КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 07 Директивы NASM

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2022

2022.03.23

Оглавление

Директивы ассемблера

Директивы NASM бывают двух типов:

- директивы пользовательского уровня;
- примитивные директивы.

Обычно каждая директива имеет как пользовательскую, так и примитивную форму.

Пользовательская форма директив реализована как макрос, вызывающий примитивные формы.

В большинстве случаев рекомендуется использовать пользовательскую форму директив.

Примитивные директивы заключаются в квадратные скобки, для пользовательских директив этого не требуется.

```
 SECTION .data  -- пользовательская форма
[SECTION .data] – примитивная форма
```

Директивы можно разделить на универсальные и частные.

Универсальные директивы – общие для всех форматов объектного файла.

В дополнение к универсальным директивам каждый формат объектного файла может опционально предоставлять дополнительные директивы для управления конкретными функциями этого формата файла.

BITS — указание разрядности выполняемого кода

Директива **BITS** указывает, код какой разрядности должен генерировать NASM — для процессора, работающего в 16-битном, 32-битном или 64-битном режиме.

Соответствующим синтаксисом будет **BITS XX**, где **XX** — 16, 32 или 64.

В большинстве случаев не требуется использовать **BITS** явно. Объектные форматы **aout**, **coff**, **elf***, **win32** и **win64** заставляют NASM выбирать соответствующий режим по умолчанию.

Объектный формат **obj** позволяет 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 случай написания 32-битного плоского файла в формате **bin**, поскольку выходной формат **bin** по умолчанию предназначен для 16-битного режима и наиболее часто используется для написания DOS .COM программ, DOS .SYS драйверов устройств, а также загрузчиков.

BITS 16/32

Когда NASM находится в режиме **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**.

BITS 64

Когда NASM находится в режиме **BITS 64**, большинство инструкций работают так же, как и в режиме **BITS 32**. Однако добавляются 8 дополнительных регистров общего назначения и **SSE**, а 16-битная адресация больше не поддерживается.

Размер адреса по умолчанию — 64 бита.

32-битную адресацию можно выбрать с префиксом **0x67**.

Размер операнда по умолчанию по-прежнему составляет 32 бита, а префикс **0x66** выбирает 16-битный размер операнда.

Для выбора 64-битного размера операнда и для доступа к новым регистрам используется префикс **REX**. NASM при необходимости вставляет префиксы **REX** автоматически.

Когда используется префикс **REX**, процессор не может обращаться к регистрам **AH**, **BH**, **CH** или **DH** (старшие унаследованные 8-битные).

Взамен предоставлен доступ к младшим 8 битам регистров **SP**, **BP**, **SI** и **DI**, как **SPL**, **BPL**, **SIL** и **DIL**, соответственно, однако это справедливо только при использовании префикса **REX**.

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.

Следует обратить внимание, что шпация в директиве необходима — BITS32 не работает!

Для совместимости с другими ассемблерами вместо **BITS 16** и **BITS 32** могут использоваться директивы **USE16** и **USE32**.

DEFAULT — изменение настроек ассемблера по умолчанию

Директива **DEFAULT** изменяет настройки ассемблера по умолчанию.

Обычно NASM по умолчанию использует режим, в котором программист должен явно указать большинство функций напрямую. Однако это иногда достает, поскольку явная форма — практически единственная, которую желают использовать.

В настоящее время **DEFAULT**'ы можно установить для **REL & ABS** и **BND & NOBND**.

BND — это префикс (с кодом префикса **REPNE**), который используется при работе с инструкциями перехода в рамках режима Intel MPX (Memory Protection Extensions — расширения защиты памяти) — расширение набора инструкций для архитектуры x86/x86-64.

В результате расширение оказалось не способным предотвратить утечки данных через уязвимости Meltdown и Spectre.

Поддержка MPX удалена из всех современных компиляторов и ядер практически сразу, как только стало известно, что это УГ ни от чего не защищает, а наоборот, привносит проблемы, формируя ошибочные представления о защищенности.

`REL & ABS` — адресация относительно `RIP`

Устанавливает, как генерируются адреса для безрегистровых инструкций в 64-битном режиме — относительно RIP или нет, абсолютные.

По умолчанию они являются абсолютными, но их можно переопределить для конкретной инструкции спецификатором **`REL`**. Однако, если задано **`DEFAULT REL`**, по умолчанию используется **`REL`**, если адресация не переопределена с помощью спецификатора **`ABS`**.

Переопределение в случае, когда используется переопределение с сегментом **`FS`** или **`GS`**, не происходит.

Особая обработка переопределений **`FS`** и **`GS`** связана с тем, что эти регистры обычно используются в качестве указателей потоков (threads) или других специальных функций в 64-битном режиме, и создание адресов относительно **`RIP`**, было бы чрезвычайно запутанным.

`DEFAULT ABS` отключает **`DEFAULT REL`**.

BND & NOBND — BND префикс (для справки, что такой есть)

Если задано **DEFAULT BND**, все **инструкции**, следующие за этой директивой и которые могут употребляться с префиксом **bnd**, имеют префикс **bnd**.

Чтобы переопределить его, можно использовать префикс **NOBND**.

```
DEFAULT BND
     call foo             ; BND will be prefixed
     nobnd call foo       ; BND will NOT be prefixed
```

DEFAULT NOBND может отключить **DEFAULT BND**, и тогда префикс **BND** будет добавлен только в том случае, если он явно указан в коде.

DEFAULT BND, как ожидается, будет нормальной конфигурацией для написания кода с поддержкой **MPX**.

SECTION или SEGMENT — описание и изменение секций

Директива **SECTION** (**SEGMENT** — это абсолютно эквивалентный синоним) указывает, в какую секцию выходного файла будет ассемблироваться код.

В некоторых объектных форматах количество и имена секций фиксированы, в других пользователь может сделать их столько, сколько захочет. Поэтому если переключиться на секцию, которая в данный момент не существует, директива **SECTION** может вести себя по-разному — либо вызвать сообщение об ошибке, либо создать новую секцию.

Объектные форматы **Unix** и **bin** поддерживают стандартизованные имена секций:

.text — для кода;

.data — для данных;

.bss — для неинициализированных данных.

Формат **obj** наоборот, не признает эти имена секций специальными и более того, удаляет ведущую точку в имени любой секции.

Макрос __SECT__ (__?SECT?__)

Директива **SECTION** необычна тем, что ее пользовательская форма функционально отличается от примитивной.

Примитивная форма **[SECTION name]** просто переключает текущую секцию на указанную.

Пользовательская форма **SECTION name** 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.

Она может исключить необходимость использования инструкции **JMP** для «перескакивания» данных, располагающихся в секции и кода.

Например, макрос **writefile**, определенный ранее в «LK05.1 Поглощающие параметры макросов», можно с пользой переписать в следующей более сложной форме:

```
%macro  writefile 2+

        [section .data]

  %%str:        db      %2
  %%endstr:

        __?SECT?__                  ; Возвращаем ассемблирование в предыдущую секцию

        mov     dx, %%str
        mov     cx, %%endstr-%%str
        mov     bx, %1
        mov     ah, 0x40
        int     0x21

%endmacro
```

Эта форма макроса после передачи строки для вывода сначала временно переключается на секцию **.data**, используя примитивную форму директивы **SECTION**, чтобы не изменять значение макроса **__?SECT?__**. Затем макрос объявляет переданную строку в секции **.data**, а затем вызывает **__?SECT?__**, чтобы вернуться к той секции, в которой программист ранее работал. Таким образом, в данной версии макроса нет необходимости включать инструкцию **JMP**, чтобы перепрыгивать через данные, а также не создаст проблем, если создавая сложный модуль формата **OBJ** пользователь будет ассемблировать несколько отдельных частей кода.

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 можно 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), а также какое-то значение в сегменте.

Например, **TSR**[1] 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 использовано как хранилище данных для работающей TSR. Символ '**tsr_end**' может быть использован для расчета общего размера резидентной части TSR.

1 TSR -- Terminate and Stay Resident

EXTERN — импорт символов из других модулей

Директива **EXTERN** подобна 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 умеет.

Директива **EXTERN** принимает любое количество аргументов.

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 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.

Можно объявить одну и ту же переменную как **EXTERN** более одного раза — NASM спокойно проигнорирует второе и последующие переопределения.

REQUIRED — безусловный импорт символов из других модулей

Ключевое слово **REQUIRED** аналогично ключевому слову **EXTERN**.

Разница в том, что ключевое слово **EXTERN** версии 2.15 не генерирует неизвестные символы.

Это позволяет предотвратить использование общих файлов заголовков, что может приводить к тому, что компоновщик загрузит кучу ненужных модулей.

Если требуется старое поведение, следует использовать вместо **EXTERN** ключевое слово **REQUIRED**.

GLOBAL — экспорт символов в другие модули

Директива **GLOBAL** комплементарна директиве **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 и **EXTERN** позволяют вводить после двоеточия специфичный синтаксис конкретного объектного формата. К примеру объектный формат **elf** позволяет 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.

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 похоже на

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

Однако, есть отличие — если одна и та же common-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 intarray 100:4  ; работает в ELF: выравнивание по 4-байтной границе
```

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

`STATIC` — локальные символы в модулях

В отличие от **`EXTERN`** и **`GLOBAL`**, **`STATIC`** является локальным символом, но должен именоваться в соответствии с глобальными правилами декорирования имен[2] (названо по аналогии с ключевым словом C **`static`** применительно к функциям или глобальным переменным).

```
static foo
foo:
        ; codes
```

В отличие от **`GLOBAL`**, **`STATIC`** не позволяет вводить после двоеточия специфичный синтаксис конкретного объектного формата.

2 global mangling rules

Выходные форматы

В дополнение к описанным выше универсальным директивам каждый объектный формат может опционально предоставлять дополнительные директивы, служащие для управления особенностями этого формата.

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.

NASM выбирает имя по умолчанию для выходного файла на основе имени входного файла и указанного выходного формата. При этом расширение имени входного файла (**`.asm`**, **`.s`** или др.) удаляется и вместо него подставляется расширение соответствующего выходного формата.

`bin` — плоский бинарный формат

Формат **`bin`** не создает объектных файлов — в выходной файл не генерируется ничего, кроме написанного кода.

Такие «чисто бинарные» файлы используются в MS-DOS как исполняемые 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 явно не указана директива **`SECTION`**, написанный код будет направлен по умолчанию в секцию **`.text`**.

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

По умолчанию формат **`bin`** не создает суффикса для имени файла — он просто оставляет исходное имя файла, предварительно удалив его суффикс. Таким образом, NASM по умолчанию будет ассемблировать **`binprog.asm`** в бинарный файл **`binprog`**.

ORG — начало бинарного файла

Формат **bin** предусматривает дополнительную директиву **ORG**. Функция этой директивы — задать 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 делать перемещения 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.

bin-расширение директивы SECTION

Выходной формат **bin** расширяет директиву **SECTION** (или **SEGMENT**), позволяя 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 одновременно 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.

`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.

win32 — объектные файлы Win32 Microsoft

Выходной формат **win32** генерирует объектные файлы Microsoft Win32, передаваемые обычно компоновщикам Microsoft.

Надо отметить, что компиляторы Borland Win32 не используют этот формат, вместо него они используют **obj**.

win32 подразумевает по умолчанию расширение имени выходного объектного файла **.obj**.

Следует иметь ввиду, 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.

coff — общий формат объектных файлов

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

elf — объектные файлы ELF ОС Linux

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

elf-расширения директивы SECTION

Формат **elf** позволяет 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 задает параметр выравнивания секции.

Если не указано 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
```

Пр этом любые секции, не являющиеся **.text**, **.data** или **.bss** обрабатываются так же, как и секция **other**.

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 совместно используемой (shared) библиотеки. Для задания дополнительной информации NASM поддерживает некоторые расширения директивы **GLOBAL**.

Чтобы указать, чем является глобальная переменная — функцией или объектом данных, следует поставить после имени глобальной переменной двоеточие и написать **function** или **data** (**object** является синонимом **data**). Например, строка

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

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

После ввода спецификатора можно 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.

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.

aout — объектные файлы a.out Linux

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).

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.

rdf — перемещаемые динамические объектные файлы

Выходной формат rdf создает объектные файлы RDOFF.

RDOFF — это собственный 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.

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

dbg — формат для отладки

Выходной **dbg** формат в конфигурации NASM по умолчанию отсутствует.

Для включения этого формата в процессе сборки NASM следует 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, собирающимся 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.