### 4.6 Контекстный стек

Локальные по отношению к макроопределению метки иногда не обеспечивают необходимую гибкость: вполне возможно, что иногда вы захотите разделить метки между несколькими макровызовами. Примером может служить цикл REPEAT ... UNTIL, в котором расширению макроса REPEAT может понадобиться ссылаться на метки, определенные в макросе UNTIL. Ситуация еще более усложнится, когда вы захотите сделать эти циклы вложенными.

NASM обеспечивает данный уровень гибкости при помощи контекстного стека. Препроцессор поддерживает стек контекстов, каждый из которых характеризуется именем. Добавление нового контекста в стек осуществляется директивой %push, а извлечение из стека — директивой %pop. Вы можете определять метки, являющиеся локальными по отношению к определенному контексту в стеке.

#### *4.6.1 %push и %pop: Создание и удаление контекста*

Директива %push используется для создания нового контекста и помещения его на вершину контекстного стека. Эта директива требует указания одного аргумента, а именно имени контекста. Например:

```
%push foobar
```

Эта команда помещает в стек новый контекст foobar. Вы можете иметь в стеке несколько контекстов с одним и тем же именем: они все равно будут отличаться друг от друга.

Директива %pop, не требующая аргументов, удалает самый верхний контекст из стека и разрушает его вместе с любыми связанными с ним метками.

#### *4.6.2 Контектно-локальные метки*

Точно так же, как использование %%foo вводит локальную по отношению к определенному макросу метку, конструкция %$foo используется для определения метки, локальной по отношению к контексту на вершине контекстного стека. Таким образом, пример с циклом REPEAT – UNTIL может быть реализован следующим образом:

```
%macro repeat 0
%push repeat
%$begin:
%endmacro

%macro until 1
          j%-1 %$begin
%pop
%endmacro
```

Вызовы могут производиться, например, так:

```
          mov cx,string
          repeat
          add cx,3
          scasb
          until e
```

В этом примере будет сканироваться каждый четвертый байт строки с целью поиска байта, равного AL. *(Прим.перев. Скорее всего, в примере ошибка)*.

Если вам требуется определить или получить доступ к меткам, локальным к контексту, находящемуся *ниже* вершины стека, вы можете использовать %$$foo, или %$$$foo для еще более "глубокого" контекста и т.д.

### *4.6.3 Контекстно-локальные однострочные макросы*

NASM позволяет определять однострочные макросы, локальные по отношению к определенному контексту. Например,

```
%define %$localmac 3
```

будет определять однострочный макрос %$localmac, локальный к контексту на вершине стека. Само собой, после создания еще одного контекста директивой %push, данный макрос может быть доступен по имени %$$localmac.

### *4.6.4 %repl: Переименование контекста*

Если вам требуется изменить имя контекста, находящегося на вершине стека , вы можете выполнить %pop с последующим %push; однако это будет иметь "побочный" эффект в виде разрушения всех локальных по отношению к извлеваемому контексту меток и макросов.

В NASM для этой цели предусмотрена директива %repl, *изменяющая* имя контекста без затрагивания связанных с этим контекстом локальных меток и макросов. Теперь вы можете заменить деструктивный код

```
%pop
%push newname
```

на недеструктивную версию %repl newname.

### *4.6.5 Пример использования контекстного стека: Блок IF*

В данном примере для реализации блока IF как набора макросов использованы почти все возможности контекстного стека, включая конструкцию условного ассемблирования %ifctx.

```
%macro if 1
    %push if
    j%-1 %$ifnot
%endmacro

%macro else 0
    %ifctx if
        %repl else
        jmp %$ifend
        %$ifnot:
    %else
        %error "Перед 'else' ожидается 'if' !"
    %endif
%endmacro

%macro endif 0
    %ifctx if
        %$ifnot:
        %pop
    %elifctx else
        %$ifend:
        %pop
    %else
        %error "Перед 'endif' ожидается 'if' или 'else'!"
```

```
    %endif
%endmacro
```

Данный код более устойчив, чем макросы REPEAT и UNTIL из параграфа 4.6.2, так как он использует условное ассемблирование для проверки правильного порядка следования макросов (например, нельзя перед if вызвать endif) и привлекает директиву %error, если порядок нарушен.

Кроме того, макрос endif способен справиться сразу с двумя разными условиями: следует ли он сразу за if или за else. Достигается это опять же за счет условного ассемблирования, при котором производятся разные действия в зависимости от того, что на вершине стека — if или else.

Макрос else должен сохранить контекст в стеке, чтобы метка %$ifnot, на которую ссылается if была той же самой, что и определенная в макросе endif, но в то же время он должен изменить имя контекста, чтобы endif знал, что тут поработал else. Он делает это при помощи %repl.

Пример использования макроса выглядит следующим образом:

```
cmp ax,bx
if ae
  cmp bx,cx
  if ae
    mov ax,cx
  else
    mov ax,bx
  endif
else
  cmp ax,cx
  if ae
    mov ax,cx
  endif
endif
```

Макросы блока IF совершенно спокойно обрабатывают вложенность посредством сохранения контекста, вводимого внутри if поверх контекста, описанного извне if; таким образом else и endif всегда ссылаются на последние if или else, не имеющие на этот момент пары.

### 4.7 Стандартные макросы

NASM вводит набор стандартных макросов, которые на момент начала обработки любого исходного файла будут уже определены. Если вам позарез нужно, чтобы программа ассемблировалась без предопределенных макросов, можете для очистки препроцессорного пространства имен использовать директиву %clear.

Большинство пользовательских директив ассемблера реализованы как макросы, вызывающие примитивные директивы; все они описываются в главе 5. Оставшийся набор стандартных макросов описан ниже.

#### *4.7.1 __NASM_MAJOR__ и __NASM_MINOR__: Версия NASM*

Однострочные макросы __NASM_MAJOR__ и __NASM_MINOR__ разворачиваются соответственно в старшую и младшую части номера версии NASM. Так, в NASM 0.96 __NASM_MAJOR__ будет определен как 0, а __NASM_MINOR__ — как 96.

#### *4.7.2 __FILE__ и __LINE__: Имя файла и номер строки*

Как и в препроцессоре C, NASM позволяет пользователю узнать имя файла и номер строки, содержащие текущую инструкцию. Макрос __FILE__ разворачивается в строковую константу, представляющую собой имя текущего входного файла (которое в ходе ассемблирования может изменяться, если используется директива %include), а __LINE__ разворачивается в числовую константу, означающую текущий номер строки во входном файле.

Эти макросы могут быть использованы, например, для передачи макросу отладочной информации, так как вызов __LINE__ внутри макроопределения (неважно, одно- или многострочного) будет возвращать номер строки *макровызова*, а не строки *определения*. Так, например, для определения в какой части кода наступает крах, пишется подпрограммка stillhere, которой передается в EAX номер строки, а на выходе получается что-то вроде "строка 155: я еще жива". Затем вы пишете макрос

```
%macro notdeadyet 0
          push eax
          mov eax,__LINE__
          call stillhere
          pop eax
%endmacro
```

и "утыкаете" ваш код вызовами notdeadyet до тех пор, пока не найдете точку краха.

### *4.7.3 STRUC и ENDSTRUC: Объявление структурных типов данных*

Ядро NASM не содержит внутренних механизмов для определения структур данных; вместо этого сделан довольно мощный препроцессор, который кроме всего прочего способен реализовать структуры данных в виде набора макросов. Для определения структур данных используются макросы STRUC и ENDSTRUC.

STRUC принимает один параметр, являющийся именем типа данных. Данное имя описывается как символ со значением 0, затем к нему присоединяется суффикс _size и оно определяется как EQU с размером структуры. После того, как STRUC выполнена, вы описываете структуру данных путем определения полей при помощи семейства псевдо-инструкций RESB. В конце описания вы должны вызвать ENDSTRUC.

Например, для определения структуры mytype, содержащей двойное слово, слово, байт и строку, вы можете написать:

```
          struc mytype
mt_long:  resd 1
mt_word:  resw 1
mt_byte:  resb 1
mt_str:   resb 32
          endstruc
```

Данный код определяет шесть символов: mt_long как 0 (смещение от начала структуры mytype до поля с двойным словом), mt_word как 4, mt_byte как 6, mt_str как 7, mytype_size как 39 и собственно mytype как ноль.

Причиной, по которой имя структуры описывается как ноль, является побочный эффект, позволяющий структурам работать с механизмом локальных меток: если члены вашей структуры имеют метки с именами, совпадающими с именами меток других структур, вы можете переписать приведенный выше код следующим образом:

```
          struc mytype
.long:    resd 1
.word:    resw 1
.byte:    resb 1
.str:     resb 32
          endstruc
```

Здесь описываются смещения к полям структуры в виде mytype.long, mytype.word, mytype.byte и mytype.str.

Так как NASM не имеет *встроенной* поддержки структур, он не поддерживает формы нотации как в языке C с использованием точки для ссылки на элементы структуры (за исключением нотации локальных меток), поэтому код наподобие mov ax,[mystruc.mt_word] будет ошибочным. Константа mt_word подобна любым другим константам, поэтому корректным синтаксисом в этом случае будет mov ax,[mystruc+mt_word] или mov ax,[mystruc+mytype.word].

### 4.7.4 ISTRUC, AT и IEND: Объявление экземпляров структур

Имея определение структуры, вы вслед за этим обычно захотите объявить экземпляры этой структуры в сегменте данных (иначе зачем она вообще нужна). NASM предусматривает для этого простой способ с использованием механизма ISTRUC. Для объявления в программе структуры типа mytype вы должны написать следующий код:

```
mystruc:  istruc mytype
          at mt_long, dd 123456
          at mt_word, dw 1024
          at mt_byte, db 'x'
          at mt_str, db 'Привет, фуфел!', 13, 10, 0
          iend
```

Функцией макроса AT является продвижение позиции ассемблирования (при помощи префикса TIMES) в корректную точку заданного поля структуры и затем объявления указанных данных. Вследствие этого, поля структуры должны объявляться в том же самом порядке, в каком они следовали при определении.

Если данные, передаваемые в поле структуры, не помещаются на одной строке, оставшаяся их часть может просто следовать за строкой с AT. Например:

```
          at mt_str, db 123,134,145,156,167,178,189
          db 190,100,0
```

В зависимости от личных предпочтений, вы можете также полностью пропустить код на строке AT и начать поле структуры со следующей строки:

```
          at mt_str
          db 'Привет, фуфел!'
          db 13,10,0
```

### 4.7.5 ALIGN и ALIGNB: Выравнивание данных

Макросы ALIGN и ALIGNB предоставляют удобный способ выравнивания кода или данных по словам, двойным словам, параграфам (16 байт) или другим границам. (В некоторых ассемблерах для этой цели служит директива EVEN). Синтаксис ALIGN и ALIGNB следующий:

```
          align 4               ; выравнивание по 4-байтной границе
          align 16              ; выравнивание по параграфам
          align 8,db 0          ; заполнение 0 вместо NOP
          align 4,resb 1        ; выравнивание 4 в BSS (неиниц. данные)
          alignb 4              ; эквивалент предыдущей строки
```

Оба макроса требуют, чтобы их первый аргумент был степенью двойки; они подсчитывают число дополнительных байт, требуемых для подгонки длины текущей секции до соответствующей границы (произведение со степенью двойки) и затем осуществляют выравнивание путем применения к своему второму аргументу префикса TIMES.

Если второй аргумент не задан, используется значение по умолчанию: NOP для ALIGN и RESB 1 для ALIGNB. Если второй аргумент задан, оба макроса становятся эквивалентными. Обычно вы должны использовать ALIGN в секциях кода и данных, а ALIGNB — в секции BSS. Тогда никакого второго аргумента не понадобится (кроме, конечно, специальных случаев).

Так как ALIGN и ALIGNB являются простыми макросами, проверки ошибок в них нет: они не могут сообщить вам о том, что переданный аргумент не является степенью двойки или что второй аргумент генерирует более одного байта кода. В любом таком случае они будут "молча делать плохие вещи".

ALIGNB (или ALIGN со вторым аргументом RESB 1) могут использоваться при определении структур:

```
          struc mytype2
```

```
mt_byte:  resb 1
          alignb 2
mt_word:  resw 1
          alignb 4
mt_long:  resd 1
mt_str:   resb 32
          endstruc
```

Таким образом гарантируется, что члены структуры осмысленно выровнены относительно ее базы.

И последнее замечание: ALIGN и ALIGNB работают относительно начала секции, а не начала адресного пространства в конечном исполнимом файле. Например, выравнивание по параграфам в секциях, гарантирующих свое выравнивание только по двойным словам — пустая трата времени. NASM не в состоянии проверить, что характеристики выравнивания секции подходят для использования ALIGN или ALIGNB.

## *Глава 5: Директивы ассемблера*

Перевод: AsmOS group, © 2001

Хотя NASM и пытается избежать бюрократизм ассемблеров наподобие MASM и TASM, он вынужден поддерживать *несколько* директив. Все они описаны в этой главе.

Существует два типа директив NASM: *пользовательские* и *примитивные*. Обычно каждая директива имеет как пользовательскую, так и примитивную форму. В большинстве случаев рекомендуется использовать пользовательскую форму директив, которая реализована как макрос, вызывающий примитивные формы. Примитивные директивы заключаются в квадратные скобки; для пользовательских директив этого не требуется.

В дополнение к описанным в этой главе универсальным директивам каждый объектный формат может опционально предоставлять дополнительные директивы, служащие для управления особенностями этого формата. Такие дополнительные директивы описываются далее в главе, посвященной выходным форматам файлов (глава 6).

### 5.1 BITS: Указание разрядности выполняемого кода

Директива BITS указывает, код какой разрядности должен генерировать NASM: для процессора, работающего в 16-битном режиме, или для процессора в 32-битном режиме. Соответствующим синтаксисом будет BITS 16 или BITS 32.

В большинстве случаев вам не потребуется использовать BITS явно. Объектные форматы aout, coff, elf и win32, разработанные для 32-битных операционных систем, вынуждают NASM выбирать 32-битный режим по умолчанию. Объектный формат obj позволяет вам описывать каждый сегмент как USE16 или USE32, поэтому NASM будет соответственно настраивать режим работы и здесь использование директивы BITS также не требуется.

Наиболее вероятным использование директивы BITS представляется при написании 32-битного плоского бинарного файла. (Выходной формат bin по умолчанию предназначен для 16-битного режима, т.к. он наиболее часто используется для написания DOS .COM программ, DOS .SYS драйверов устройств, а также загрузчиков).

*Не нужно* задавать BITS 32 для использования 32-битных инструкций в 16-битных DOS-программах; если вы это сделаете, ассемблер сгенерирует некорректный код, так как он получится 32-битным и на 16-битных платформах будет не работоспособен.

Когда NASM находится в состоянии BITS 16, инструкции, использующие 32-битные данные, префиксируются байтом 0x66, а 32-битные адреса – байтом 0x67. В состоянии BITS 32 справедливо обратное: 32-битные инструкции не нуждаются в префиксе, инструкции, использующие 16-битные данные, префиксируются байтом 0x66, а 16-битные адреса – байтом 0x67.

Директива BITS имеет абсолютно эквивалентные примитивные формы: [BITS 16] и [BITS 32]. Пользовательская форма инструкции является макросом, который не делает ничего, кроме как вызывает соответствующую примитивную форму.

### 5.2 SECTION или SEGMENT: Описание и изменение секций

Директива SECTION (SEGMENT — это абсолютно эквивалентный синоним) указывает, в какую секцию выходного файла будет ассемблирован код, который вы пишете. В некоторых объектных форматах количество и имена секций фиксированы; в других пользователь может сделать их столько, сколько захочет. Поэтому если вы захотите переключиться на секцию, которая в данный момент не существует, директива SECTION может либо вызвать сообщение об ошибке, либо создать новую секцию.

Объектные форматы Unix и bin поддерживают стандартизованные имена секций: .text — для кода, .data — для данных и .bss — для неинициализированных данных. Формат obj наоборот, не признает эти имена секций специальными и более того, удаляет ведущую точку в имени любой секции.

#### *5.2.1 Макрос __SECT__*

Директива SECTION необычна тем, что ее пользовательская форма функционально отличается от примитивной. Примитивная форма [SECTION xyz] просто переключает текущую секцию на указанную. Пользовательская форма SECTION xyz сначала определяет однострочный макрос __SECT__ для примитивной директивы [SECTION], которую собирается выдать, и затем выдает ее. Таким образом, пользовательская директива

```
          SECTION .text
```

развернется в две строки:

```
%define __SECT__ [SECTION .text]
          [SECTION .text]
```

Пользователи могут использовать это в своих собственных макросах. Например, макрос writefile, описанный в параграфе 4.2.3, может быть с успехом переписан в следующей, более изощренной форме:

```
%macro writefile 2+
          [section .data]
%%str:    db %2
%%endstr:
          __SECT__
          mov dx,%%str
          mov cx,%%endstr-%%str
          mov bx,%1
          mov ah,0x40
          int 0x21
%endmacro
```

Данная форма макроса, записывающего строку в файл, сначала временно переключается на секцию данных, используя при этом примитивную форму директивы SECTION, не модифицирующую __SECT__. Далее в секции данных объявляется строка и затем вызывается __SECT__, переключающий контекст на любую секцию, в которой пользователь работал до этого. Это исключает необходимость использования инструкции JMP (как в предыдущей версии макроса) для "перескакивания" данных, а также предотвращает возникновение ошибок в модуле OBJ формата, где пользователь потенциально может использовать различные секции кода.

### 5.3 ABSOLUTE: Определение абсолютных меток

О директиве ABSOLUTE можно думать как об альтернативной форме SECTION: она направляет последующий код не в физическую секцию, а в гипотетическую, начинающуюся с указанного абсолютного адреса. В данном режиме вы можете использовать только инструкции семейства RESB.

ABSOLUTE используется следующим образом:

```
          absolute 0x1A
kbuf_chr  resw 1
kbuf_free resw 1
kbuf      resw 16
```

В данном примере область данных PC BIOS описана как сегмент, начинающийся с адреса 0x1A: приведенный выше код определяет kbuf_chr по адресу 0x1A, kbuf_free по адресу 0x1C и kbuf по адресу 0x1E.

Пользовательская форма ABSOLUTE, так же, как и SECTION, переопределяет макрос __SECT__ в месте своего вызова. Директивы STRUC и ENDSTRUC определены как макросы, использующие директиву ABSOLUTE (и соответственно, __SECT__).

ABSOLUTE в качестве аргумента принимает не только абсолютные константы: это может быть выражение (на самом деле критическое выражение: см. параграф 3.7), а также какое-то значение в сегменте. Например, TSR может реутилизировать свой настроечный код в качестве run-time BSS следующим образом:

```
          org 100h              ; это .COM - программа
          jmp setup             ; код setup идет последним
          ; здесь расположена резидентная часть TSR
setup:    ; теперь идет код, инсталлирующий TSR
          absolute setup
runtimevar1 resw 1
runtimevar2 resd 20
tsr_end:
```

Здесь определяется несколько переменных "на верхушке" setup-кода, так что после завершения его работы это пространство может быть реутилизировано как хранилище данных для работающей TSR. Символ 'tsr_end' может быть использован для расчета общего размера резидентной части TSR.

### 5.4 EXTERN: Импорт символов из других модулей

Директива EXTERN подобна директиве MASM EXTRN и ключевому слову extern в C: она используется для объявления символа, который определен в некотором другом модуле. Не все объектные форматы поддерживают внешние переменные: формат bin этого не может.

Директива EXTERN принимает столько аргументов, сколько вам необходимо. Каждый аргумент является именем символа:

```
          extern _printf
          extern _sscanf,_fscanf
```

Некоторые объектные форматы обеспечивают дополнительные возможности директивы EXTERN. В любом случае, дополнительный синтаксис отделяется от имени символа двоеточием. Например, формат obj при помощи следующей директивы позволяет вам объявить, что базой сегмента внешних символов по умолчанию должна быть группа dgroup:

```
          extern _variable:wrt dgroup
```

Примитивная форма EXTERN отличается от пользовательской тем, что одновременно может принять только один аргумент: поддержка списка аргументов реализуется на уровне препроцессора.

Вы можете объявить одну и ту же переменную как EXTERN более одного раза: NASM спокойно проигнорирует второе и последующие переопределения.

### 5.5 GLOBAL: Экспорт символов в другие модули

Директива GLOBAL — это обратная сторона EXTERN: если один модуль объявляет символ как EXTERN и ссылается на него, то для предотвращения ошибок компоновщика необходимо, чтобы некоторый другой модуль *определил* этот символ и объявил его как GLOBAL. Некоторые ассемблеры для этой цели используют директиву PUBLIC.

Директива GLOBAL, применяемая к символу, должна появляться перед определением этого символа. Она использует тот же самый синтаксис, что и EXTERN, за исключением того, что ссылается на символ, определяемый в этом же модуле. Например:

```
          global _main
_main:    ; некоторый код
```

GLOBAL, как и EXTERN, позволяет вводить после двоеточия специфичный синтаксис объектных форматов. К примеру объектный формат elf позволяет вам указать, чем являются глобальные символы: функциями или данными:

```
          global hashlookup:function, hashtable:data
```

Как и в случае EXTERN, примитивная форма GLOBAL отличается от пользовательской восприятием одновременно только одного аргумента.

### 5.6 COMMON: Определение общих данных

Директива COMMON используется для объявления *общих переменных*. Общая переменная — это глобальная переменная, объявленная в секции неинициализированных данных, поэтому

```
          common intvar 4
```

работает так же, как и

```
          global intvar
          section .bss
intvar    resd 1
```

Отличие состоит в том, что если одна и та же переменная определена в разных модулях, во время связывания (сборки) эти переменные будут *объединены* и ссылки на intvar во всех модулях будут указывать на одно и то же место в памяти.

Директива COMMON, так же как GLOBAL и EXTERN, поддерживает специфичный синтаксис объектных форматов. Например, формат obj позволяет общим переменным быть близкими (near) или дальними (far), а формат elf — задать их выравнивание:

```
          common commvar 4:near  ; работает в OBJ
          common intarray 100:4  ; работает в ELF: выравнивание про 4-байтной
границе
```

И наконец, примитивная форма COMMON, как и в случае EXTERN и GLOBAL, отличается от пользовательской тем, что одновременно принимает только один аргумент.

## *Глава 6: Выходные форматы*

Перевод: AsmOS group, © 2001

NASM — портируемый ассемблер, разработанный для компиляции на любых платформах, поддерживающих ANSI C и создающий исполнимые файлы различных Intel x86-совместимых операционных систем. Для обеспечения этого он поддерживает большое число выходных форматов, выбираемых при помощи ключа -f командной строки. В данной главе описываются все эти форматы вместе с их специфичным синтаксисом.

Как упоминалось в параграфе 2.1.1, NASM выбирает имя по умолчанию для выходного файла на основе имени входного файла и указанного выходного формата. При этом расширение входного файла (.asm, .s или

др.) удаляется и вместо него подставляется расширение соответствующего выходного формата. Расширения файлов приводятся ниже в описаниях каждого формата.

### 6.1 bin: Плоский бинарный формат

Формат bin не создает объектных файлов: в выходной файл не генерируется ничего, кроме написанного вами кода. Такие "чисто бинарные" файлы используются в MS-DOS: как исполнимые файлы .COM и драйвера устройств .SYS. Бинарный формат полезен также при разработке операционных систем и загрузчиков.

Формат bin поддерживает только три стандартные секции с именами .text, .data и .bss. В файле, сгенерированном NASM, сначала будет идти содержимое секции .text, а затем содержимое секции .data, выровненное по 4-байтной границе. Секция .bss не хранится в выходном файле, но предполагается, что она будет расположена сразу после секции .data, опять же выровненная по 4-байтной границе.

Если вы явно не указываете директиву SECTION, написанный вами код будет направлен по умолчанию в секцию .text.

Использование формата bin переводит NASM по умолчанию в 16-битный режим (см. параграф 5.1). Чтобы использовать его для написания 32-битного кода (например, ядра ОС), вам необходимо явно указать директиву BITS 32.

По умолчанию bin не создает расширение файла: он просто оставляет исходное имя файла, удалив предварительно его расширение. Таким образом, NASM по умолчанию будет ассемблировать binprog.asm в бинарный файл binprog.

#### *6.1.1 ORG: Начало бинарного файла*

Формат bin предусматривает дополнительную к списку главы 5 директиву ORG. Функция этой директивы — задавать начальный адрес программы, с которого она располагается при загрузке в память.

Например, следующий код будет генерировать двойное слово 0x00000104:

```
          org 0x100
          dd label
label:
```

В отличие от директивы ORG, применяемой MASM-совместимыми ассемблерами, которые позволяют перемещаться в объектном файле и переписывать уже сгенерированный код, NASM-овская ORG означает только то, что и соответствующее слово: *origin (начало)*. Ее единственная функция — задавать смещение, которое будет прибавляться ко всем ссылкам на адреса внутри файла; она не допускает такого жульничества, как MASM. Дополнительные комментарии содержатся в параграфе 10.1.3.

#### *6.1.2 bin-расширение директивы SECTION*

Выходной формат bin расширяет директиву SECTION (или SEGMENT), позволяя вам указывать требования к выравниванию сегментов. Это делается путем добавления в конец строки определения секции спецификатора ALIGN. Например, строка

```
          section .data align=16
```

переключается на секцию .data и также указывает, что эта секция должна быть выровнена в памяти по границе параграфа.

Спецификатор ALIGN указывает, сколько младших бит в начальном адресе секции должны быть установлены в ноль. Указываемое значение выравнивания может быть любой степенью двойки.

### 6.2 obj: Объектные файлы OMF Microsoft

Файлы формата obj (исторически в NASM они называются obj, а не omf), создаваемые MASMом и TASMом, обычно "скармливаются" 16-битным DOS-компоновщикам, на выходе которых получаются .EXE файлы. Этот формат используется также в OS/2.

Формат obj предполагает расширение выходного файла по умолчанию .obj.

obj не является исключительно 16-битным форматом: NASM полностью поддерживает 32-битные расширения этого формата. В частности, 32-битный obj формат используется Win32-компиляторами Borland, которые не применяют новый объектный формат win32 от Майкрософт.

Формат obj не определяет специальных имен сегментов: вы можете называть ваши сегменты как угодно. Типичными именами сегментов obj формата обычно являются CODE, DATA и BSS.

Если в вашем исходном файле до явного указания директивы SEGMENT содержится код, NASM будет помещать его в собственный сегмент с именем __NASMDEFSEG.

Когда вы определяете сегменты в obj файле, NASM воспринимает их имена как символы, поэтому вы можете легко получить адрес любого определенного сегмента, например:

```
          segment data
dvar:     dw 1234
          segment code
function: mov ax,data            ; получение сегментного адреса данных
          mov ds,ax              ; и помещение его в DS
          inc word [dvar]        ; теперь эта ссылка работает
          ret
```

Формат obj разрешает также использование операторов SEG и WRT, поэтому вы можете писать код, делающий что-то вроде

```
          extern foo
          mov ax,seg foo         ; получение сегмента foo
          mov ds,ax
          mov ax,data            ; получение другого сегмента
          mov es,ax
          mov ax,[ds:foo]        ; это доступ к 'foo'
          mov [es:foo wrt data],bx  ; так это делается
```

### *6.2.1 obj-расширения директивы SEGMENT*

Выходной формат obj расширяет директиву SEGMENT (или SECTION), позволяя задавать различные характеристики определяемого сегмента. Это делается путем добавления в конец строки определения сегмента дополнительных спецификаторов. Например,

```
          segment code private align=16
```

определяет сегмент code, объявляя его при этом закрытым и требуя, чтобы код этого модуля был выровнен по границе параграфа.

Имеются следующие спецификаторы:

- PRIVATE, PUBLIC, COMMON и STACK задают комбинацию характеристик сегмента. Сегменты PRIVATE не могут быть связаны компоновщиком с любыми другими; сегменты PUBLIC и STACK на стадии компоновки объединяются; сегменты COMMON накладываются друг на друга, вместо того, чтобы следовать друг за другом.
- ALIGN используется, как уже упоминалось, для указания того, сколько младших бит в стартовом адресе сегмента должно быть установлено в 0. Значение выравнивания может быть любой степенью двойки от 1 до 4096; реально поддерживаются только значения 1, 2, 4, 16, 256 и 4096, поэтому если указать 8, оно будет округлено до 16, а 32, 64 и 128 будут округлены до 256 и т.д. Имейте в виду,

что выравнивание по 4096-битной границе является PharLap-расширением формата и может не поддерживаться некоторыми компоновщиками.

- CLASS используется для указания класса сегмента; этот спецификатор указывает компоновщику, что сегменты одного и того же класса должны быть расположены в выходном файле рядом друг с другом. Имя класса может быть любым словом, например CLASS=CODE.
- OVERLAY, как и CLASS, имеет аргумент в виде произвольного слова и предоставляет компоновщику (способному на это) оверлейную информацию.
- Сегменты могут быть объявлены как USE16 или USE32, что непосредственным образом влияет на объектный файл, а также заставляет NASM при ассемблировании такого сегмента переключаться на 16- или 32-битный режим соответственно.
- При создании объектных файлов OS/2 вы должны объявлять 32-битные сегменты как FLAT, при этом базовый сегмент по умолчанию приписывается к специальной группе FLAT, а также определяет эту группу, если она еще не существует.
- Формат obj позволяет также объявлять сегменты с предопределенными абсолютными адресами, несмотря на то, что компоновщики на данный момент не могут гибко использовать данную возможность; NASM все же позволяет вам объявить сегмент наподобие SEGMENT SCREEN ABSOLUTE=0xB800, если он вам нужен. Ключевые слова ABSOLUTE и ALIGN являются взаимоисключающими.

Атрибуты сегмента по умолчанию в NASM — это *PUBLIC, ALIGN=1, без класса, без оверлея и USE16*.

### 6.2.2 GROUP: Определение групп сегментов

Формат obj позволяет группировать сегменты, благодаря чему для ссылки на все сегменты группы может использоваться один сегментный регистр. NASM для этого поддерживает директиву GROUP. Вы можете написать:

```
segment data
     ; некоторые данные
segment bss
     ; некоторые неинициализированные данные
group dgroup data bss
```

При ассемблировании этого кода будет создана группа dgroup, содержащая сегменты data и bss. Как и в случае SEGMENT, имя директивы GROUP определяется как символ, поэтому вы можете ссылаться на переменную var в сегменте data как var wrt data или как var wrt dgroup, в зависимости от того, какое значение в данный момент находится в сегментном регистре.

Однако если вы ссылаетесь на var, объявленную в сегменте группы, NASM по умолчанию предоставляет смещение относительно начала *группы*, а не *сегмента*. Вследствие этого SEG var также будет возвращать базу группы, а не базу сегмента.

NASM позволяет сегменту являться частью более одной группы, однако при этом он будет выдавать предупреждение. Переменные, объявленные в таком сегменте, по умолчанию будут относиться к первой включающей данный сегмент группе.

Группа может не иметь ни одного сегмента; однако при этом вы можете делать WRT ссылки на группу, которая не содержит нужную переменную. OS/2, например, определяет специальную группу FLAT без всяких сегментов.

### 6.2.3 UPPERCASE: Отключение чувствительности к регистру

В отличие от NASM, некоторые OMF компоновщики нечувствительны к регистру символов; вследствие этого иногда полезно заставлять NASM генерировать объектные файлы в одном регистре. Директива формата UPPERCASE перед записью в объектный файл переводит имена всех сегментов, групп и символов в верхний регистр. В пределах исходного файла NASM остается регистро-чувствительным; однако объектные файлы, если это необходимо, могут быть составлены целиком в верхнем регистре.

UPPERCASE должна оставаться единственной в строке; она не требует никаких параметров.

### 6.2.4 IMPORT: Импортирование символов из DLL

Директива IMPORT определяет символ, импортируемый из DLL и используемый например для написания в NASM библиотеки импорта. При использовании директивы IMPORT вы должны объявить символ как EXTERN.

Директива IMPORT требует указания двух параметров, разделенных пробелом и являющихся соответственно именем импортируемого символа и именем библиотеки, из которой он импортируется. Например:

```
import WSAStartup wsock32.dll
```

Третьим (необязательным) параметром является имя, под которым импортируемый символ известен в соответствующей библиотеке. Этот параметр применяется в том случае, если вы хотите, чтобы имя символа, используемое внутри вашего кода, не совпадало с именем того же символа в библиотеке. Например:

```
import asyncsel wsock32.dll WSAAsyncSelect
```

### 6.2.5 EXPORT: Экспортирование символов в DLL

Директива EXPORT определяет глобальный символ, экспортируемый как DLL-символ в создаваемую DLL (если вы пишете DLL в NASM). Чтобы использовать директиву EXPORT, вы должны объявить символ как GLOBAL.

EXPORT требует один параметр, являющийся именем экспортируемого символа (в том виде, в каком он определен в вашем исходном файле). Необязательный второй параметр (отделенный пробелом от первого) представляет собой *внешнее* имя символа: это имя, под которым данный символ будет известен использующим эту DLL программам. Если это имя то же самое, что и внутреннее, можете оставить второй параметр пустым.

Для определения атрибутов экспортируемого символа могут быть заданы дополнительные параметры. Эти параметры, как и второй, отделяются друг от друга пробелами. Если дополнительные параметры задаются, внешнее имя также должно указываться, неважно, совпадает ли оно с внутренним или нет. Имеются следующие дополнительные атрибуты:

- resident указывает, что экспортируемое имя должно быть оставлено системным загрузчиком резидентным. Это делается для оптимизации часто используемых символом, импортируемых по имени.
- nodata указывает, что экспортируемый символ является функцией, не использующей никаких неинициализированных данных.
- parm=NNN, где NNN целое, устанавливает число параметров-слов в случае, если символ является вызовом-переходом между 32- и 16-битными сегментами.
- Атрибуты в виде числа указывают, что символ должен экспортироваться с идентификационным номером (ординалом) и представляют собой собственно этот номер.

Например:

```
export myfunc
export myfunc TheRealMoreFormalLookingFunctionName
export myfunc myfunc 1234  ; экспортирование по ординалу
export myfunc myfunc resident parm=23 nodata
```

### 6.2.6 ..start: Определение точки входа в программу

OMF-компоновщики требуют, чтобы один из компонуемых объектных файлов определял точку входа в программу (место, откуда начнется выполнение программы после загрузки). Если этот файл ассемблируется при помощи NASM, вы можете указать точку входа путем объявления в этом месте специального символа ..start.

### 6.2.7 *obj-расширения директивы EXTERN*

Если вы объявляете внешний символ директивой

```
extern foo
```

то ссылки наподобие mov ax,foo будут давать вам смещение foo относительно базы его предопределенного сегмента (как задано в модуле, где определен foo). Таким образом, доступ к содержимому foo вы будете обычно осуществлять следующим образом

```
mov ax,seg foo          ; получение базы сегмента
mov es,ax               ; перемещение ее в ES
mov ax,[es:foo]         ; и использование смещения 'foo' от нее
```

Это получается немного громоздко, особенно если вы знаете, что внешний символ будет доступен из данного сегмента или группы (например, dgroup). Так, если DS уже содержит dgroup, вы можете написать

```
mov ax,[foo wrt dgroup]
```

Однако необходимость каждый раз это печатать для получения доступа к foo очень утомительна; NASM позволяет вам объявить foo в альтернативной форме

```
extern foo:wrt dgroup
```

Эта форма заставляет NASM делать вид, что базовым сегментом foo на самом деле является dgroup; так, выражение seg foo теперь будет возвращать dgroup, а выражение foo будет эквивалентно foo wrt dgroup.

Описанный WRT-механизм может быть использован для "проявления" внешних символов по отношению к любой группе или сегменту в программе. Он также может быть применен к общим переменным (см. параграф 6.2.8 ниже).

### 6.2.8 *obj-расширения директивы COMMON*

Формат obj позволяет общим переменным быть как ближними, так и дальними; NASM обеспечивает этот механизм при помощи следующего синтаксиса:

```
common nearvar 2:near  ; 'nearvar' - ближняя переменная
common farvar 10:far   ; 'farvar' - дальняя переменная
```

Дальние общие переменные могут иметь размер больше 64 Кб, поэтому спецификация OMF говорит, что они объявляются как число *элементов* данного размера. Так, 10-байтная дальняя переменная должна быть объявлена как 10 однобайтных элементов, 5 двухбайтных, 2 пятибайтных или 1 десятибайтный элемент.

Некоторые OMF-компоновщики при разрешении общих переменных, объявленных более чем в одном модуле, требуют совпадения размеров элемента и переменной. Поэтому NASM должен позволять задавать размер элемента дальних переменных. Он делает это при помощи следующего синтаксиса:

```
common c_5by2 10:far 5 ; два пятибайтных элемента
common c_2by5 10:far 2 ; пять двухбайтных элементов
```

Если размер элемента не задан, он предполагается по умолчанию равным 1. Не требуется также и явное указание ключевого слова FAR, так как только дальние общие переменные могут иметь размер элемента. Исходя из этого, приведенные выше объявления будут эквивалентны следующим:

```
common c_5by2 10:5     ; два пятибайтных элемента
common c_2by5 10:2     ; пять двухбайтных элементов
```

В дополнение ко всему вышесказанному, директива COMMON в obj поддерживает также и спецификацию WRT-умолчаний наподобие тому, как это работает для EXTERN (см. параграф 6.2.7). Поэтому вы можете объявлять такие, например, вещи:

```
common foo 10:wrt dgroup
common bar 16:far 2:wrt data
common baz 24:wrt data:6
```

### 6.3 win32: Объектные файлы Win32 Microsoft

Выходной формат win32 генерирует объектные файлы Microsoft Win32, передаваемые обычно компоновщикам Microsoft. Заметьте, что компиляторы Borland Win32 не используют этот формат, вместо него они используют obj (см. параграф 6.2).

win32 подразумевает по умолчанию расширение выходного объектного файла .obj.

Имейте в виду, что хотя Майкрософт и утверждает, что объектные файлы Win32 следуют стандарту COFF, объектные файлы, созданные компиляторами Microsoft Win32, не совместимы с COFF-компоновщиками (например, DJGPP) и наоборот. Это происходит из-за разницы во взглядах на семантику таблицы перемещений. Для создания COFF-совместимых выходных файлов для DJGPP используйте выходной формат coff NASMa; обратное также справедливо — файлы, полученные в объектном формате coff, не обрабатываются корректно компоновщиками Win32.

#### *6.3.1 win32-расширения директивы SECTION*

Как и формат obj, win32 позволяет вам указывать в строке с директивой SECTION дополнительную информацию, предназначенную для управления типом и свойствами определяемой секции. Обычно тип секции и ее свойства для стандартных имен .text, .data и .bss генерируются NASM автоматически, но при помощи приведенных спецификаторов могут быть и переопределены.

Имеются следующие спецификаторы:

- code или text (что эквивалентно) описывает секцию как секцию кода. При этом секция маркируется как читаемая и выполняемая, но не записываемая; компоновщику указывается также, что тип секции — код.
- data и bss, по аналогии с code, определяют секцию данных. Секция данных маркируется как читаемая и записываемая, но не выполняемая. Спецификатор data объявляет секцию инициализированных данных, в то время как спецификатор bss — неинициализированных.
- info описывает секцию как информационную, не включаемую компоновщиком в исполнимый файл, но способную (например) передать компоновщику информацию. Например, объявление info-секции с именем .drectve побуждает компоновщик интерпретировать ее содержание как опции командной строки.
- align= с последующим числом описывает требования к выравниванию секции. Максимум, что вы можете задать — 64: объектный формат Win32 не имеет способа запросить выравнивание большее, чем это. Если выравнивание явно не указано, для секций кода используется выравнивание по параграфам, а для секций данных — по двойным словам. Информационные секции получают по умолчанию выравнивание в 1 байт (т.е. нет его), даже если значение указано явно.

Если вы не укажете ни одного описанного выше спецификатора, NASM принимает следующие значения по умолчанию:

```
section .text code align=16
section .data data align=4
section .bss bss align=4
```

Любые другие имена секций обрабатываются так же, как и .text.

### 6.4 coff: Общий формат объектных файлов

Выходной формат coff создает COFF-объектные файлы, обрабатываемые компоновщиком DJGPP. Этот формат предусматривает по умолчанию расширение выходных файлов .o.

Формат coff поддерживает те же самые расширения директивы SECTION, что и win32, однако спецификатор align и секция типа info не поддерживаются.

### 6.5 elf: Объектные файлы ELF Линукс

Выходной формат elf генерирует объектные файлы ELF32 (Executable and Linkable Format), используемые в Линукс. Этот формат использует по умолчанию расширение .o выходных файлов.

#### *6.5.1 elf-расширения директивы SECTION*

Как и формат obj, elf позволяет вам указывать в строке с директивой SECTION дополнительную информацию, предназначенную для управления типом и свойствами определяемой секции. Обычно тип секции и ее свойства для стандартных имен .text, .data и .bss генерируются NASM автоматически, но при помощи приведенных спецификаторов могут быть и переопределены.

Имеются следующие спецификаторы:

- alloc описывает секцию, которая при запуске программы загружается в память. noalloc описывает незагружаемые секции, такие как информационные или секции комментариев.
- exec описывает секцию, которая при запуске программы может выполняться (разрешено ее выполнение). Секции noexec выполняться не могут.
- write описывает секцию, в которую после запуска программы разрешается запись. В секции nowrite запись запрещена.
- progbits описывает секцию, явно сохраняемую в объектном файле: обычно это секции кода и инициализированных данных. nobits описывает секции, не присутствующие в файле, такие как BSS (неинициализированных данных).
- align= с последующим числом, как и obj, задает параметр выравнивания секции.

Если вы не укажете ни одного описанного выше спецификатора, NASM принимает следующие значения по умолчанию:

```
section .text progbits alloc   exec nowrite align=16
section .data progbits alloc noexec   write align=4
section .bss    nobits alloc noexec   write align=4
section other progbits alloc noexec nowrite align=1
```

(Любые секции, не являющиеся .text, .data или .bss обрабатываются так же, как и секция other).

#### *6.5.2 Позиционно-независимый код: Специальные символы формата elf и WRT*

Спецификация ELF содержит достаточное число возможностей написания позиционно-независимого кода (PIC), благодаря которым разделяемые библиотеки ELF весьма гибкие. Однако это также означает, что NASM должен быть способен генерировать разнообразные замысловатые типы релокейшнов объектных ELF-файлов.

Так как ELF не поддерживает сегментные ссылки, оператор WRT для своей обычной цели не используется; вместо этого формат elf использует WRT для других целей, а именно для специальных PIC-типов перемещений.

elf определяет пять специальных символов, которые вы можете использовать с правой стороны оператора WRT для получения PIC-типов перемещений. Это ..gotpc, ..gotoff, ..got, ..plt и ..sym. Их функции описаны ниже:

- Ссылка на символ, помеченный как база глобальной таблицы смещений (GOT) при помощи wrt ..gotpc, будет в результате давать расстояние от начала текущей секции до глобальной таблицы смещений. (Для ссылки на GOT используется обычно стандартный символ

_GLOBAL_OFFSET_TABLE_). Таким образом, для получения реального адреса GOT, вам необходимо прибавить к результату конструкцию $$.

- Ссылка на определенную позицию в одной из ваших собственных секций при помощи wrt ..gotoff будет давать расстояние от начала GOT до заданной позиции, поэтому прибавив это расстояние к адресу GOT, вы получите реальный адрес нужной вам позиции.
- Ссылка на внешний или глобальный символ при помощи wrt ..got заставляет компоновщик построить элемент GOT, содержащий адрес символа, поэтому ссылка дает расстояние от начала GOT до этого элемента. Таким образом, прибавляя к полученной ссылке адрес GOT, вы получите адрес, по которому содержится адрес символа.
- Ссылка на имя процедуры при помощи wrt ..plt заставляет компоновщик построить элемент таблицы компоновки процедуры (PLT) и ссылка в результате дает адрес элемента PLT. Вы можете использовать это обычно только в контексте создания относительных перемещений (т.е. цели для CALL или JMP), так как ELF абсолютно не имеет перемещаемых типов для ссылки на элементы PLT.
- Ссылка на имя символа при помощи wrt ..sym заставляет NASM записать обычное перемещение, однако вместо того, чтобы сделать перемещение относительным к началу секции и затем добавить смещение символа, он создаст запись смещения непосредственно к запрашиваемому символу. Различие необходимо из-за особенностей динамического компоновщика.

Более полное объяснение использования типов перемещений для написания в NASM разделяемых библиотек дано в параграфе 8.2.

### 6.5.3 elf-расширения директивы GLOBAL

Объектные файлы ELF могут содержать больше информации о глобальном символе, чем просто его адрес: они могут содержать размер символа, а также его тип. Это сделано не только для удобства при отладке, это просто необходимо, если программа пишется в виде разделяемой библиотеки. Для задания дополнительной информации NASM поддерживает некоторые расширения директивы GLOBAL.

Поставив после имени глобальной переменной двоеточие и написав function или data (object является синонимом data), вы можете указать, чем является глобальная переменная — функцией или объектом данных. Например, строка

```
        global hashlookup:function, hashtable:data
```

экспортирует глобальный символ hashlookup как функцию, а hashtable — как объект данных.

После ввода спецификатора вы можете также указать в виде числового выражения (которое может включать метки и даже опережающие ссылки) размер ассоциированных с символом данных, например:

```
        global hashtable:data (hashtable.end - hashtable)
hashtable:
        db this,that,theother  ; здесь некоторые данные
.end:
```

Это заставит NASM автоматически подсчитать длину таблицы и поместить эту информацию в символьную таблицу ELF.

Объявление типа и размера глобальных символов требуется при написании разделяемых библиотек. Дополнительную информацию вы можете найти в параграфе 8.2.4.

### 6.5.4 elf-расширение директивы COMMON

ELF позволяет задавать требования к выравниванию общих переменных. Это делается путем помещения числа (которое должно быть степенью двойки) после имени и размера общей переменной и отделения этого числа (как обычно) двоеточием. Например, массив двойных слов, который должен быть выровнен по двойным словам:

```
        common dwordarray 128:4
```

Эта строка объявляет массив размером 128 байт и требует, чтобы он был выровнен по 4-байтной границе.

### 6.6 aout: Объектные файлы a.out Линукс

Формат aout генерирует объектные файлы a.out в форме, используемой устаревшими Линукс системами. (Он отличается от других объектных файлов a.out магическим числом в первых четырех байтах файла. Также некоторые реализации a.out, например NetBSD, поддерживают позиционно-независимый код, который реализация Линукс не знает).

Формат a.out подразумевает расширение выходных файлов по умолчанию .o.

Этот формат очень простой. Он не поддерживает специальных директив и символов, не использует SEG или WRT и в нем нет расширений никаких стандартных директив. Он поддерживает только три стандартных секции с именами .text, .data и .bss.

### 6.7 aoutb: Объектные файлы a.out NetBSD/FreeBSD/OpenBSD

Формат aoutb генерирует объектные файлы a.out в форме, используемой различными BSD-клонами UNIX: NetBSD, FreeBSD и OpenBSD. Для большинства объектных файлов этот формат не отличается от aout за исключением магического числа в первых четырех байтах файла. Однако формат поддерживает (как и формат elf) позиционно-независимый код, поэтому вы можете использовать его для написания разделяемых библиотек BSD.

Расширение объектных файлов формата aoutb по умолчанию .o.

Формат не поддерживает специальных директив и символов и имеет только три стандартных секции с именами .text, .data и .bss. Несмотря на это, для обеспечения типов перемещений в позиционно-независимом коде он поддерживает использование WRT так же, как это делает elf. Более подробно это описано в параграфе 6.5.2.

aoutb, как и elf поддерживает также расширение директивы GLOBAL: см. параграф 6.5.3.

### 6.8 as86: Объектные файлы as86 Линукс

16-битный ассемблер Линукс as86 имеет свой собственный нестандартный формат объектных файлов. Хотя его компаньон компоновщик ld86 выдает что-то близкое к обычным бинарникам a.out, объектный формат, используемый для взаимодействия между as86 и ld86, все же не является a.out.

NASM на всякий случай поддерживает данный формат как as86. Расширение выходного файла по умолчанию для данного формата .o.

Формат as86 — это очень простой объектный формат (с точки зрения NASM). Он не поддерживает специальных директив и символов, не использует SEG и WRT, и в нем нет никаких расширений стандартных директив. Он поддерживает только три стандартных секции с именами .text, .data и .bss.

### 6.9 rdf: Перемещаемые динамические объектные файлы

Выходной формат rdf создает объектные файлы RDOFF. RDOFF — это "доморощенный" формат объектных файлов, разработанный вместе с NASM и отражающий в себе внутреннюю структуру ассемблера.

RDOFF не используется никакими широко известными операционными системами. Однако тот, кто пишет собственную систему, возможно захочет использовать его в качестве собственного объектного формата, так как разработан он прежде всего для упрощения и содержит очень мало бюрократии в заголовках файлов.

Архив Unix NASM и архив DOS с исходниками имеют подкаталог rdoff, содержащий набор RDOFF-утилит: RDF-компоновщик, менеджер статических библиотек, утилита, делающая дамп RDF-файла, и программа, загружающая и выполняющая RDF-исполнимый файл под Линукс.

Формат rdf поддерживает только стандартные секции с именами .text, .data и .bss.

### *6.9.1 Требование библиотеки: Директива LIBRARY*

RDOFF содержит механизм "требования библиотеки", которая будет связана с модулем как во время загрузки, так и при выполнении. Это осуществляется директивой LIBRARY, которая принимает один аргумент, являющийся именем модуля:

```
library mylib.rdl
```

## 6.10 dbg: Формат для отладки

Выходной dbg формат в конфигурации NASM по умолчанию отсутствует. Если вы строите собственный исполнимый файл NASM из исходников, то можете для включения этого формата определить символ OF_DBG в файле outform.h или командной строке компилятора.

Формат dbg не создает объектных файлов как таковых; вместо этого он создает текстовый файл, содержащий полный список всех транзакций между ядром NASM и модулем выходных форматов. Это обычно нужно людям, намеревающимся написать собственные выходные драйвера и которые благодаря этому формату могут получить картину различных запросов основной программы к выходному драйверу и увидеть, в каком порядке они осуществляются.

Для простых файлов можно использовать dbg формат так:

```
nasm -f dbg filename.asm
```

в результате чего генерируется диагностический файл filename.dbg. Однако это не будет работать на файлах, разработанных под различные объектные форматы, так как каждый формат определяет собственные макросы (обычно пользовательские формы директив), не определенные в формате dbg. Поэтому здесь необходимо запускать NASM дважды, с препроцессированием выбранного объектного формата:

```
nasm -e -f rdf -o rdfprog.i rdfprog.asm
nasm -a -f dbg rdfprog.i
```

Здесь rdfprog.asm препроцессируется в rdfprog.i, оставляя при этом выбранным объектный формат rdf. Это нужно, чтобы специальные директивы RDF правильно конвертировались в примитивную форму. Затем препроцессированный исходный файл обрабатывается в формате dbg и при этом генерируется окончательный диагностический файл.

Такой обходной путь обычно не будет работать с программами, предназначенными для формата obj, так как директивы SEGMENT и GROUP последнего имеют побочные эффекты определения имен сегментов и групп как символов; dbg этого не делает, поэтому программа ассемблироваться не будет. Если вам позарез нужно получить dbg-трассировку исходников, написанных для obj, вы можете обойти это, определив символы самостоятельно (например, при помощи EXTERN).

Формат dbg принимает любые имена секций и любые директивы, протоколируя все их в свой выходной файл.

# *Глава 7: Написание 16-битного кода (DOS, Windows 3/3.1)*



В данной главе рассмотрены некоторые общие вопросы создания 16-битного кода, выполняющегося под MS-DOS или Windows 3.x: как скомпоновать программы для получения .EXE или .COM файлов, как создать драйвер устройства .SYS, а также как ассемблерный код взаимодействует с 16-битными компиляторами и с Borland Pascal.

## 7.1 Получение .EXE файлов

Любые большие программы, написанные под DOS, необходимо создавать как .EXE файлы: только они имеют необходимую внутреннюю структуру для захвата более одного 64K сегмента. Программы Windows также требуется создавать как .EXE файлы, так как .COM файлы Windows не поддерживает.

Обычно .EXE файлы генерируются при помощи выходного формата obj (при этом создаются один или более .OBJ файлов, связываемых затем друг с другом компоновщиком). Однако при помощи выходного формата bin и некоторых макросредств NASM поддерживает также непосредственное создание простых .EXE файлов DOS (заголовок .EXE файла конструируется при помощи DB и DW). Спасибо Yann Guidon за содействие при кодировании этого.

В будущем NASM может быть станет поддерживать и "родной" выходной .EXE формат.

### *7.1.1 Использование формата obj для получения .EXE файлов*

В данном параграфе описан обычный способ создания .EXE файлов путем компоновки друг с другом .OBJ файлов.

Большинство 16-битных языков программирования поставляются с собственным компоновщиком; если у вас нет ни одного, возьмите с x2ftp.oulu.fi свободно распространяемый компоновщик VAL, упакованный в формате LZH. LZH-архиватор можно найти на ftp.simtel.net. На www.pcorner.com имеется еще один бесплатный компоновщик FREELINK (только он без исходников), и наконец, на www.delorie.com можно взять компоновщик djlink, написанный DJ Delorie.

При компоновке нескольких .OBJ файлов в один .EXE файл вы должны убедиться, что только один из них (.OBJ) имеет точку входа (при помощи специального символа ..start, определяемого obj форматом: см. параграф 6.2.6). Если ни один из модулей не определяет точки входа, компоновщик не будет знать, какое значение записать в заголовок выходного файла в качестве стартового адреса; если же точка входа определена в нескольких файлах, компоновщик не сможет понять, *какое* именно значение использовать.

Здесь приводится пример исходного файла, который ассемблируется NASMом в .OBJ файл и им же компонуется в .EXE. На этом примере продемонстрированы основные принципы определения стека, инициализации сегментных регистров и объявления точки входа. Данный файл содержится также в подкаталоге test NASM-архивов под именем objexe.asm.

```
          segment code

..start:  mov ax,data
          mov ds,ax
          mov ax,stack
          mov ss,ax
          mov sp,stacktop
```

Эта инициализационная часть кода устанавливает DS на сегмент данных и инициализирует SS и SP для указания на вершину стека. Заметьте, что после записи в SS прерывания неявно запрещаются на время выполнения следующей команды, в качестве которой подразумевается загрузка SP. Это необходимо для корректной инициализации стека.

Заметьте также, что в начале данного кода определен символ ..start, который в результирующем исполнимом файле будет являться точкой входа.

```
          mov dx,hello
          mov ah,9
          int 0x21
```

Здесь начинается основная программа: загрузка в DS:DX указателя на приветствующее сообщение (hello является неявной ссылкой на сегмент data, загруженный в DS настроечным кодом, поэтому полный указатель корректен) и вызов DOS-функции вывода строки на экран.

```
          mov ax,0x4c00
          int 0x21
```

Здесь программа завершается при помощи другого системного DOS-вызова.

```
          segment data
```

```
hello:    db 'Привет, фуфел!', 13, 10, '$'
```

Сегмент данных содержит строку, которую нужно отобразить на экране.

```
          segment stack stack
          resb 64
stacktop:
```

Данный код объявляет сегмент стека, содержащий 64 байта неинициализированного стекового пространства, где символ stacktop указывает на его вершину. Директива segment stack stack определяет сегмент под *именем* stack, *тип* которого также STACK. В нашем случае не требуется дальнейшее выполнение программы, но если в ней не определить сегмент STACK, компоновщики вероятнее всего выдадут предупреждение или сообщение об ошибке.

Приведенный выше файл будет ассемблироваться в .OBJ файл, затем компоноваться NASM в корректный .EXE файл, который при запуске будет выводить на экран строку 'Привет, фуфел!' и затем завершаться.

### *7.1.2 Использование формата bin для получения .EXE файлов*

Формат .EXE является достаточно простым, поэтому построение .EXE файлов возможно путем написания чисто бинарной программы с последующим помещением в ее начало 32-битного заголовка. Структура заголовка несложная, поэтому он может быть создан обычными командами DB и DW. Исходя из вышесказанного, для непосредственного создания .EXE файлов может быть использован формат bin.

В архиве NASM имеется подкаталог misc, в котором находится файл макросов exebin.mac. В этом файле определены три макроса: EXE_begin, EXE_stack и EXE_end.

Для создания файла при помощи формата bin вы должны включить в свой исходный файл директиву %include exebin.mac, загружающую в него пакет требуемых макросов. Затем для генерации заголовка файла вы должны выполнить макрокоманду EXE_begin (не имеет аргументов). После этого следует обычный для формата bin код программы — вы можете использовать все три стандартные секции .text, .data и .bss. В конце файла вы должны вызвать макрос EXE_end (без аргументов), который для маркировки размеров секции определяет некоторые символы, ссылающиеся на заголовок кода, сгенерированный макросом EXE_begin.

В данной модели написанный вами код стартует с адреса 0x100, как и обычный .COM файл — в действительности, если вы удалите 32-битный заголовок из сгенерированного .EXE файла, то получите работающую .COM программу. Все базы сегментов в полученном файле одинаковы, поэтому размер программы ограничен 64К (опять же, как и .COM файл). Имейте в виду, что директива ORG используется макросом EXE_begin, поэтому вы не должны применять ее самостоятельно.

Вы не можете прямо ссылаться на значение базы вашего сегмента, к сожалению это потребовало бы перемещений в заголовке, что реализовать гораздо сложнее. Поэтому вы должны получать базу сегмента копированием ее из CS.

При запуске полученного .EXE файла пара SS:SP настраивается на указание вершины 2Кб стека. Вызвав макрос EXE_stack, вы можете изменить размер стека по умолчанию. Например, для изменения размера стека вашей программы до 64 байт вы должны вызвать EXE_stack 64.

В подкаталоге архива NASM содержится простая программа binexe.asm, из которой .EXE файл создается вышеописанным способом.

### 7.2 Получение .COM файлов

В то время, как большие DOS-программы должны писаться в виде .EXE файлов, небольшие часто лучше и проще написать как .COM файлы. .COM файлы являются чисто бинарными, поэтому большинство их может быть создано при помощи выходного формата bin.

### *7.2.1 Использование формата bin для получения .COM файлов*

.COM файлы загружаются в свой сегмент по смещению 100h (сегмент может меняться). Выполнение начинается с адреса 100h, т.е. программа по этому адресу стартует. Таким образом, при написании .COM программы ваш исходный файл должен выглядеть наподобие следующего:

```
          org 100h
          section .text
start:    ; сюда поместите код
          section .data
          ; сюда поместите данные
          section .bss
          ; здесь находятся неинициализированные данные
```

Формат bin помещает секцию .text в начале файла, поэтому вы можете объявлять данные или BSS перед написанием собственно кода.

Секция BSS (неинициализированные данные) не занимает места в самом .COM файле: вместо этого адреса BSS-элементов разрешаются относительно адреса конца файла, т.е. при запуске программы это пространство будет являться свободной памятью, поэтому вы не должны предполагать, что оно будет заполнено нулями или чем-либо еще — там находится просто мусор.

Для ассемблирования приведенной выше программы вы должны использовать следующую командную строку:

```
nasm myprog.asm -fbin -o myprog.com
```

Если явно не указать имя выходного файла, формат bin создаст файл с именем myprog; в этом случае вы можете просто переименовать его так, как вам нужно.

### *7.2.2 Использование формата obj для получения .COM файлов*

Если вы пишете .COM программу с применением более одного модуля, то возможно захотите ассемблировать несколько .OBJ файлов и затем собрать их в одну программу. Вы можете это сделать двумя путями: при помощи компоновщика, способного непосредственно создавать .COM файлы (TLINK это может) или применив конвертер EXE2BIN для преобразования .EXE файла, полученного на выходе компоновщика, в .COM файл.

Если вы это делаете, вам нужно позаботиться о нескольких вещах:

- Кодовый сегмент первого объектного файла должен начинаться со строки вида RESB 100h. Это гарантирует начало кода по смещению 100h относительно начала сегмента, так чтобы компоновщик или программа конвертации не корректировали адресные ссылки при генерации .COM файла. Другие ассемблеры для данной цели используют директиву ORG, однако в NASM ORG является дополнительной директивой выходного формата bin и не означает то же самое, что в MASM-совместимых ассемблерах.
- Вам не нужно определять сегмент стека.
- Все ваши сегменты должны быть в одной и той же группе, чтобы все смещения на символы как в коде, так и в данных были смещениями относительно одной и той же базы сегмента. Это нужно для того, чтобы при загрузке .COM файла все сегментные регистры содержали одно и то же значение.

## 7.3 Получение .SYS файлов

Драйверы устройств MS-DOS — .SYS файлы — это чисто бинарные файлы, во всем похожие на .COM, за исключением того, что они запускаются по нулевому смещению, а не по смещению 100h. Поэтому если вы пишете драйвер при помощи формата bin, директива ORG вам не нужна, так как по умолчанию смещение для bin всегда нулевое. Соответственно вам не требуется указывать в начале кодового сегмента RESB 100h, если вы используете формат obj.

.SYS файлы начинаются с заголовочной структуры, содержащей указатели на различные подпрограммы внутри драйвера. Данная структура должна быть определена в начале сегмента кода, несмотря на то, что в действительности кодом она не является.

Дополнительную информацию о формате .SYS файлов и данных, содержащихся в их заголовочной структуре, вы можете почерпнуть в часто задаваемых вопросах конференции comp.os.msdos.programmer.

### 7.4 Взаимодействие с 16-битными C-программами

В данном параграфе описываются основы создания ассемблерных подпрограмм, которые вызывают или которые вызываются из программ C. Для осуществления этого обычно нужно написать ассемблерный модуль в виде .OBJ файла и скомпоновать его с C-модулями.

#### *7.4.1 Внешние символьные имена*

Компиляторы C имеют соглашения, в соответствии с которыми имена всех глобальных символов (функций или данных) образуются путем префиксирования имени из C-программы символом подчеркивания. Так, например, функция, о которой C-программист думает как о printf, для программиста на ассемблере является _printf. Это означает, что в своей ассемблерной программе вы можете определять символы без лидирующего знака подчеркивания, не боясь при этом, что они случайно совпадут с именами C-символов.

Если вам неудобно использовать знаки подчеркивания, вы можете определить макросы для замены директив GLOBAL и EXTERN следующим образом:

```
%macro cglobal 1
          global _%1
%define %1 _%1
%endmacro
%macro cextern 1
          extern _%1
%define %1 _%1
%endmacro
```

(Данные формы макросов принимают только один аргумент; если вам требуется больше, используйте конструкцию %rep).

Если вы определите внешний символ как

```
          cextern printf
```

макрос развернет его в следующие строки:

```
          extern _printf
%define printf _printf
```

Thereafter, you can reference printf as if it was a symbol, and the preprocessor will put the leading underscore on where necessary.

После этого вы можете ссылаться на printf, а препроцессор, где это нужно, будет помещать ведущий знак подчеркивания. Макрос cglobal работает точно также.

#### *7.4.2 Модели памяти*

NASM прямо не поддерживает механизма различных моделей памяти, реализованных в C; вы должны отслеживать это самостоятельно. Это означает, что вы должны учитывать следующее:

- В моделях, имеющих один сегмент кода (tiny, small и compact), функции являются ближними. Это значит, что указатели на функции при сохранении в сегменте данных или помещении в стек являются 16-битными и содержат только поле смещения (регистр CS никогда не изменяет свое

значение и всегда содержит сегментную часть полного адреса функции) и что такие функции вызываются инструкцией near CALL и возврат из них производится при помощи RETN (что в NASM является синонимом RET). Следовательно, вы должны писать свои подпрограммы с использованием RETN, а также вызывать внешние С-подпрограммы при помощи ближней инструкции CALL.

- В моделях, использующих более одного сегмента кода (medium, large и huge), функции являются дальними. Это значит, что длина указателей функций составляет 32 бита (16 бит смещение и 16 бит сегмент) и что функции вызываются при помощи CALL FAR (или CALL seg:offset) и возврат из них производится при помощи RETF. При использовании таких моделей вы должны писать свои подпрограммы так, чтобы возврат из них производился по RETF, а внешние подпрограммы вызывать при помощи CALL FAR.
- В моделях, использующих единственный сегмент данных (tiny, small и medium), указатели на данные являются 16-битными, содержащими только поле смещения (регистр DS не изменяет своего значения и всегда представляет сегментную часть полного адреса).
- В моделях, использующих более одного сегмента данных (compact, large и huge), длина указателей на данные составляет 32 бит, 16 бит из которых является смещением, а другие 16 бит — сегментом. Вы должны стараться в своих подпрограммах не модифицировать DS без необходимости, а после модификации — всегда восстанавливать. В то же время регистр ES свободен и вы можете использовать его для доступа к содержимому, на которое ссылается 32-битный указатель.
- Модель памяти huge позволяет одиночным элементам данных превышать размер 64К. В любых других моделях вы можете получить доступ ко всем элементам данных простым арифметическим манипулированием переданного поля смещения (неважно, присутствует поле сегмента или нет). В модели памяти huge к вычислению указателей надо подходить более тщательно.
- В большинстве моделей памяти имеется сегмент данных *по умолчанию*, сегментный адрес которого хранится в DS на протяжении всего выполнения программы. Этот сегмент данных обычно совпадает с сегментом стека, хранящемся в SS, поэтому и локальные переменные функций (хранящиеся в стеке), и глобальные элементы данных могут быть легко доступны без изменения DS. Большие элементы данных обычно хранятся в других сегментах. Однако некоторые модели памяти (хотя обычно они нестандартные) используют SS и DS по другому. Будьте внимательны в этом случае по отношению к локальным переменным функций.

В моделях с единственным сегментом кода этот сегмент называется _TEXT, поэтому ваш сегмент должен иметь то же самое имя для компоновки его в то же место, что и основной сегмент кода. В моделях с единственным сегментом данных или с сегментом данных по умолчанию последний именуется как _DATA.

### *7.4.3 Определения и вызовы функций*

Соглашения по вызовам С в 16-битных программах описываются ниже.

- Вызывающая программа помещает параметры функции в стек один за другим в обратном порядке следования (так что первый аргумент функции помещается в стек последним).
- Вызывающая программа выполняет инструкцию CALL для передачи управления подпрограмме. Эта инструкция в зависимости от модели памяти может быть как ближней, так и дальней.
- Подпрограмма получает управление и обычно (несмотря на то, что это не требуется в функциях, которым не нужен доступ к своим параметрам) начинается с сохранения SP в BP с целью дальнейшего использования BP в качестве базы указателя для нахождения параметров в стеке. Однако частью соглашений о вызовах является сохранение содержимого BP любой функцией С. Следовательно подпрограмма, если она использует BP как указатель кадра, должна предварительно поместить в стек его содержимое.
- Подпрограмма может получить свои параметры через BP. Слово [BP] хранит предыдущее значение BP, помещенное в стек; следующее слово, [BP+2], хранит смещение адреса возврата, помещенное в стек инструкцией CALL. В ближних функциях после этого ([BP+4]) начинаются параметры; в дальных функциях по адресу [BP+4] находится сегментная часть адреса возврата и параметры начинаются с [BP+6]. Самый левый параметр функции доступен по смещению из BP, т.к. в стек он был помещен последним; следующие параметры соответственно доступны по следующим смещениям. Таким образом, в функциях с переменным числом параметров, подобных printf, помещение параметров в стек в обратном порядке означает, что функция знает, где находится ее первый параметр, сообщающий число и тип оставшихся параметров.
- Подпрограмма после этого может увеличить значение SP, например для распределения места в стеке для локальных переменных, которые после этого будут доступны по отрицательным смещениям от BP.

- Подпрограмма, если она возвращает значение вызывающей программе, должна передавать это значение в AL, AX или DX:AX в зависимости от размера последнего. Результаты, являющиеся числами с плавающей точкой иногда (в зависимости от компилятора) возвращаются в регистре сопроцессора ST0.
- Как только подпрограмма завершит свою работу, она восстанавливает значение SP из BP (если она распределяла локальное пространство стека), затем изымает из стека предыдущее значение BP и в зависимости от модели памяти возвращается в вызвавшую программу через RETN или RETF.
- Когда вызывающая программа возвратит себе управление, параметры функции остаются в стеке, поэтому для удаления их к SP обычно прибавляется непосредственная константа (вместо выполнения серии медленных инструкций POP). Поэтому если функция случайно (например, из-за несоответствия прототипов) будет вызвана с неверным числом параметров, стек при возврате останется в осмысленном состоянии, т.к. вызвавшая программа, которая *знает*, сколько параметров она поместила в стек, удалит их.

Поучительно сравнить данное соглашение о вызовах с программами на Паскале (см. параграф 7.5.1). Паскаль имеет более простое соглашение, т.к. в нем нет функций с переменным числом параметров и подпрограмма знает, сколько параметров ей передается и способна самостоятельно удалить их из стека путем указания в инструкциях RET или RETF непосредственного значения. Параметры помещаются в стек слева-направо, а не справа-налево как в C, поэтому компилятор может дать лучшую гарантию последовательности выполнения без снижения производительности.

Исходя из вышесказанного, вы можете определить C-подобную функцию следующим образом (в примере использована модель памяти small):

```
          global _myfunc
_myfunc:  push bp
          mov bp,sp
          sub sp,0x40             ; 64 байта локального пространства стека
          mov bx,[bp+4]           ; первый параметр функции
          ; некоторый код
          mov sp,bp               ; отмена "sub sp,0x40" выше
          pop bp
          ret
```

Для больших моделей памяти вы должны заменить RET в данной функции на RETF и брать первый параметр не из [BP+4], а из [BP+6]. Естественно, если один из параметров будет указателем, смещения следующих параметров будут зависеть от модели памяти: дальние указатели занимают в стеке 4 байта, в то время как короткие — два.

Если посмотреть с другой стороны, то для вызова C-функции из вашего ассемблерного кода вы должны сделать что-то наподобие следующего:

```
          extern _printf
          ; здесь идет супер-пупер-прога...
          push word [myint]       ; целое значение - параметр
          push word mystring      ; указатель на мой сегмент данных
          call _printf
          add sp,byte 4           ; 'byte' экономит размер
          ; затем следует сегмент данных...
          segment _DATA
myint     dw 1234
mystring  db 'Это число -> %d <- должно быть 1234, фуфел!^#39;,10,0
```

Этот ассемблерный код, использующий модель памяти small, эквивалентен C-коду

```
int myint = 1234;
printf("Это число -> %d <- должно быть 1234, фуфел!\n", myint);
```

В больших моделях памяти кодирование функции вызова выглядит похоже, но все-таки несколько отличается. В приведенном ниже примере подразумевается, что регистр DS уже содержит базу сегмента _DATA. Если это не так, вы должны его проинициализировать.

```
        push word [myint]
        push word seg mystring ; Теперь сохраняем в стеке сегмент, и...
        push word mystring     ; ... смещение "mystring"
        call far _printf
        add sp,byte 6
```

Целое число по прежнему будет занимать в стеке одно слово, так как большая модель памяти не влияет на размер типа данных int. В то же время первый аргумент для printf (помещаемый в стек последним), является указателем и поэтому состоит из двух частей — сегмента и смещения. Сегмент должен сохраняться в памяти вторым, поэтому в стек он помещается первым. (Конечно PUSH DS будет иметь более короткую инструкцию, чем PUSH WORD SEG mystring, если DS инициализирован так, как подразумевается в приведенном примере). Затем следует дальний вызов call far, как это определено для больших моделей памяти; после возврата из подпрограммы регистр стека увеличивается на 6 (а не на 4) с целью коррекции на размер дополнительного слова, помещенного туда ранее.

### *7.4.4 Доступ к элементам данных*

Для получения доступа к переменным C или объявления переменных, к которым C в свою очередь может обратиться, вы должны всего лишь объявить имена как EXTERN или GLOBAL соответственно. (Имена требуют лидирующего знака подчеркивания, см. параграф 7.4.1.) Таким образом, объявленная в C переменная int i может быть доступна из ассемблера как

```
        extern _i
        mov ax,[_i]
```

Чтобы объявить собственную целую переменную, к которой C-программа сможет обратиться как extern int j, вы должны сделать следующее (убедитесь, что эта переменная находится в сегменте _DATA):

```
        global _j
_j      dw 0
```

Для получения доступа к C-массиву вам нужно знать размер компонетов последнего. Например, переменные типа int имеют размер два байта (слово), поэтому если C-программа объявляет массив как int a[10], вы можете обратиться к элементу a[3] при помощи инструкции mov ax,[_a+6]. (Байтовое смещение 6 получено путем умножения индекса 3 требуемого элемента на размер элементов массива 2). Размеры базовых типов C для 16-битных компиляторов: 1 для char, 2 для short и int, 4 для long и float, 8 для double.

Чтобы получить доступ к структуре данных C, вам необходимо знать смещение интересующего вас поля от базы этой структуры. Вы можете сделать это либо преобразовав определение C-структуры в определение NASM-структуры (при помощи STRUC), либо рассчитать это смещение и использовать его "как есть".

Чтобы правильно использовать структуры C, вы должны изучить руководство по вашему C-компилятору, чтобы знать, как он организует структуры данных. NASM не делает специального выравнивания для членов его собственных структур STRUC, поэтому если C-компилятор делает это, вы можете задать такое смещение самостоятельно. Обычно вы можете предположить, что структура наподобие

```
struct {
    char c;
    int i;
} foo;
```

будет иметь длину 4 байта, а не 3, так как поле int выравнивается по двухбайтной границе. Однако такие особенности имеют тенденцию конфигурироваться компилятором C при помощи ключей командной строки, либо директив #pragma, поэтому вы должны выяснить, как именно это делает ваш компилятор.

### *7.4.5 c16.mac: Макросы для 16-битного C-интерфейса*

В подкаталоге misc архива NASM имеется файл макросов c16.mac. В нем определены три макроса: proc, arg и endproc. Они предназначены для использования в определениях C-подобных процедур и автоматизируют большинство работ по слежению за соблюдением соглашения о вызовах.

Ниже приведен пример ассемблерной функции, использующей этот набор макросов:

```
          proc _nearproc
%$i       arg
%$j       arg
          mov ax,[bp + %$i]
          mov bx,[bp + %$j]
          add ax,[bx]
          endproc
```

Здесь определяется процедура _nearproc, принимающая два аргумента, первый (i) — это целое и второй (j) — указатель на целое. Процедура возвращает i + *j.

Заметьте, что макрос arg при его разворачивании содержит в первой строке EQU, которая в результате определяет %$i как смещение от BP. При этом используются контекстно-локальные переменные (локальные к контексту, сохраняемому в контекстном стеке макросом proc и удаляемому оттуда макросом endproc), поэтому в других процедурах может быть использовано то же самое имя аргумента. Конечно, вы можете этого *не делать*.

По умолчанию представленный набор макросов создает код для ближних функций (модели памяти tiny, small и compact). Чтобы генерировать код для дальних функций (модели medium, large и huge), вы должны определить %define FARCODE. Данное определение изменяет тип возвращаемой endproc инструкции, а также начальную точку смещения аргументов. Набор макросов совершенно не зависит от того, являются ли указатели данных дальними или нет.

Макрос arg может принимать дополнительный параметр, представляющий собой размер аргумента. Если размер не задан, по умолчанию принимается 2, т.к. большинство параметров функций вероятно будут иметь тип int.

Эквивалент представленной выше функции для модели памяти large будет таким:

```
%define FARCODE
          proc _farproc
%$i       arg
%$j       arg 4
          mov ax,[bp + %$i]
          mov bx,[bp + %$j]
          mov es,[bp + %$j + 2]
          add ax,[bx]
          endproc
```

Так как j теперь будет дальним указателем, в этом примере используется аргумент макроса arg, определяющий параметр размером 4. Когда мы читаем значение по адресу j, мы должны загрузить как смещение, так и сегмент.

### 7.5 Взаимодействие с программами Borland Pascal

Взаимодействие с программами на Паскале в концепции схоже по взаимодействию с 16-битными C-программами, однако имеются следующие различия:

- Требуемые для взаимодействия с C-программами ведущие символы подчеркивания в Паскале не нужны.
- Модель памяти всегда большая: функции и указатели данных являются дальними, но длина отдельного элемента данных не должна превышать 64K. (На самом деле некоторые функции остаются ближними, но эти функции локализованы в модуле Паскаля и *никогда* не вызываются извне. Все функции ассемблера, осуществляющие Паскаль-вызовы, а также все функции Паскаля, обращающиеся к ассемблерному коду, *должны быть дальними*). В то же время все объявленные в Паскаль-программе статические данные помещаются в сегменте данных по умолчанию, т.е. его сегментный адрес при передаче управления вашему ассемблерному коду будет содержаться в DS. В сегменте данных не располагаются только локальные переменные (они находятся в сегменте стека)

и переменные, память для которых выделяется динамически. Однако все *указатели* данных являются дальними.
- Соглашение о вызовах функций отличается от C — описание приведено далее.
- Некоторые типы данных, такие как строки, хранятся по другому.
- Имеются ограничения на имена сегментов, которые вам разрешено использовать — Borland Pascal будет игнорировать код или данные, объявленные в сегменте с неподходящим именем. Эти ограничения также описаны ниже.

### *7.5.1 Соглашение о вызовах в Pascal*

Ниже описываются соглашения о вызовах в 16-битных Паскаль-программах.

- Вызывающая программа помещает параметры функции в стек один за другим в обычном порядке (слева-направо, так что первый аргумент функции помещается также первым).
- Вызывающая программа для передачи управления подпрограмме выполняет дальнюю инструкцию CALL.
- Подпрограмма получает управление и обычно (несмотря на то, что это не требуется в функциях, которым не нужен доступ к своим параметрам) начинается с сохранения SP в BP с целью дальнейшего использования BP в качестве базы указателя для нахождения параметров в стеке. Однако частью соглашений о вызовах является сохранение содержимого BP любой функцией. Следовательно подпрограмма, если она использует BP как указатель кадра, должна предварительно поместить в стек его содержимое.
- Подпрограмма может получить доступ к своим параметрам относительно BP. Слово [BP] адресует в стеке предыдущее значение BP. Следующее слово, [BP+2], адресует смещение адреса возврата, а [BP+4] — сегмент адреса возврата. Параметры начинаются со смещения [BP+6]. По этому смещению от BP доступен самый "правый" параметр функции, так как в стек он был помещен последним; следующие параметры идут соответственно по более большим смещениям.
- В процессе выполнения подпрограмма может увеличить значение SP с целью выделения в стеке места под свои локальные переменные. Эти переменные будут доступны по отрицательным от BP смещениям.
- Подпрограмма должна передавать результат выполнения назад в вызвавшую программу через AL, AX или DX:AX, в зависимости от размера значения. Результаты в виде чисел с плавающей точкой возвращаются через регистр ST0. Результаты типа Real (собственные типы Борланда — числа с плавающей точкой, прямо не обрабатываемые в FPU) возвращаются в группе DX:BX:AX. Чтобы возвратить результат типа String, вызывающая программа перед помещением в стек параметров помещает туда указатель на временную строку, а подпрограмма по этому адресу возвращает строковое значение. Указатель не является параметром и не должен удаляться из стека инструкцией RETF.
- Когда подпрограмма заканчивает свою работу, она восстанавливает содержимое SP из BP, достает из стека предыдущее значение BP и возвращается через RETF. Здесь используется форма RETF с непосредственным операндом, представляющим собой число байт, снимаемых с вершины стека в качестве параметров. Снятие параметров со стека — это побочный эффект инструкции возврата.
- Дополнительных действий от вызвавшей программы не требуется, так как параметры функции при получении ей управления уже удалены из стека.

Исходя из вышесказанного, вы должны определить функцию в стиле Паскаль, принимающую два аргумента типа Integer, следующим образом:

```
          global myfunc
myfunc:   push bp
          mov bp,sp
          sub sp,0x40             ; резервируется 64 байта в стеке
          mov bx,[bp+8]           ; первый аргумент функции
          mov bx,[bp+6]           ; второй аргумент функции
          ; код, который наверное что-то делает
          mov sp,bp               ; отмена "sub sp,0x40" выше
          pop bp
          retf 4                  ; общий размер аргументов 4
```

С другой стороны, для вызова Паскаль-функции из вашего ассемблерного кода, вы должны сделать что-то вроде следующего:

```
        extern SomeFunc
        ; тут идет какой-то код...
        push word seg mystring ; Теперь в стек помещается сегмент и...
        push word mystring     ; ... смещение строки "mystring"
        push word [myint]      ; одна из переменных
        call far SomeFunc
```

Этот код эквивалентен следующим строкам на Паскале:

```
procedure SomeFunc(String: PChar; Int: Integer);
    SomeFunc(@mystring, myint);
```

### *7.5.2 Ограничение имен сегментов в Borland Pascal*

Так как внутренний формат модуля Borland Pascal полностью отличается от OBJ, при компоновке модуля с реальным OBJ файлом выполняется очень поверхностная работа по чтению и пониманию различной информации из последнего. Вследствие этого объектные файлы, предназначенные для компоновки с Паскаль-программами, должны удовлетворять нескольким ограничениям:

- Процедуры и функции должны находиться в сегменте с именем CODE, CSEG, или заканчивающимся на _TEXT.
- Инициализированные данные должны находиться в сегменте с именем CONST или заканчивающимся на _DATA.
- Неинициализированные данные должны находиться в сегменте с именем DATA, DSEG, или заканчивающимся на _BSS.
- Любые другие сегменты, имеющиеся в объектном файле, полностью игнорируются. Директивы GROUP и атрибуты сегментов также игнорируются.

### *7.5.3 Использование c16.mac с Pascal-программами*

Пакет макросов c16.mac, описанный в параграфе 7.4.5, может быть также использован для облегчения написания функций, вызываемых из программ на Паскале. Для этого вам нужно определить %define PASCAL. Данное определение делает все функции дальними (это подразумевает FARCODE), а также генерирует инструкции возврата из подпрограммы в форме, имеющей операнд.

Определение PASCAL не изменяет код, рассчитывающий смещения аргументов; вы должны объявлять аргументы вашей функции в обратном порядке. Например:

```
%define PASCAL
        proc _pascalproc
%$j     arg 4
%$i     arg
        mov ax,[bp + %$i]
        mov bx,[bp + %$j]
        mov es,[bp + %$j + 2]
        add ax,[bx]
        endproc
```

Концептуально здесь определяется та же самая подпрограмма, что и в параграфе 7.4.5: функция принимает два аргумента, целое число и указатель на целое и возвращает сумму целого и содержимого, на которое ссылается указатель. Отличие между этим кодом и версией C для большой модели памяти состоит в том, что вместо FARCODE определяется PASCAL, а аргументы объявляются в обратном порядке.

## *Глава 8: Написание 32-х битного кода (Unix, Win32, DJGPP)*



Эта глава повествует о наиболее распространенных проблемах, возникающих при написании 32-ух разрядного кода для Win32 или Unix, или для сборки (linking) с Си-кодом, полученным компилятором Си Unix-стиля, таким как DJGPP. Здесь также рассматривается как писать ассемблерный код для

взаимодействия (interface) с кодом, полученным 32-ух битным компилятором с Си и как создавать перемещаемый код для разделяемых библиотек.

Почти весь 32-ух битный код и практически весь код, выполняемый под Win32, DJGPP или под любой вариант Unix для ПК выполняется в *плоской* (*flat*) модели памяти. Это означает, что сегментные регистры и механизм страничной адресации уже установлены заранее для того, чтобы дать вам одно и то же 32-битное адресное пространство в 4 Гб, не зависимо относительно какого сегмента вы работаете, и поэтому вы должны полностью игнорировать (не использовать) все сегментные регистры. Когда вы пишите приложение под плоскую модель памяти, вам никогда не потребуется замещение сегментов или изменение значения каких-либо сегментных регистров, и адреса сегмента кода, которые вы передаете инструкциям CALL и JMP остаются в том же адресном пространстве, что и адреса сегмента данных через которые вы обращаетесь к переменным и адреса сегмента стека, которые вы используете для доступа к локальным переменным и параметрам процедур. Каждый адрес имеет размер 32 бита и содержит только смещение (offset-ную) часть адреса.

### 8.1 Интерфейс с 32-ух битными программами на Си

Все рассуждения в параграфе 7.4, относящиеся к интерфейсу с 16-ти битными программами на Си также применимы и к 32-ух битным. Отсутствие моделей памяти или сегментации не должно вызывать у вас беспокойства.

#### *8.1.1 Внешние символьные имена*

Большинство 32-ух битных Си-компиляторов поддерживают конвенцию, используемую в 16-ти битных компиляторах: имена всех глобальных имен (функций или переменных) они определяют префиксом “подчеркивание” (например: extrn_link в Си будет _extrnlink в линкуемом файле), добавляемое к имени, действующем в Си-программе. Правда, не все это делают: спецификация ELF указывает, что все имена в Си-программе *не имеют* предваряющего подчеркивания в их эквивалентах на языке ассемблера.

Старый Си-компилятор в Линуксе a.out, все компиляторы для Win32, DJGPP, NetBSD и FreeBSD, все они используют предваряющее “подчеркивание”; для этих компиляторов макросы cextern и `cglobal`, как они описаны в параграфе 7.4.1, будут работать. Для ELF, разумеется, предваряющее подчеркивание не используется.

#### *8.1.2 Определение и вызов функций*

Конвенция Си для вызова в 32-ух битных программах приведена ниже. В этом описании использованы выражения *вызывающий код* и *вызываемая функция* для того, чтобы указать делает ли функция вызов, или функция получает управление.

- Вызывающий код проталкивает параметры в стек один за другим в обратном порядке (справа налево, таким образом, первый объявленный аргумент функции будет помещен в стек последним).
- Затем вызывающий код выполняет ближний вызов, чтобы передать управление вызываемой функции.
- Вызываемая функция, получая управление, и обычно (хотя это не всегда необходимо в функциях, которые не обращаются к своим параметрам) начиная с сохранения значений ESP в EBP,чтобы можно было использовать EBP как базовый указатель для доступа к параметрам в стеке. Однако, вызывающий код возможно делает тоже самое (устанавливает EBP на свой стек), поэтому по этой части конвенции о вызове, состояние EBP должно быть сохранено во всех Си-функциях. Поэтому вызываемая функция, если она хочет установить EBP как указатель на свои параметры должна сначала сохранить в стеке его предыдущее значение.
- Вызываемая функция может обращаться к своим параметрам относительно EBP. Двойное слово по адресу [EBP] содержит предыдущее значение EBP, т.к. оно было сохранено в стеке; следующее двойное слово по адресу [EBP+4] содержит адрес возврата, протолкнутый туда инструкцией CALL. После этого начинаются параметры — с адреса [EBP+8]. Самый левый параметр функции, т.к. он был помещен в стек последним, доступен по этому смещению от EBP; остальные расположены по бОльшим смещениям. Поэтому, в таких функциях как printf, которые имеют переменное число параметров, заталкивание параметров в обратном порядке позволяет функции узнать где находится первый параметр, в котором содержится информация о количестве и типах остальных.

- Вызываемая функция также может уменьшить значение ESP для того, чтобы зарезервировать место в стеке для локальных переменных, которые будут доступны как отрицательные смещения относительно EBP.
- Вызываемая функция, если она хочет вернуть значение вызывающему коду, должна оставить это значение в AL, AX или EAX, в зависимости от размера. Значения с плавающей запятой обычно возвращаются в ST0.
- Как только вызываемая функция закончила свои основные действия, она восстанавливает ESP из EBP если она резервировала место в стеке, затем выталкивает предыдущее значение EBP, и возвращает управление через RET (эквивалентно, RETN).
- Когда вызывающий код снова получает контроль от вызываемой функции, параметры функции все еще остаются в стеке, поэтому обычно к значению ESP прибавляется непосредственное значение, чтобы убрать их (вместо выполнения нескольких медленных инструкций POP). Поэтому, если функция случайно будет вызвана с неверным числом параметров, т.е. не соответствующем прототипу, стек будет возвращен вызывающим кодом в то состояние, в котором он находился до вызова этой функции, который *знает* сколько параметров было помещено в стек, и удаляет их оттуда.

Существует альтернативная конвенция вызова, используемая в Win32 программах для вызовов через Windows API, а также для функций, вызываемых Windows API, таких как оконные процедуры (window procedures): они описаны так, чтобы использовать Microsoft __stdcall конвенцию. Это очень похоже на конвенцию Паскаля, в вызываемой функции очищаются параметры из стека, используя параметр с инструкцией RET. Однако, параметры также передаются справа налево.

Если вы определите функцию в стиле языка Си следующим образом:

```
          global _myfunc
_myfunc:  push ebp
          mov ebp,esp
          sub esp,0x40            ; 64 байта для локальных переменных
          mov ebx,[ebp+8]         ; Первый параметр функции
          ; еще какой-нибудь код
          leave                   ; mov esp,ebp / pop ebp
          ret
```

С другой стороны, чтобы вызывать Си-функцию из вашего ассемблерного кода, вы должны сделать что-то вроде этого:

```
          extern _printf
          ; и затем...
          push dword [myint]      ; одна из моих переменных целого типа
          push dword mystring     ; указатель в моем сегменте данных
          call _printf
          add esp,byte 8          ; `byte' уменьшит размер кода
          ; и теперь объявления данных...
          segment _DATA
myint     dd 1234
mystring  db 'Это число -> %d <- должно быть 1234',10,0
```

Этот фрагмент кода – ассемблерный эквивалент Си-кода

```
int myint = 1234;
printf("Это число -> %d <- должно быть 1234\n", myint);
```

### *8.1.3 Доступ к переменным*

Чтобы получить доступ к Си-переменным, или чтобы объявить переменные, к которым может обращаться Си, вам достаточно объявить имена как GLOBAL или EXTERN. (Опять же, имена нужно предварять подчеркиванием, как это описано в параграфе 8.1.1.) Таким образом, Си-переменные, объявленные как int i могут быть доступны из ассемблера как

```
          extern _i
```

```
        mov eax,[_i]
```

И чтобы объявить ваши собственные переменные, которые будут доступны Си-программе как int j, делайте это так (проверьте, что вы ассемблируете это в _DATA сегменте, если это необходимо):

```
        global _j
_j      dd 0
```

Чтобы обращаться с Си-массивами необходимо знать размер элементов этого массива. Например, int переменные имеют размер 4 байта, поэтому если в Си-программе объявлен массив как int a[10], можно обращаться к a[3] так: mov ax,[_a+12]. ( Смещение 12 полчается в результате умножения номера в массиве, 3 на размер элемента массива, 4.) Размеры базовых типов в 32-х разрядных Си-компиляторах: 1 для char, 2 для short, 4 для int, long и float, и 8 для double. Указатель, содержащий 32-ух битный адрес имеет также размер 4 байта.

Чтобы обращаться к структурам языка Си, необходимо знать смещение от базового адреса структуры до интересующего вас поля. Вы можете просто это делать, конвертируя определения Си-структур в определения структур NASM (STRUC), или вычисляя одно смещение и используя только его.

Чтобы делать это проще, вам необходимо причитать руководство по вашему компилятору языка Си и найти как он организует структуры данных. NASM не делает специальных выравниваний членов структуры в его макросе STRUC, поэтому вы должны указывать выравнивания самостоятельно, если Си-компилятор генерирует их. Обычно вы можете обнаружить, что подобная структура

```
struct {
    char c;
    int i;
} foo;
```

имеет размер не 5 байт, а 8, так как int поле будет выровнено на границу двойного слова. Однако, эту возможность иногда можно настроить в Си-компиляторе, используя параметры командной строки или #pragma — директивы, поэтому можно найти как ваш компилятор это делает.

### *8.1.4 c32.mac: Вспомогательные макросы для 32-ух битного интерфейса с Си*

Файл макроса c32.mac включен в архив NASM, в каталог misc. В этом файле определены 3 макроса: proc, arg и endproc. Они введены для использования в определении процедур в стиле Си, и она автоматизируют операции, необходимые для соблюдения конвенции вызова.

Пример ассемблерной функции, использующей этот набор макросов приведен ниже:

```
        proc _proc32
%$i     arg
%$j     arg
        mov eax,[ebp + %$i]
        mov ebx,[ebp + %$j]
        add eax,[ebx]
        endproc
```

Этот фрагмент определяет _proc32 как процедуру с двумя аргументами, первый (i) является integer и второй (j) является указателем на integer. Процедура возвращает i + *j.

Заметьте, что макрос arg при его разворачивании содержит в первой строке EQU, которая в результате определяет %$i как смещение от BP. При этом используются контекстно-локальные переменные (локальные к контексту, сохраняемому в контекстном стеке макросом proc и удаляемому оттуда макросом endproc), поэтому в других процедурах может быть использовано то же самое имя аргумента. Конечно, вы можете этого *не делать*.

arg можно передать необязательный параметр, указывающий размер аргумента. Если размер не задан, он предполагается равным 4-ем, потому что абсолютное большинство параметров функции будут типа int или указателями.

### 8.2 Написание разделяемых библиотек для NetBSD/FreeBSD/OpenBSD и Linux/ELF

ELF замещает старый объектный формат a.out для Линукса, потому что он поддерживает перемещаемый код (position-independent code — PIC), который позволяет писать разделяемые библиотеки намного проще. NASM поддерживает особености перемещаемого кода для ELF, поэтому вы можете писать разделяемый библиотеки для Линукс ELF на NASM.

NetBSD, и его близкие родственники FreeBSD и OpenBSD, используют другой подход, добавив поддержку перемещаемого кода в формат a.out. NASM поддерживает это как формат aoutb для скомпилированных файлов, поэтому вы можете писать разделяемые библиотеки для BSD на NASM тоже.

Операционная система загружает разделяемую PIC (пермещаемый код) библиотеку, делая отображение в память файла библиотеки в произвольно выбранное место в адресном пространстве выполняемого процесса. Поэтому содержимое секции кода библиотеки должно не зависеть от места в памяти, куда она загружена.

Поэтому, вы не можете обращаться к вашей переменной таким вот способом:

```
mov eax,[myvar]         ; ОШИБКА
```

Вместо этого, линковщик предоставляет область памяти, называемую *глобальной таблицей смещений (global offset table)*, или просто ГТС (GOT); ГТС расположена на постоянном расстоянии от кода вашей библиотеки, поэтому если вы сможете узнать куда загружена ваша библиотека (что обычно осуществляется комбинацией CALL и POP), вы сможете получить адрес ГТС, и затем загрузить адрес вашей переменной из сгенерированной линковщиком записи в ГТС.

Секция *data* PIC (перемещяемый код) разделяемой библиотеки не имеет подобных ограничений: поскольку секция данных доступна для записи, она может быть скопирована в память каким бы то ни было образом, не только отображением в страницы из файла библиотеки, поэтому как только она будет скопирована, она может быть также перемещена. Поэтому вы можете обычный способ для доступа в секции данных, особо не заботясь об этом (но посмотрите в параграфе 8.2.4 предостережения).

#### *8.2.1 Получение адреса ГТС*

Каждый фрагмент кода в вашей разделяемой библиотеке должен определить ГТС как внешнее имя:

```
extern _GLOBAL_OFFSET_TABLE_   ; в ELF
extern __GLOBAL_OFFSET_TABLE_  ; в BSD a.out
```

В начале каждой функции вашей разделяемой библиотеки, которая собирается обращаться к вашим секциям data или BSS, должен вычисляться адрес ГТС. Это обычно осуществляется написанием функции в такой форме:

```
func:     push ebp
          mov ebp,esp
          push ebx
          call .get_GOT
.get_GOT: pop ebx
          add ebx,_GLOBAL_OFFSET_TABLE_+$$-.get_GOT wrt ..gotpc
          ; Тело функции начинается отсюда
          mov ebx,[ebp-4]
          mov esp,ebp
          pop ebp
          ret
```

(Для BSD, снова, имя _GLOBAL_OFFSET_TABLE нужно предварить вторым знаком подчеркивания.)

Первые две сточки этой функции — это просто стандартное начало для Си, чтобы установить стековый кадр, и последние три строчки — стандартное завершение Си-функции. Третья строка, и с четвертой по последнюю сктроки, сохраняют и восстанавливают регистр EBX, потому что PIC (перемещаемый код) разделяемая библиотека использует этот регистр для сохранения адреса ГТС.

Интересный фрагмент это инструкция CALL и следующие за ней две строки. Комбинация CALL и POP получают адрес метки .get_GOT, без узнавания в точности куда загружена программа (потому что инструкция CALL кодируется относительно текущей позиции). Инструкция ADD позволяет использовать специальный PIC (перемещаемый код) тип размещения: GOTPC размещение. Со спецификатором WRT ..gotpc размещение имен (здесь _GLOBAL_OFFSET_TABLE_, специальное имя, связанное с ГТС) дается как смещение от начала секции. (Вобще-то, ELF кодирует это как смещение от поля операндов инстуркции ADD, но NASM нарочно это упрощает, поэтому этот метод подходит для обоих ELF и BSD.) Таким образом, затем инструкция *добавляет* начало секции, чтобы получить настоящий адрес ГТС, и вычитает значение .get_GOT которое находится в EBX. Поэтому, в то время, когда инструкция завершится, EBX содержит адрес ГТС.

Если вы не следили за рассуждениями, не беспокойтесь: никогда не приходится получать адрес ГТС другими способами, поэтому вы можете поместить эти три инструкции в макрос и не обращать на них внимания:

```
%macro get_GOT 0
          call %%getgot
%%getgot: pop ebx
          add ebx,_GLOBAL_OFFSET_TABLE_+$$-%%getgot wrt ..gotpc
%endmacro
```

### *8.2.2 Нахождение ваших локальных переменных*

Имея ГТС, вы можете использовать ее для получения адресов ваших переменных. Большинство переменных будут постоянно находится в секциях, которые вы объявили; к ним можно получить доступ, используя ..gotoff специальный тип WRT. Вот как это работает:

```
          lea eax,[ebx+myvar wrt ..gotoff]
```

Выражение myvar wrt ..gotoff вычисляется, затем разделяемая библиотека линкуется, чтобы оно стало смещением локальной переменной myvar от начала ГТС Поэтому, прибавление его к EBX, как это показано выше, помещает настоящий адрес myvar в EAX.

Если объявить переменные как GLOBAL без указания их размера, они разделяются между фрагментами кода в библиотеке, но не могут быть экспортированы из библиотеки в программу, которая ее загрузила. Они будут по прежнему в вашей обычных секциях data и BSS, поэтому доступ к ним можно получить таким же методом, что и к локальным переменных, используя описанный ранее механизм ..gotoff.

Обратите внимание, что из-за специфики в BSD a.out формата описатели этого перемещаемого типа, должно существовать хотя бы одно нелокальное имя в той же секции, что и адрес, к которому вы обращаетесь.

### *8.2.3 Нахождение внешних и общих переменных*

Если вашей библиотеки требуется получить внешнюю переменную (внешнюю по отношению к *library*, не только к одному модулю в ней), необходимо использовать тип ..got, чтобы этого добиться. Тип ..got, вместо того, чтобы давать смещение от базы ГТС до переменной, дает смещение от базы ГТС до *элемента* ГТС, содержащего адрес переменной. Линковщик настраивает этот элемент ГТС когда делает библиотеку, и динамический линковщик помещает правильный адрес во время загрузки. Таким образом, чтобы получить адрес внешней переменной extvar в EAX, вам понадобиться написать:

```
          mov eax,[ebx+extvar wrt ..got]
```

Эта строчка загружает адрес extvar из элемента ГТС. Линковщик, когда он делает разделяемую библиотеку, собирает вместе все перемещения типа ..got, и создает ГТС, поэтому это обеспечивает существование каждого необходимого элемента.

Общие переменные должны использоваться таким же образом.

### *8.2.4 Экспортирование имен в библиотеку пользователя*

Чтобы экспортировать имена пользователю библиотеки, необходимо объявить являются ли они функциями или данными, и если это данные, необходимо указать размер переменной. Это нужно для того, чтобы динамический линковщик создает таблицу связей входов процедур для каждой экспортированной функции, а также перемещает экспортированные переменные из секции данных библиотеки в которой они были объявлены.

Таким образом, чтобы экспортировать функцию в библиотеку пользователя, необходимо использовать

```
          global func:function   ; объявить это как функцию
func:     push ebp
          ; etc.
```

А чтобы экспортировать переменные, такие как массивы, понадобится такой код

```
          global array:data array.end-array ; и указать размер...
array:    resd 128
.end:
```

Будьте осторожны: если экспортировать переменную в библиотеку пользователя, объявив ее как GLOBAL и указать ее размер, переменная окажется в сегменте данных главной программы, вместо сегмента данных вашей библиотеки, в котором объявлена. Поэтому получать доступ к вашей глобальной переменной вам нужно, используя механизм ..got, вместо ..gotoff, как если бы она была внешней (которая, вобщем, таковой и стала).

В равной мере, если необходимо сохранить адрес экспортированной глобальной переменной в вашем сегменте данных, вам ен удастся это сделать стандартным образом:

```
dataptr:  dd global_data_item    ; ОШИБКА
```

NASM интерпретирует этот код как обычное резервирование, в котором global_data_item просто смещение от начала сегмента .data (или чего-то другого); поэтому это определение будет указывать на ваш сегмент данных, вместо экспортирования глобальной переменной, которая находится в другом месте.

Вместо вышеописанного кода, тогда, используйте

```
dataptr:  dd global_data_item wrt ..sym
```

что использует специальный тип WRT ..sym, чтобы указать NASM-у искать в таблице имен особое имя для этого объявления, место простого определения относительно базы сегмента.

Тотже метод будет работать и для функций: объявление одной из ваших функций следующим образом

```
funcptr:  dd my_function
```

предоставит пользователю адрес вашего кода, тогда как

```
funcptr:  dd my_function wrt ..sym
```

предоставит адрес процедуры сборки (линковки) таблицы для этой функции, который вызывающая прогамма будет полагать как местоположение функции. Получение этого адреса — правильный способ вызова функции.

### *8.2.5 Вызов процедур вне библиотеки*

Вызов процедур извне вашей разделяемой библиотеки может быть осуществлен через *таблицу сборки процедуры(procedure linkage table)*, или ТСП(PLT). ТСП помещается по известному смещению от того места, куда загружена библиотека, поэтому библиотечный код может делать вызовы к ТСП в переносимом (не зависимого от места) стиле. Внутри ТСП есть код для перехода на смещения, расположенные в ГТС, поэтому вызовы из функции к другим разделяемым библиотекам или к коду в главной программе могут быть незаметно направлены на их реальные местоположения.

Чтобы вызывать внешний код, необходимо использовать другой специальный PIC (переносимый код) тип переноса, WRT ..plt. Это намного проще, чем использовать ГТС: просто замещается вызовы, например, такие CALL printf на версию, использующую ТСП — CALL printf WRT ..plt.

### *8.2.6 Создание библиотечного файла*

Написав несколько модулей с кодом и ассемблировав их в .o файлы, вы создаете вашу разделяемую библиотеку командами наподобие таких:

```
ld -shared -o library.so module1.o module2.o       # for ELF
ld -Bshareable -o library.so module1.o module2.o   # for BSD
```

Для ELF, если вашу разделяемую библиотеку предполагается использовать в системных каталогах, таких как /usr/lib или /lib, это обычно делают, используя параметр -soname при сборке (линковке), чтобы сохранить конечный библиотечный файл с именем, с номером версии в библиотеке:

```
ld -shared -soname library.so.1 -o library.so.1.2 *.o
```

А вы потом скопируете файл library.so.1.2 в библиотечный каталог, и создадите символьную ссылку library.so.1 к нему.

## *Глава 9: Смешивание 16- и 32-битного кода.*

Перевод: AsmOS group, © 2001

Эта глава повествует о некоторых проблемах, связанных с использованием необычных способов адресации и инструкций перехода, возникающих при написании кода операционных систем, таких мест как инициализация защищенного режима, которые требуют, чтобы код, который оперирует с сегментами смешанных адресаций, например, код из 16-битного сегмента пытается модифицировать данные в 32-битном, или инструкции перехода между сегментами разной разрядности.

### 9.1 Переходы между сегментами смешанной разрядности

Наиболее распространенные формы инструкций смешанной разрядности встречаются при написании 32-битной ОС: как только закончился ваш код настройки в 16-битном режиме, например, загрузка ядра, вам придется запускать его, переключаясь в защищенный режим и прыгая в 32-битный сегмент на точку входя ядра. В полностью 32-биной ОС, есть потребность в использовании только инструкций смешанной разрядности, потому что любой код до ее создания может быть выполнен как чистый 16-битный, и все написанное для нее после, может быть чистым 32-битным.

Пусть переход должен быть осуществлен по 48-битному дальнему адресу, так как сегмент, в который происходит переход — 32-битный. Однако, он должен быть ассемблирован в 16-битном сегменте, поэтому, если мы для примера напишем,

```
jmp 0x1234:0x56789ABC  ; ошибка!
```

то это будет неправильно работать, так как смещение будет обрезано до 0x9ABC и переход окажется обычным 16-битным.

В коде инициализации ядра Линукса, из-за неспособности as86 генерировать необходимые инструкции, их кодируют вручную, используя директиву DB. NASM способен сделать это лучше, он действительно самостоятельно генерирует правильный код. Вот как это делается правильно:

```
jmp dword 0x1234:0x56789ABC  ; правильно
```

Префикс DWORD (строго говоря, он должен следовать *после* двоеточия, так как это объявление размера двойного слова для смещения, но NASM поддерживает и такую форму записи, потому что обе они не двусмысленны) сообщает, что смещение должно рассматриваться как дальнее, в предположении, что вы умышленно совершаете переход их 16-битного сегмента в 32-битный.

Вы можете сделать обратное действие, делая переход из 32-битного сегмента в 16-битный, указав префикс WORD:

```
jmp word 0x8765:0x4321 ; 32 to 16 bit
```

Если префикс WORD указан в 16-битном режиме, или префикс DWORD в 32-битном, они будут проигнорированы, потому что они переводят NASM в режим, в котором он и так находится.

**9.2 Адресация между сегментами различной разрядности**

Если ваша ОС является смесью 16 и 32-битной, или если вы пишете расширитель ДОС, наверняка захотите использовать несколько 16-битных и несколько 32-битных сегментов. Но с другой стороны, вам придется писать код в 16-битном сегменте, который обращается к данным в 32-битном сегменте, или наоборот.

Если данные в 32-битном сегменте расположены в передлах первых 64К сегмента, то к ним можно обращаться, используя обычные 16-битные операции, но рано или поздно, вам понадобится совершать 32-битную адресацию из 16-битного сегмента.

Самый легкий путь сделать это, убедится, что вы используете регистр для адреса, с этих пор каждый эффективный адрес, содержащийся в 32-битном регистре принудительно будет 32-битным адресом. Поэтому вы можете сделать так

```
mov eax,offset_into_32_bit_segment_specified_by_fs
mov dword [fs:eax],0x11223344
```

Это хорошо, но немного громоздко (потому что мы проигрываем инструкцию и регистр) если вам уже известно точное смещение. Архитектура x86 поддерживает 32-битную эффективную адресацию, чтобы указать только 4-байтное смещение, поэтому почему NASM не мог бы генерировать инструкцию лучше для этих целей?

Он может. Как описано в параграфе 9.1, понадобится только предварить адрес ключевым словом DWORD, и это будет 32-битным адресом:

```
mov dword [fs:dword my_offset],0x11223344
```

Еще, как описано в параграфе 9.1, NASM не придирчив к использованию префикса DWORD, он может идти до или после замещения сегмента, поэтому неплохо будет выглядеть код с этой инструкцией вот так:

```
mov dword [dword fs:my_offset],0x11223344
```

Не удивляйтесь, что префикс DWORD *вне* квадратных скобок, он контролирует размер данных, сохраняемых по этому адресу, а один внутри квадратных скобок, который указывает на длину адреса. Последнее можно очень просто продемонстрировать:

```
mov word [dword 0x12345678],0x9ABC
```

Это объявление помещает 16 бит данных по адресу, указанному по 32-битному смещению.

Вы также можете указать префикс WORD или DWORD месте с префиксом FAR для косвенных дальних переходов или вызовов. Например:

```
call dword far [fs:word 0x4321]
```

Эта инструкция содержит адрес, указанный 16 битным смещением; она загружает 48-битный дальний указатель из него (16-битного сегмента и 32-битного смещения), и вызывает этот адрес.

### 9.3 Другие инструкции смешанного размера

Другой способ, который вы можете использовать для доступа к данным, является применение инструкций для работы со строками (LODSx, STOSx и так далее) или инструкции XLATB. Поскольку эти инструкции не имеют параметров, может показаться, что нет простого способа заставить их работать с 32-битными адресами из 16-битного сегмента.

Как раз для этого и предназначены префиксы a16 и a32 NASM-а. Если вы пишите LODSB в 16-битном сегменте, но предполагаете обращаться к строке из 32-битного сегмента, загрузите адрес в ESI и затем напишите

```
a32 lodsb
```

Этот префикс даставит использовать 32-битную адресацию, это означает, что LODSB загружает из [DS:ESI] вместо [DS:SI]. Чтобы получить доступ к строке в 16-битном сегменте из 32-битного, можно использовать соответствующий префикс a16.

Префиксы a16 и a32 могут быть применимы к любой инструкции из таблицы инструкций NASM-а, но для большинства из них создаются необходимые формы адресации и без них. Эти префиксы необходимы только для инструкций с неявной адресацией: CMPSx (параграф A.24), SCASx (параграф A.229), LODSx (параграф A.117), STOSx (параграф A.243), MOVSx (параграф A.137), INSx (параграф A.98), OUTSx (параграф A.149) и XLATB (параграф A.269). Также, различные инструкции push и pop (PUSHA и POPF также как и более частоиспользуемые PUSH и POP) могут использоваться с префиксами a16 и a32 для выбора SP или ESP для использования в качестве указателя стека, в случае, если размер сегмента стека имеет разрядность, отличную от кодового сегмента.

PUSH и POP, когда они используются с сегментными регистрами в 32-битном режиме, также имеют немного необычное поведение, когда они помещают в стек и выталкивают из него 4 байта за один раз, из них два верхних игнорируются и нижние два используются как значения сегментых регистров, с которыми вызывались инструкции. Чтобы зафиксировать 16-битное поведение инструкций push и pop для операций с сегментными регистрами, вы можете использовать префикс размера операнда o16:

```
o16 push ss
o16 push ds
```

Этот код сохраняет двойные слово стекового пространства, запихивая два сегментных регистра в пространство, которое при нормальных обстоятельствах используется только под одну операцию помещения в стек.

(Вы также можете использовать префикс o32 для указания 32-битного поведения, когда вы в 16-битном режиме, но это оказывается менее полезно.)

## *Глава 10: Разрешение проблем*

Перевод: AsmOS group, © 2001

Эта глава описывает некоторые из проблем, с которыми, насколько известно, обычно сталкиваются пользователи NASM, и методы их преодоления. Также здесь описан способ сообщения об ошибках NASM команде его разработчиков, в случае, если вы столкнетесь с трудностями, не описанными в данной главе.

### 10.1 Общие проблемы

#### *10.1.1 Генерация NASM неэффективного кода*

Я получаю много сообщений об ошибках, в которых говорится, что NASM генерирует неэффективный или неверный (при использовании инструкций типа ADD ESP, 8) код. Это преднамеренная особенность дизайна, связанная с предсказуемостью вывода: видя инструкцию типа "ADD ESP, 8", NASM генерирует команду,

оставляющую место для 32-битного смещения. Если вы хотите использовать оптимизированную по размеру инструкцию, то должны использовать "ADD ESP, BYTE 8". Это не ошибка, это просто моя точка зрения.

### *10.1.2 Мои "JUMPы" вне диапазона*

Также люди жалуются, что когда они используют переходы по условию (которые являются типа SHORT по умолчанию), которые пытаются перейти слишком далеко, то NASM сообщает "SHORT JUMP OUT OF RANGE", вместо использования более длинного перехода.

Это снова частичная проблема предсказуемости, имеющая также практическую причину. NASM не имеет возможности узнать, для какого типа процессора будет использоваться этот код, так что он не может решить сам использовать переход типа NEAR, потому что не знает, что программа предназначается для процессора 386 и выше. Аналогично, NASM может заменить инструкцию JNE типа SHORT, которая пыталась бы выйти за диапазон, на более короткую команду JE, использующей JUMP NEAR — это разумное решение для процессоров ниже 386, но не эффективное для процессоров, имеющих хорошее предсказание перехода и в состоянии осуществить JMP NEAR. Так что, повторяю — это пользователю, а не ассемблеру решать, какие команды должны быть сгенерированы.

Люди, пишущие загрузочный сектор в двоичном формате, часто жалуются, что ORG работает не так, как им хотелось бы: чтобы поместить сигнатуру 0xAA55 в конец 512-ти байтного загрузочного сектора, те, кто привык к MASM, имеют тенденцию делать следующее

```
ORG 0
; код загрузчика
ORG 510
DW 0xAA55
```

В NASM ORG не предназначена для этого. Правильный способ заключается в использовании директивы TIMES:

```
ORG 0
; код загрузчика
TIMES 510-($-$$) DB 0
DW 0xAA55
```

Директива TIMES вставит необходимое количество нулей в код вплоть до 510-го байта. Еще одно преимущество этого метода в том, что если вы выйдете за допустимые размеры загрузочного сектора, то NASM перехватит эту ошибку еще во время ассемблирования и сообщит об этом, так что вам не придется "доводить" программу дизассемблированием и поиском ошибки.

### *10.1.4 TIMES не работает*

Другая обычная проблема с вышеупомянутым кодом бывает у тех, кто пишет

```
TIMES 510-$ DB 0
```

полагая, что $ - это просто число (как и 510), и что разница тоже будет числом и эту разницу можно спокойно преподнести TIMES.

NASM — *модульный* ассемблер: различные составляющие части разработаны так, чтобы их было легко повторно использовать по отдельности, так что они не обмениваются информацией без необходимости. В последствии, даже зная, что выходной двоичный формат имеет ORG 0 (и секция .text будет начинаться с 0), эта информация не поступит назад к оценщику выражений. Так что, с точки зрения оценщика, $ — не просто число, а адрес, включающий смещение от базы (основы) сегмента. Поэтому разница между $ и 510 тоже будет включать значение смещения от базы сегмента (0x1234:0x5678 вместо 0x5678, например). Значение, включающее смещение, нельзя передавать в качестве параметра директиве TIMES.

Решение можно увидеть в предыдущем примере:

```
TIMES 510-($-$$) DB 0
```

Здесь $ и $$ содержат смещения относительно одного и того же сегмента, так что их разница — простое число. Это решит проблему и будет сгенерирован правильный код.

### 10.2 Дефекты (ошибки)

Мы пока еще не выпускали ни одной версии NASM с дефектами, о которых мы *знаем*. Хотя это и не исключает наличия множества таких дефектов, о которых мы просто не знаем. О любом, найденном вами, вы должны сообщить по адресу hpa@zytor.com.

Прежде чем сообщать о дефекте, пожалуйста, внимательно прочитайте параграф 2.2, там перечислены преднамеренно созданные особенности, которые можно принять за ошибки в NASM. (Если же что-то не описано там и кажется вам ошибкой, то будем рады получить ваши подробные комментарии, а не просто письмо типа "Это - ошибка"). После этого прочитайте параграф 10.1, и не сообщайте об ошибке, если она уже там описана.

Если вы создаете отчет об ошибке, *пожалуйста*, сообщите нам всю нижеследующую информацию:

- Под какой ОС вы запускаете NASM: DOS, Linux, NetBSD, Win16, Win32, VMS (я бы удивился :) ), какая-то еще.
- Если вы запустили NASM из-под DOS или Win32, то сообщите, какой из компиляторов вы используете — стандартный из дистрибутива или откомпилированный вами. Если это "ваш" компилятор, то попробуйте воссоздать ошибку на "родном" из дистрибутива. Эта информация поможет нам быстрее исправить ошибку.
- Какую версию NASM вы используете и подробности того, как вы вызвали сбой. Сообщите нам полностью командную строку и переменные окружения NASM (если есть).
- Какие версии других программ вы используете и как вы их вызываете. Если сбой возникает во время линковки, то сообщите версию и название компоновщика, его командную строку. Если возникает проблема при линковке с объектным файлом, созданным другим компилятором, то сообщите нам, что за компилятор, версию и командную строку или опции, использованные вами. (Если вы использовали IDE, то попробуйте воспользоваться версией компилятора для командной строки).
- Если это возможно, то вышлите нам исходный файл, вызывающий ошибку. Если же могут возникнуть проблемы связанные с авторским правом (например, вы можете воспроизвести ошибку в отдельном файле проекта), тогда имейте в виду следующее: во-первых, весь исходный код, высланный к нам, будет использоваться *только* для обнаружения и исправления ошибки NASM, и мы удалим все копии у себя; во-вторых, мы предпочли бы *не* получать большие куски кода. Чем меньше файл — тем лучше. Гораздо легче работать с файлом из трех строчек кода, *показывающего* ошибку, чем с полнофункциональной программой из десятков тысяч строк кода. (Конечно же, некоторые ошибки могут возникнуть в *только* большем файле, мы это понимаем).
- Точное описание смысла проблемы. Просто "Это не работает" *не* сможет помочь! Пожалуйста, объясните точно - что вы хотели получить и не получили (или наоборот - получили то, чего не должно было быть). Например: "NASM показывает ошибку в 3-ей строке, а она в 5-ой"; "NASM сообщает об ошибке, а я *уверен*, что ее нет"; "NASM не сообщил об ошибке, которая точно есть"; "Объектный файл, созданный из этого кода, не подходит моему компоновщику"; "Девятый байт выходного файла равен 66, а я думаю, что должно быть 77".
- Если вы уверены, что выходной файл не верен, то пошлите его нам. Это поможет нам определить, создаст ли наша собственная копия NASM такой же файл или проблема в переносимости между нашей платформой разработки и вашей. Мы можем принимать двоичные файлы в виде MIME-прикреплений, формате uuencode и даже BinHex. Так же мы можем предоставить вам ftp-сайт, на который вы сможете загрузить данный файл, но получить его по почте для нас было бы удобней.
- Любую другую информацию или файлы, которые могут быть полезны. Если, например, проблема в том, что NASM не может сгенерировать объектный файл, в то время, как TASM создает его без проблем, то пошлите нам *оба* объектных файла, чтобы мы могли увидеть, что делает TASM в отличие от NASM.