КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 06 Препроцессор 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

2022.03.18

Оглавление

Препроцессор NASM (продолжение)....3
Условное ассемблирование....3
%ifdef — проверка присутствия однострочного макроса....4
%ifmacro — проверка существования многострочного макроса....5
%if — проверка произвольных числовых выражений....6
%ifidn и %ifidni — проверка на точную идентичность текста....7
%ifid, %ifnum, %ifstr — проверка типов символов....8
%ifempty — тест на пустое развертывание....11
%iftoken — тест на одиночный токен....12
%ifenv — тест на существование переменной среды....13
%error, %warning, %fatal — сообщения об ошибках, определенных пользователем....14
Подключение других файлов....16
%include — включение внешнего файла в исходный файл....16
Пример макроса stud_io.inc (из листинга удалено имеющее отношение к FreeBSD)....17
%pathsearch — поиск по пути включения....22
%depend — добавить зависимые файлы....23
Контекстный стек....24
%push и %pop — создание и удаление контекстов....25
%ifctx — тестирование контекстного стека....26
Контекстно-локальные метки....27
Контекстно-локальные однострочные макросы....28
%repl — переименование контекста....29
Пример использования контекстного стека — блочные IF....30
(@) Директивы препроцессора, относящиеся к стеку....33
%arg....34
%stacksize....35
%local....36
Стандартные макросы....38
__NASM_MAJOR__ и __NASM_MINOR__ — версия NASM....38
__FILE__ и __LINE__ — имя файла и номер строки....39
__BITS__ — текущий режим BITS....40
__OUTPUT_FORMAT__ — текущий выходной формат....41
__DEBUG_FORMAT__ — текущий отладочный формат (Current Debug Format)....42
Макросы времени и даты ассемблирования....43
__USE_package__ — тест включения пакета....45
__PASS__ — проход ассемблера....46
Структурные типы данных....47
STRUC и ENDSTRUC — объявление структурных типов данных....47
ISTRUC, AT и IEND — объявление экземпляров структур....50
Управление выравниванием....52
ALIGN и ALIGNB — выравнивание данных и кода....52
SECTALIGN — выравнивание секции....54
Стандартные пакеты макросов (макропакеты)....55
altreg — альтернативные имена регистров....55
smartalign — интеллектуальный макрос ALIGN....56
fp — макросы с плавающей точкой....58
ifunc — целочисленные функции....59
masm — совместимость с MASM....60

Препроцессор NASM (продолжение)

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, можно использовать более одного оператора **%elif**.

Поддерживаются проверки:

%ifdef, **%ifndef** — определен однострочный макрос или нет;

%imacro, **%ifnmacro** — определен многострочный макрос или нет;

%if — произвольных числовых выражений;

%ifidn — идентичность текста;

%ifid, **%ifnum**, **%ifstr** — тип параметра (идентификатор/адрес, число, строка);

`%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 можно использовать следующий код:

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

Для определения **`DEBUG`** можно использовать ключ **`-dDEBUG`** командной строки для создания версии программы, выдающей отладочные сообщения, а без ключа — создавать окончательный релиз.

Для осуществления обратной проверки (отсутствия определения макроса) можно использовать вместо **`%ifdef`** оператор **`%ifndef`**.

Также можно тестировать наличие определения макроса в блоках **`%elif`** при помощи операторов **`%elifdef`** и **`%elifndef`**.

`%ifmacro` — проверка существования многострочного макроса

Директива **`%ifmacro`** действует так же, как директива **`%ifdef`**, за исключением того, что проверяет наличие многострочного макроса.

Например, при работе с большим проектом может не оказаться возможности управлять макросами в библиотеке. В этом случае можно создать макрос с одним именем, если он еще не существует, и другим именем, если макрос с таким именем существует.

`%ifmacro` считается истинным, если определение макроса с заданным именем и количеством аргументов вызовет конфликт определений. Например:

```
%ifmacro MyMacro 1-3
     %error "MyMacro 1-3" causes a conflict with an existing macro.
%else
     %macro MyMacro 1-3
             ; insert code to define the macro
     %endmacro
%endif
```

Это создаст макрос «**`MyMacro 1-3`**», если еще не существует макроса, который мог бы конфликтовать с ним, и выдаст предупреждение, если возникнет конфликт определений.

Можно проверить отсутствие макроса, используя **`%ifnmacro`** вместо **`%ifmacro`**.

Дополнительные тесты могут быть выполнены в блоках **`%elif`** с помощью **`%elifmacro`** и **`%elifnmacro`**.

`%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`** (Фибоначчи) выше.

Выражения, указываемые для `%if`, а также его эквивалента `%elif`, являются критическими.

`%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 **`<>`**.

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).

`%ifidn` и `%ifidni` — проверка на точную 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 else-эквивалент **`%elifidn`** и обратные формы **`%ifnidn`** и **`%elifnidn`**. Соответственно, **`%ifidni`** имеет эквивалент **`%elifidni`** и обратные формы **`%ifnidni`** и **`%elifnidni`**.

`%ifid`, `%ifnum`, `%ifstr` — проверка типов символов

Иногда 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 разделе «LK05 Поглощающие параметры макросов» макрос **writefile** может быть расширен:

```
%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 вызовами:

```
          writefile [file], strptr, strlen
          writefile [file], "Привет!", 13, 10
```

В первом случае **strptr** используется в качестве адреса уже объявленной строки, а **strlen** — как длина этой строки.

Во втором случае макросу передается строка, которую макрос объявляет и получает ее адрес и длину самостоятельно.

Следует обратить 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**.

`%ifempty` — тест на пустое развертывание

Конструкция условного ассемблирования **`%ifempty`** ассемблирует соответствующий код тогда и только тогда, когда после развертывания параметры вообще не содержат никаких токенов, за исключением пробелов.

Также, как обычно, предусмотрены варианты **`%elifempty`**, **`%ifnempty`** и **`%elifnempty`**.

`%iftoken` — тест на одиночный токен

Некоторые макросы захотят делать разные вещи в зависимости от того, передается ли ему одна лексема (с помощью **`%+`**), или последовательность из нескольких лексем.

Конструкция **`%iftoken`** ассемблирует последующий код тогда и только тогда, когда развернутые параметры состоят ровно из одной лексемы, возможно, окруженной пробелами.

Например,

```
%iftoken 1
```

будет ассемблировать последующий код, а

```
%iftoken -1
```

нет, поскольку **`-1`** содержит две лексемы — унарный оператор «минус» и число 1.

Также, как и для других директив, работают варианты **`%eliftoken`**, **`%ifntoken`** и **`%elifntoken`**.

`%ifenv` — тест на существование переменной среды

Конструкция условного ассемблирования **`%ifenv`** ассемблирует соответствующий код тогда и только тогда, когда существует переменная среды **`%!variable`**.

Также, как обычно, предусмотрены варианты **`%elifenv`**, **`%ifnenv`** и **`%elifnenv`**.

Как и для переменной **`%!`**, аргумент должен быть записан в виде строки, если он содержит символы, которые недопустимы в идентификаторе.

`%error`, `%warning`, `%fatal` — сообщения об ошибках, определенных пользователем

Директива препроцессора **`%error`** заставляет NASM сообщать об ошибках, случающихся на стадии ассемблирования. Например, при ассемблировании исходного текста можно проверить, определен ли нужный макрос:

```
%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.

Аналогично **`%warning`** выдает предупреждение, но позволяет продолжить ассемблирование:

```
%ifdef F1
    ; do some setup
%elifdef F2
    ; do some different setup
%else
    %warning "Не определены ни F1, ни F2, поэтому выбираем F1."
    %define F1
%endif
```

%error и **%warning** выдаются только на финальном проходе ассемблирования. Это позволяет их использовать вместе с проверками условного ассемблирования, результаты которых зависят от значений символов.

%fatal немедленно прекращает ассемблирование независимо от прохода. Это полезно, когда нет смысла продолжать сборку дальше, тем более, если продолжение, скорее всего, вызовет массу сбивающих с толку сообщений об ошибках.

Заключать строку сообщения после **%error**, **%warning** или **%fatal** в кавычки необязательно. В этом случае в строке раскрываются однострочные макросы, которые можно использовать для отображения дополнительной информации для пользователя.

Например:

```
%if foo > 64
    %assign foo_over foo-64
    %error foo is foo_over bytes too large
%endif
```

Подключение других файлов

%include — включение внешнего файла в исходный файл

Препроцессор NASM'а, используя очень похожий на C синтаксис, позволяет подключать к файлу с исходным кодом 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), а также 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; MACROS_MAC
```

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

Пример макроса `stud_io.inc` (из листинга удалено имеющее отношение к FreeBSD)

```
;; File stud_io.inc for both Linux and FreeBSD.
;; Copyright (c) Andrey Vikt. Stolyarov, 2009, 2015
;; I, the author, hereby grant everyone the right to use this
;; file for any purpose, in any manner, in it's original or
;; modified form, provided that any modified versions are
;; clearly marked as such.

; generic 3-param syscall
%macro _syscall_3 4
   push edx
   push ecx
   push ebx ; it is senseless to save eax as it holds the return

   push %1
   push %2
   push %3
   push %4
   pop edx  ; <- P4
   pop ecx  ; <- P3
   pop ebx  ; <- P2
   pop eax  ; <- P1
   int 0x80

   pop ebx
   pop ecx
   pop edx
%endmacro
```

```
; syscall_exit is the only syscall we use that has 1 parameter
%macro _syscall_exit 1
   mov ebx, %1  ; exit code
   mov eax, 1   ; 1 = sys_exit
   int 0x80
%endmacro

%elifdef STUD_IO_FREEBSD
...

%endif

;; system dependent part ends here
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; %1: descriptor   %2: buffer addr   %3: buffer length
; output: eax: прочитано байт
%macro _syscall_read 3
   _syscall_3 3, %1, %2, %3
%endmacro

; %1: descriptor   %2: buffer addr   %3: buffer length
; output: eax: записано байт
%macro _syscall_write 3
   _syscall_3 4, %1, %2, %3
%endmacro
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro   PRINT 1
   pusha
   pushf
   jmp %%astr
%%str db %1, 0
%%strln  equ$-%%str
%%astr:  _syscall_write 1, %%str, %%strln
   popf
   popa
%endmacro

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro   PUTCHAR 1
   pusha
   pushf
%ifstr %1
   moval, %1
%elifnum %1
   moval, %1
%elifidni %1,al
   nop
%elifidni %1,ah
   moval, ah
%elifidni %1,bl
   moval, bl
%elifidni %1,bh
   moval, bh
%elifidni %1,cl
   moval, cl
```

```
%elifidni %1,ch
   mov al, ch
%elifidni %1,dl
   mov al, dl
%elifidni %1,dh
   mov al, dh
%else
   mov al, %1  ; let's hope it is a memory location such as [var]
%endif
   sub    esp, 2  ; reserve memory for buffer
   mov    edi, esp
   mov    [edi], al
   _syscall_write 1, edi, 1
   add    esp, 2
   popf
   popa
%endmacro

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro   GETCHAR 0
   pushf
   push   edi
   sub esp, 2
   mov edi, esp
   _syscall_read 0, edi, 1
   cmp eax, 1
   jne %%eof_reached
   xor eax,eax
   mov al, [edi]
   jmp %%gcquit
```

```
%%eof_reached:
   xor eax, eax
   not eax        ; eax := -1
%%gcquit:
   add esp, 2
   pop edi
   popf
%endmacro

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro FINISH 0-1 0
   _syscall_exit %1
%endmacro
```

`%pathsearch` — поиск по пути включения

Директива **`%pathsearch`** принимает однострочное имя макроса и имя файла. Если указанный файл существует, директива объявляет или переопределяет указанный однострочный макрос как версию имени файла с полным маршрутом. Если файл не существует, он передается без изменений. Например,

```
%pathsearch foo_bin "foo.bin"
```

с **`-Ibins/`** в пути включения может в результате определить макрос **`foo_bin`** как строку «**`bins/foo.bin`**».

%depend — добавить зависимые файлы

Директива **%depends** принимает имя файла и добавляет его в список файлов, которые будут генерироваться как генерация зависимостей, когда используются опции командной строки **-M** и их «родственники». Она не производит никакого вывода.

Обычно используется вместе с **%pathsearch**. Например, упрощенная версия стандартной макро-обертки директивы **INCBIN** выглядит следующим образом:

```
%imacro incbin 1-2+ 0
%pathsearch dep %1
%depend dep
        incbin dep,%2
%endmacro
```

Код сначала разворачивает расположение файла для макроса **dep**, затем добавляет его в списки зависимостей и, наконец, выдает директиву **INCBIN** на уровне ассемблера.

Контекстный стек

Наличие меток, которые являются локальными в определении макроса, иногда недостаточно мощно — иногда необходимо иметь возможность совместно использовать одни и те же метки в разных макровызовах.

Примером может быть цикл **`REPEAT ... UNTIL`**, в котором раскрытие макроса **`REPEAT`** должно иметь возможность ссылаться на метку, определенную макросом **`UNTIL`**. При этом может понадобиться возможность вкладывать эти циклы.

NASM обеспечивает такой уровень мощности с помощью контекстного стека.

Препроцессор поддерживает стек контекстов, каждый из которых характеризуется именем. Можно добавить новый контекст в стек с помощью директивы **`%push`** и удалить один с помощью **`%pop`**. Также можно определить метки, которые являются локальными для определенного контекста в стеке.

`%push` и `%pop` — создание и удаление контекстов

Директива **`%push`** используется для создания нового контекста и размещения его на вершине стека контекстов.

Директива **`%push`** принимает необязательный аргумент — имя контекста. Например:

```
%push     foobar
```

помещает в стек новый контекст, называемый **`foobar`**.

Может быть несколько контекстов в стеке с одним и тем же именем, но их все еще можно различить.

Если имя не указано, контекст не имеет имени (обычно это используется, когда и **`%push`**, и **`%pop`** находятся внутри одного определения макроса.)

Директива **`%pop`**, принимая один необязательный аргумент, удаляет верхний контекст из стека контекстов и уничтожает его вместе со всеми связанными с ним метками. Если аргумент указан, он должен соответствовать имени текущего контекста, в противном случае будет выдана ошибка.

`%ifctx` — тестирование контекстного стека

Директива условного ассемблирования **`%ifctx`** вызовет ассемблирование последующего кода тогда и только тогда, когда верхний контекст в стеке контекстов препроцессора имеет то же имя, что и один из аргументов. Как и **`%ifdef`**, также поддерживаются обратные формы и **`%elif`** — **`%ifnctx`**, **`%elifctx`** и **`%elifnctx`**.

Контекстно-локальные метки

Так же, как использование **%%foo** определяет метку, которая является локальной для конкретного вызова макроса, в котором она используется, использование **%$foo** определяет метку, которая является локальной для контекста на вершине контекстного стека. Таким образом, упомянутый выше пример **REPEAT** и **UNTIL** может быть реализован с помощью следующего кода:

```
%macro repeat 0
  %push   repeat
  %$begin:          ; локальная для контекста метка
%endmacro

%macro until 1
         j%-1 %$begin ; условный переход на контекстно-локальную метку (инверсный CC)
  %pop
%endmacro
```

и вызывается, например, так:

```
        mov       edi, string
        repeat
          scasb               ; вычитает байт [edi] из AL, устанавливает флаги и edi++
          add     edi, 3      ; -> на следующий четвертый байт
        until     e
```

который будет сканировать каждый четвертый байт строки в поисках содержимого, которое в **AL**. Если необходимо определить или получить доступ к меткам, локальным для контекста, расположенного ниже верхнего в стеке, можно использовать **%$$foo** или **%$$$foo** для контекста, который ниже данного и так далее.

Контекстно-локальные однострочные макросы

NASM также позволяет с помощью такого же префикса **%$** определять однострочные макросы, которые являются локальными для определенного контекста:

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

определит однострочный макрос **%$localmac** как локальный для контекста на вершине стека.

При этом, после последующей выдачи **%push** к нему все еще можно будет получить доступ по имени **%$$localmac**.

`%repl` — переименование контекста

Если необходимо изменить имя контекста на вершине стека (например, чтобы он по-другому реагировал на **`%ifctx`**), можно выполнить **`%pop`** с последующим **`%push`**, но это будет иметь побочный эффект в виде уничтожения всех локальных для контекста меток и макросов, связанных с контекстом, который только что был извлечен.

NASM предоставляет директиву **`%repl`**, которая заменяет контекст другим именем, не затрагивая связанные макросы и метки, чтобы была возможность заменить разрушающий код

```
%pop
%push    newname
```

на неразрушающую версию

```
%repl newname
```

Пример использования контекстного стека — блочные IF

В этом примере для реализации оператора блочного IF в виде набора макросов используются почти все функции контекстного стека, включая конструкцию условного ассемблирования **%ifctx** (проверка контекста на вершине стека).

```
%macro if 1
    %push if            ; создаем новый контекст
    j%-1  %$ifnot       ; инверсный Jcc на контекстно-локальную метку %$ifnot
%endmacro

%macro else 0
  %ifctx if             ; если в контексте "if"
        %repl   else    ; переименовываем контекст на "else"
        jmp     %$ifend ; безусловный переход на контекстно-локальную метку %$ifend
        %$ifnot:
  %else                 ; если не в контексте "if" (нарушен баланс "скобок")
        %error  "expected `if' before `else'"
  %endif
%endmacro

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

Этот код более надежен, чем макросы **REPEAT** и **UNTIL**, приведенные в разделе «Контекстно-локальные метки», поскольку он использует условное ассемблирование для проверки того, что макросы выдаются в правильном порядке (например, не вызывается **endif** перед **if**), и выдает **%error**, если это не так.

Также макрос **endif** имеет дело с двумя разными случаями — он идет либо после **if**, либо после **else**. Это достигается за счет использования условного ассемблирования для выполнения разных действий в зависимости от того, является ли контекст на вершине стека **if** или **else**.

Макрос **else** должен сохранять контекст в стеке, чтобы ссылка на макрос **%$if**, на который не ссылается макрос **if**, был таким же, как тот, который определен макросом **endif**, но должен изменить имя контекста, чтобы **endif** знал, что произошло «вмешательство» **else**. Это делается с помощью **%repl**. Пример использования этих макросов может выглядеть так:

```
        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 блочного **IF** вполне успешно справляются с вложением посредством добавления другого контекста, описывающего внутреннее **if**, поверх того, которое описывает внешнее **if**. Таким образом **else** и **endif** всегда относятся к последнему несогласованному **if** или **else**.

(@) Директивы препроцессора, относящиеся к стеку

Следующие директивы препроцессора позволяют использовать метки для ссылки на локальные переменные, размещенные в стеке.

```
%arg
%stacksize
%local
```

Будут рассматриваться в разделе о передаче параметров в процедуры ЯВУ

%arg

Директива **%arg** используется для упрощения обработки параметров, передаваемых через стек. Передача параметров на основе стека используется многими языками высокого уровня, включая C, C++ и Pascal.

Хотя в NASM есть макросы, которые пытаются дублировать эту функцию, их синтаксис не особенно удобен в использовании и несовместим с TASM.

Пример, показывающий использование директивы **%arg** без каких-либо внешних макросов:

```
some_function:

  %push      mycontext          ; сохранение текущего контекста
  %stacksize large              ; говорит NASM использ. для доступа к параметрам bp
  %arg       i:word, j_ptr:word ; аргументы вызова

      mov     ax,[i]            ; mov ax,[bp+4]
      mov     bx,[j_ptr]        ; mov bx,[bp+6]
      add     ax,[bx]
      ret

  %pop                        ; восстановление оригинального контекста
```

%stacksize

Директива **%stacksize** используется вместе с директивами **%arg** и **%local**.

Она сообщает NASM размер по умолчанию, который будет использоваться для последующих директив **%arg** и **%local**. Директива **%stacksize** принимает один обязательный аргумент из набора: **flat**, **flat64**, **large** или **small**.

```
%stacksize flat
```

NASM будет использовать адресацию параметров в стеке относительно **ebp** и предполагается, что для доступа к метке используется ближняя форма вызова (то есть в стеке находится **eip**).

```
%stacksize flat64
```

NASM будет использовать адресацию параметров в стеке относительно **rbp** и предполагается, что для доступа к параметру используется ближняя форма вызова (т.е. в стеке находится **rip**).

```
%stacksize large
```

Для адресации параметров, размещенных в стеке, используется **bp** и предполагается, что для доступа к параметрам используется дальняя форма вызова (т.е. что в стеке находятся **ip** и **cs**).

```
%stacksize small
```

Эта форма также использует регистр **bp** для адресации параметров в стеке, но отличается от **large**, поскольку также предполагает, что в стек помещается старое значение **bp** (т.е. ожидается инструкция **ENTER**). Другими словами, ожидается, что регистры **bp**, **ip** и **cs** находятся на вершине стека, под любым локальным пространством, которое могло быть выделено с помощью **ENTER**.

Эта форма, вероятно, наиболее полезна при использовании в сочетании с директивой **%local**.

%local

Директива **%local** используется для упрощения использования локальных временных переменных, выделенных в кадре стека.

Примером таких переменных являются автоматические локальные переменные в C.

Директива **%local** наиболее полезна при использовании с директивой **%stacksize**, а также совместима с директивой **%arg**. Она позволяет упростить ссылку на переменные в стеке, которые обычно выделяются с помощью инструкции **ENTER**. Пример ее использования следующий:

```
silly_swap:

    %push mycontext             ; сохраняем текущий контекст
    %stacksize small            ; говорим NASM, чтобы тот использовал bp для доступа
    %assign %$localsize 0       ; размер резервируемой памяти в стеке (внутренняя)
    %local old_ax:word, old_dx:word

        enter   %$localsize, 0  ; enter/leave – поддержка процедурных ЯВУ (32 бита)
        mov     [old_ax],ax     ; swap ax & bx
        mov     [old_dx],dx     ; and swap dx & cx
        mov     ax,bx
        mov     dx,cx
        mov     bx,[old_ax]
        mov     cx,[old_dx]
        leave                   ; restore old bp
        ret                     ;

    %pop                        ; restore original context
```

Директива **%local** внутренне использует переменную **%$localsize**. Поэтому, прежде чем можно будет использовать директиву %local, переменная должна быть в текущем контексте определена. Несоблюдение этого правила приведет для каждой переменной, объявленной **%local**, к выдаче синтаксической ошибки.

Также ее можно использовать при построении инструкции ENTER с выделением памяти требуемого размера, как показано в примере.

Стандартные макросы

NASM вводит набор стандартных макросов, которые на момент начала обработки любого исходного файла будут уже определены.

Если необходимо, чтобы программа ассемблировалась без предопределенных макросов, для очистки препроцессорного пространства имен следует использовать директиву

```
%clear
```

__NASM_MAJOR__ и __NASM_MINOR__ — версия NASM

Однострочные макросы **__NASM_MAJOR__** и **__NASM_MINOR__** разворачиваются, соответственно, в старшую и младшую части номера версии NASM.

Например, в NASM 2.13 **__NASM_MAJOR__** будет определен как **2**, а **__NASM_MINOR__** — как **13**.

Числа, полученные в результате развертывания, можно использовать в директивах условного ассемблирования, чтобы использовать (или, наоборот, не использовать) возможности NASM более высоких версий или устаревшие.

__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.

Вызов **__LINE__** внутри макроопределения (неважно, одно- или многострочного) будет возвращать номер строки макровызова, а не строки макроопределения.

Эти макросы могут быть использованы, например, для передачи макросу отладочной информации. Так, например, для определения в какой части кода наступает крах, пишется подпрограмма **stillhere**, которой передается в **EAX** номер строки, а на выходе получается что-то вроде «**строка 155: я еще жива**». Затем пишется макрос

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

и «вставляется в код вызовами **notdeadyet** до тех пор, пока не будет найдена точка краха.

__BITS__ — текущий режим BITS

Стандартный макрос **__BITS__** обновляется всякий раз, как с помощью директив **BITS XX** или **[BITS XX]** устанавливается соответствующий режим **BITS**. **XX** — это число, означающее режим, которое может принимать значения 16, 32, 64.

__BITS__ получает соответствующее числовое значение и делает его глобально доступным. Данный макрос полезен при написании макросов, зависящих от режима выполнения.

__OUTPUT_FORMAT__ — текущий выходной формат

Стандартный макрос **__OUTPUT_FORMAT__** содержит имя текущего выходного формата, как оно было указано в опции **-f** либо установлено по умолчанию.

```
%ifidn __OUTPUT_FORMAT__, win32
    %define NEWLINE 0Dh, 0Ah                ; <CR><LF>
%elifidn __OUTPUT_FORMAT__, elf32
    %define NEWLINE 0Ah                     ; <LF>
%endif
```

Так же полезен макрос для определения syscal

```
%ifidn __OUTPUT_FORMAT__, elf32
    %define syscall int 80h             ; 0F05
%elifidn __OUTPUT_FORMAT__, elf64
    %define syscall syscall             ; CD80
%else
    %error Выходной формат __OUTPUT_FORMAT__ не поддерживается
%endif
```

`__DEBUG_FORMAT__` — текущий отладочный формат (Current Debug Format)

Если была разрешена генерация отладочной информации стандартный макрос **`__DEBUG_FORMAT__`** будет содержать имя текущего отладочного формата, как оно было указано в опциях **`-F`** или **`-g`**, либо установлено по умолчанию.

Если генерация отладочной информации не была разрешена, или спефикатор отладочного формата **`null`**, значение **`__DEBUG_FORMAT__`** не определено.

Макросы времени и даты ассемблирования

NASM предоставляет различные макросы, которые содержат метку времени сеанса сборки.

Макросы **__DATE__ и __TIME__** дают дату и время сборки в виде строк в формате ISO 8601 «ГГГГ-ММ-ДД» и «ЧЧ: ММ: СС», соответственно.

Макросы **__DATE_NUM__** и **__TIME_NUM__** дают дату и время сборки в числовом виде — в формате ГГГГММДД и ЧЧММСС соответственно.

Макросы **__UTC_DATE__** и **__UTC_TIME__** дают дату и время сборки в универсальном времени (UTC) в виде строк в формате ISO 8601 «YYYY-MM-DD» и «HH: MM: SS» соответственно.

Если платформа хоста не предоставляет время UTC, эти макросы не определены.

Макросы **__UTC_DATE_NUM__** и **__UTC_TIME_NUM__** дают универсальную дату и время сборки (UTC) в числовой форме; в формате ГГГГММДД и ЧЧММСС соответственно. Если время UTC недоступно на платформе хоста, эти макросы не определены.

Макрос **__POSIX_TIME__** определяется как число, содержащее количество секунд, прошедших с POSIX-эпохи (1 января 1970 года 00:00:00 UTC), исключая любые високосные секунды. Данная величина вычисляется с использованием времени UTC, если оно доступно на платформе хоста, в противном случае оно вычисляется с использованием местного времени, как если бы оно было UTC.

Все экземпляры макросов времени и даты в одном сеансе сборки дают согласованный выход. Например, в сеансе сборки, начатом через 42 секунды после полуночи 1 января 2010 года в Минске (часовой пояс UTC + 3), эти макросы будут иметь следующие значения, если, конечно, будет правильно настроенная среда с правильными часами:

```
__DATE__              "2010-01-01"
__TIME__              "00:00:42"
__DATE_NUM__          20100101
__TIME_NUM__          000042
__UTC_DATE__          "2009-12-31"
__UTC_TIME__          "21:00:42"
__UTC_DATE_NUM__      20091231
__UTC_TIME_NUM__      210042
__POSIX_TIME__        1262293242
```

__USE_package__ — тест включения пакета

Директива %use — подключение стандартного макропакета (Standard Macro Package)

Директива **%use** похожа на **%include**, но в отличие от нее вместо включения всего содержимого файла она включает только поименованные стандартные макропакеты.

Стандартные макропакеты являются компонентами NASM и описаны в главе 6 руководства «Стандартные макропакеты».

В отличие от директивы **%include** пакеты, поименованные в директиве %use не требуют заключения в кавычки, хотя это и разрешается. Такаим образом следующие строки кода эквивалентны:

```
%use altreg
%use 'altreg'
```

Стандартные макропакеты защищены от повторного включения. Если с помощью директивы **%use** включен какой либо стандартный пакет макросов, автоматически определяется однострочный макрос вида **__USE_package__**. Это позволяет проверить, вызывается ли конкретный пакет или нет.

Например, если был включен пакет **altreg**, то будет определен макрос **__USE_ALTREG__**.

__PASS__ — проход ассемблера

Макрос **__PASS__** определен как 1 на подготовительных этапах и 2 на последнем проходе.

В режиме только препроцессора он равен 3, а при запуске только для генерации зависимостей (при указании опций **-M** или **-MG**) он равен 0.

Следует избегать использования этого макроса, если это возможно — при неправильном его использовании достаточно легко сгенерировать очень странные ошибки, а также может изменить-ся семантика этого макроса в будущих версиях NASM.

Структурные типы данных

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 или два параметра (аналог тега структуры).

Первый параметр — это ***имя типа данных***.

Второй необязательный параметр — это базовое смещение структуры.

Имя типа данных определяется как символ со значением базового смещения, а имя типа данных с добавленным к нему суффиксом **_size** задает размер структуры (определяется с помощью **EQU**).

После того, как макрос **STRUC** завершен, идет описание структуры данных путем определения полей при помощи семейства псевдоинструкций **RESB**.

В конце описания вызывается **ENDSTRUC**.

Например, определение структуры **mytype** может иметь вид:

```
            struc mytype  ; 0
mt_long:    resd 1        ; 0
mt_word:    resw 1        ; 4
mt_byte:    resb 1        ; 6
mt_str:     resb 32       ; 7
            endstruc
```

```
            struc mytype  ; 0
mt_long:    resd 1        ; 0
mt_word:    resw 1        ; 4
mt_byte:    resb 1        ; 6
mt_str:     resb 32       ; 7
            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**;

mytype как **0**.

Если члены структуры имеют метки с именами, совпадающими с именами меток других структур, можно переписать приведенный выше код следующим образом:

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

В этом примере описываются смещения к полям структуры в виде **mytype.long**, **mytype.word**, **mytype.byte** и **mytype.str**.

NASM не имеет встроенной поддержки структур, поэтому 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] ;  mystruc – имя экземпляра структуры
```

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]
```

ISTRUC, AT и IEND — объявление экземпляров структур

Для объявления экземпляра структуры в сегменте данных (иначе зачем она вообще нужна) NASM предусматривает способ с использованием механизма **ISTRUC**.

Для объявления (определения) в программе структуры типа **mytype**

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

необходимо написать:

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

Функцией макроса **AT** является продвижение позиции ассемблирования в корректную точку заданного поля структуры и последующего объявления указанных данных. В связи с этим 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
```

Также можно полностью пропустить код на строке **AT** и начать поле структуры со следующей строки:

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

Управление выравниванием

ALIGN и ALIGNB — выравнивание данных и кода

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 в секциях кода и данных должно использоваться **ALIGN**, а **ALIGNB** — в секции BSS. Тогда никакого второго аргумента не понадобится (кроме, конечно, специальных случаев).

В прежних версиях **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». В 2.15 такая проверка уже есть.

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.

ALIGN и **ALIGNB** работают относительно начала секции, а не начала адресного пространства в конечном исполняемом 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**.

SECTALIGN — выравнивание секции

Макрос **SECTALIGN** позволяет изменять атрибут выравнивания секции в выходном файле.

В отличие от атрибута **align =** (который разрешен только при определении секции) макрос SECTALIGN можно использовать в любое время. Например, директива

```
SECTALIGN 16
```

устанавливает требования к выравниванию секции, равному 16 байтам. После увеличения выравнивание нельзя уменьшить — величина выравнивания может только увеличиваться.

Следует обратить внимание, что **ALIGN** неявно вызывает макрос **SECTALIGN**, поэтому требования к выравниванию активного раздела могут быть обновлены. Это поведение по умолчанию, и если по какой-то причине необходимо, чтобы **ALIGN** вообще не вызывала **SECTALIGN**, следует использовать директиву

```
SECTALIGN OFF
```

при этом ее можно включить снова

```
SECTALIGN ON
```

Следует обратить внимание, что **SECTALIGN <ON|OFF>** влияет только на директивы **ALIGN/ALIGNB**, но не на явную директиву **SECTALIGN**.

Стандартные пакеты макросов (макропакеты)

Директива **%use** подключает один из стандартных пакетов макросов, входящих в состав дистрибутива NASM и вкомпилированных в двоичный файл NASM.

Директива **%use** работает как директива **%include**, но включаемое содержимое предоставляется самим NASM.

Имена стандартных пакетов макросов нечувствительны к регистру и могут быть заключены в кавычки.

Начиная с версии 2.15 в NASM есть директивы **%ifusable** и **%ifusing**, которые помогают пользователю понять, доступен ли отдельный пакет в этой версии NASM (**%ifusable**) или конкретный пакет уже загружен (**%ifusing**).

altreg — альтернативные имена регистров

Стандартный пакет макросов **altreg** предоставляет альтернативные имена регистров.

Он предоставляет числовые имена регистров для всех регистров (не только **R8** – **R15**), определенные Intel-псевдонимы **R8L** – **R15L** для младших байтов регистра (в отличие от стандартных имен NASM/AMD **R8B** – **R15B**) и имена **R0H**– **R3H** (по аналогии с **R0L** – **R3L**) для **AH**, **CH**, **DH** и **BH**.

Пример использования:

```
%use altreg

proc:
      mov r0l,r3h                    ; mov al,bh
      ret
```

smartalign — интеллектуальный макрос ALIGN

Стандартный пакет макросов **smartalign** предоставляет более мощный макрос **ALIGN**, чем стандартный (и обратно совместимый).

Когда пакет **smartalign** подключен, то если **ALIGN** используется без второго аргумента, NASM сгенерирует последовательность инструкций, более эффективную, чем последовательность инструкций **NOP**.

Кроме того, если заполнение превышает определенный порог, то NASM сгенерирует **jmp** через все заполнение.

Какие инструкции будут конкретно генерироваться, можно управлять с помощью макроса **ALIGNMODE**. Этот макрос принимает два параметра — один режим и необязательное переопределение порогового значения **jmp**.

Если (по какой-либо причине) необходимо полностью отключить **jmp**, следует установить значение порога **jmp** на **–1** (или установите его на **nojmp**). Возможны следующие режимы:

generic — работает на всех процессорах x86 и должен иметь приемлемую производительность. Порог перехода по умолчанию равен 8. Этот режим является режимом по умолчанию.

nop — заполнение инструкциями **NOP**. Единственное отличие от стандартного макроса **ALIGN** состоит в том, что NASM все еще может перепрыгивать через большую область заполнения. Порог перепрыгивания по умолчанию равен 16.

k7 — оптимизация для AMD K7 (Athlon/Althon XP). Эти инструкции должны работать на всех процессорах x86. Порог перепрыгивания по умолчанию равен 16.

k8 — оптимизация для AMD K8 (Opteron/Althon 64). Эти инструкции должны работать на всех процессорах x86. Порог перепрыгивания по умолчанию равен 16.

p6 — оптимизация для процессоров Intel. При этом используются длинные инструкции **NOP**, впервые представленные в Pentium Pro. Эта оптимизация несовместима со всеми ЦП семейства 5 или ниже, а также с некоторыми ЦП VIA и некоторыми решениями для виртуализации. Порог перепрыгивания по умолчанию равен 16.

Макрос, содержащий текущий режим выравнивания, определен, как **__?ALIGNMODE?__**.

Внутри этого пакета макросов используется несколько других макросов, которые начинаются с **__?ALIGN_**,

`fp` — макросы с плавающей точкой

Этот пакет содержит следующие удобные макросы с плавающей точкой:

```
%define Inf             __?Infinity?__
%define NaN             __?QNaN?__
%define QNaN            __?QNaN?__
%define SNaN            __?SNaN?__

%define float8(x)       __?float8?__(x)
%define float16(x)      __?float16?__(x)
%define bfloat16(x)     __?bfloat16?__(x)
%define float32(x)      __?float32?__(x)
%define float64(x)      __?float64?__(x)
%define float80m(x)     __?float80m?__(x)
%define float80e(x)     __?float80e?__(x)
%define float128l(x)    __?float128l?__(x)
%define float128h(x)    __?float128h?__(x)
```

Он также определяет многострочный макрос **`bf16`**, который можно использовать аналогично директивам **`Dx`** для других чисел с плавающей запятой:

```
bf16 -3.1415, NaN, 2000.0, +Inf
```

`ifunc` — целочисленные функции

Этот пакет содержит набор макросов, реализующих целочисленные функции. На самом деле они реализованы как специальные операторы, но наиболее удобный доступ к ним осуществляется через этот пакет макросов. Предоставляются следующие макросы:

Целочисленные логарифмы

Эти функции вычисляют целочисленный логарифм по основанию 2 своего аргумента, рассматриваемого как целое число без знака. Единственное различие между функциями — это их соответствующее поведение, если предоставленный аргумент не является степенью двойки.

Функция **`ilog2e()`** (псевдоним **`ilog2()`**) генерирует ошибку, если аргумент не является степенью двойки.

Функция **`ilog2f()`** округляет аргумент до ближайшей степени двойки. Если ее аргумент равен нулю, функция возвращает ноль.

Функция **`ilog2c()`** округляет аргумент до ближайшей степени двойки.

Функции **`ilog2fw()`** (псевдоним **`ilog2w()`**) и **`ilog2cw()`** генерируют предупреждение, если аргумент не является степенью двойки, но в остальном ведут себя как **`ilog2f()`** и **`ilog2c()`** соответственно.

`masm` — совместимость с MASM

Начиная с версии 2.15, NASM имеет пакет совместимости с MASM с минимальной функциональностью, предназначенный для использования в основном с машинно-сгенерированным кодом. Он не включает никаких «удобных для программистов» сокращений и никоим образом не поддерживает **ASSUME**, набор символов или структуры в стиле MASM.

Чтобы включить пакет, используйте директиву:

```
%use masm
```

В настоящее время пакет совместимости MASM обеспечивает следующее поведение:

- ключевые слова **FLAT** и **OFFSET** распознаются и игнорируются;

- ключевое слово **PTR** обозначает ссылку на память, как если бы аргумент был заключен в квадратные скобки:

```
        mov eax,[foo]                ; ссылка на память
        mov eax,dword ptr foo        ; ссылка на память
        mov eax,dowrd ptr flat:foo   ; ссылка на память
        mov eax,offset foo           ; адрес
        mov eax,foo                  ; адрес (неоднозначный синтаксис в MASM)
```

- синтаксис **SEGMENT ... ENDS**:

```
      segname SEGMENT
          ...
      segname ENDS
```

- синтаксис **PROC ... ENDP**;

```
        procname PROC [FAR]
             ...
        procname ENDP
```

PROC также определит **RET** как макрос, расширяющийся до **RETF**, если указан **FAR**, и **RETN** в противном случае. Любое ключевое слово после **PROC**, кроме **FAR**, игнорируется.

- TBYTE ключевое слово как псевдоним для TWORD;
- END директива игнорируется.
- в 64-битном режиме по умолчанию используется относительная адресация.

Кроме того, NASM теперь изначально поддерживает, независимо от того, используется этот пакет или нет:

- ? и синтаксис **DUP** для директив объявления данных **DB** и т. д.;
- синтаксис **displacement [base + index]** для операций с памятью вместо **[base + index + displacement]**.
- **seg: [addr]** вместо синтаксиса **[seg: addr]**.
- чистое смещение может быть дано инструкцией **LEA** без квадратных скобок:

```
        lea rax,[foo]                ; standard syntax
        lea rax,foo                  ; also accepted
```