Mikl
Ушел с форума
13980 / 6996 / **810**
Регистрация: 11.11.2010
Сообщений: 12,580

Руководство по препроцессору FASM

09.09.2014, 12:53. **Просмотров** 10863. **Ответов** 7
Метки нет (Все метки)

Руководство по препроцессору FASM

перевод TAJGA FASM Tutorial by vid - FASM preprocessor guide
перевел S.T.A.S. | последняя редакция: 22 июня 2004г.

Содержание

4

Mikl
Ушел с форума
13980 / 6996 / **810**
Регистрация: 11.11.2010
Сообщений: 12,580

1. Об этом документе

Я написал это потому что вижу, как многие задают вопросы на форуме FASM, связанные с непониманием идей или особенностей препроцессора. (Я не отговариваю Вас задавать такие вопросы, непонимание чего-то - это вполне нормально, и если Ваш вопрос не чересчур сложен, кто-нибудь наверняка на него ответит).

Если Вам что-нибудь из туториала покажется непонятным, пожалуйста, напишите на форум FASM, форум WASM, автору или переводчику.

2. Общие понятия

2.1. Что такое препроцессор

Препроцессор - это программа (или чаще - часть компилятора), которая преобразует исходный текст непосредственно перед компиляцией. К примеру, если Вы используете какой-либо кусок кода довольно часто, можно дать ему некое имя и заставить препроцессор повсеместно заменять это имя в исходном тексте на соответствующий ему код.

Другой пример - Вы хотите имитировать инструкцию, которая на самом деле не существует. В таком случае препроцессор может заменять её последовательностью инструкций дающих желаемый эффект.

Препроцессор просматривает исходный текст и заменяет некоторые вещи другими. Но как объяснить препроцессору, что именно он должен делать? Для этих целей существуют директивы препроцессора. О них мы и будем говорить.

Препроцессор понятия не имеет о инструкциях, директивах компилятора и прочих подобных вещах. Для него существуют собственные команды, и он игнорирует всё остальное.

2.2. Комментарии ";"

Подобно большинству ассемблеров, комментарии в FASM начинаются с точки с запятой "**;**". Всё, что следует за этим символом до конца строки игнорируется и удаляется из исходника.

К примеру, исходный текст

Assembler Выделить код

```
1  ; заполним 100h байтов адресуемых EDI нулями
2  xor eax, eax    ; обнуляем eax
3  mov ecx, 100h/4
4  rep stosd
```

после препроцессора превращается в

Assembler Выделить код

```
1  xor eax,eax
2  mov ecx,100h/4
3  rep stosd
```

ПРИМЕЧАНИЕ: **;** можно рассматривать как директиву препроцессора, удаляющую текст начиная с этого символа до конца строки.

ПРИМЕЧАНИЕ: Строка, полностью состоящая из комментария не будет удалена. Она становится пустой строкой (см. пример выше). Это будет важно в дальнейшем.

2.3. Перенос строки (Line Break "\")

Если строка выглядит слишком длинной, возможно разделить её на несколько, используя символ "****". При обработке препроцессором следующая строка будет добавлена к текущей.

Например:

Assembler Выделить код

```
1  db  1, 2, 3,\
2      4, 5, 6,\
3      7, 8, 9
```

будет преобразовано в:

Assembler Выделить код

```
1  db  1,2,3,4,5,6,7,8,9
```

Конечно, **** в составе текстовой строки или комментария не вызовет объединения строк. Внутри текстовой строки этот символ воспринимается как обычный ASCII символ (как и всё остальное заключённое между кавычками ' или "). Комментарии же удаляются без анализа того, что в них написано.

В строке после символа **** могут быть только пробелы или комментарии.

Ранее, я упоминал, что строка, состоящая только из комментария не удаляется, а заменяется на пустую строку. Это значит, что код, подобный этому:

Assembler Выделить код

```
1  db  1, 2, 3,\
2  ;   4,5,6,\   - закомментировано
3      7, 8, 9
```

преобразуется в:

Assembler Выделить код

```
1  db  1, 2, 3
2      7, 8, 9
```

и вызовет ошибку. Выход из положения - помещать символ **** до комментария:

Assembler Выделить код

```
1  db  1, 2, 3,\
2  \;  4,5,6     - правильно закомментировано
3      7, 8, 9
```

в результате будет:

Assembler Выделить код

```
1  db  1, 2, 3, 7, 8, 9
```

как мы и хотели.

2.4. Директива *INCLUDE*

Синтаксис:

Assembler Выделить код

```
1  include <некая строка содержащая имя файла file_name>
```

Эта директива вставляет содержимое файла **file_name** в исходный текст. Вставленный текст, естественно, тоже будет обработан препроцессором. Имя файла (и путь к нему, если он есть) должны быть заключены в кавычки **"** или апострофы **'**.
Например:

Assembler Выделить код

```
1  include 'file.asm'
2  include 'HEADERS\data.inc'
3  include '..\lib\strings.asm'
4  include 'C:\config.sys'
```

Можно также использовать переменные окружения ОС, помещая их имена между символами **%**:

Assembler Выделить код

```
1  include '%FASMINC%\win32a.inc'
2  include '%SYSTEMROOT%\somefile.inc'
3  include '%myproject%\headers\something.inc'
4  include 'C:\%myprojectdir%\headers\something.inc'
```

2.5. Strings preprocessing

You may have problem to include ' in string declared using 's or " in string declared using "s. For this reason you must place the character twice into string, in that case it won't end string and begin next as you may think, but it will include character into string literaly.

For example:

Assembler Выделить код

```
1  db 'It''s okay'
```

will generate binary containing string **It's okay**.
It's same for **"**.

3. Присваивания (Equates)

3.1. Директива *EQU*

Простейшая команда препроцессора.

Синтаксис:

Assembler Выделить код

```
1  <name1> equ <name2>
```

Это команда говорит препроцессору, что необходимо заменить все последующие <**name1**> на <**name2**>.
Например:

Assembler Выделить код

```
1  count   equ 10  ; это команда препроцессора
2  mov ecx, count
```

преобразуется в:

Assembler Выделить код

```
1  mov ecx, 10
```

Ещё пример:

Assembler Выделить код

```
1  mov eax, count
2  count   equ 10
3  mov ecx, count
```

преобразуется в:

Assembler Выделить код

Assembler Выделить код

```
1  mov eax, count
2  mov ecx,10
```

потому что препроцессор заменит **count** только после директивы **equ**.
Даже это работает:

Assembler Выделить код

```
1  10  equ 11
2  mov ecx, 10
```

после обработки препроцессором, получим:

Assembler Выделить код

```
1  mov ecx, 11
```

Обратите внимание, **name1** может быть любым идентификатором. Идентификатор - это всего лишь набор символов, завершаемый пробелом (**space**), символом табуляции (**tab**), концом строки (**EOL**), комментарием **;**, символом переноса строки **** или оператором, включая операторы ассемблера и/или специальные символы вроде **,** или **}**.
name2 может быть не только единичным идентификатором, берутся все символы до конца строки. **name2** может и отсутствовать, тогда **name1** будет заменен на пустое место.

Например:

Assembler Выделить код

```
1  10  equ 11, 12, 13
2  db  10
```

получим:

Assembler Выделить код

```
1  db  11, 12, 13
```

3.2. Директива *RESTORE*

Можно заставить препроцессор прекратить заменять идентификаторы, определённые директивой **EQU**. Это делает директива **RESTORE**

Синтаксис:

Assembler Выделить код

```
1  restore <name1>
```

name1 - это идентификатор определённый ранее в директиве **EQU**. После этой команды **name1** больше не будет заменяться на **name2**.
Например:

Assembler Выделить код

```
1  mov eax, count
2  count   equ 10
3  mov eax, count
4  restore count
5  mov eax, count
```

получим:

Assembler Выделить код

```
1  mov eax, count
2  mov eax, 10
3  mov eax, count
```

Обратите внимание, что для определений сделанных директивой **EQU** работает принцип стека. То есть, если мы два раза определим один и тот же идентификатор используя **EQU**, то после однократного использования **RESTORE** значение идентификатора будет соответствовать определённому первой директивой **EQU**.
Например:

Assembler Выделить код

Assembler Выделить код

```
1   mov eax, count
2   count   equ 1
3   mov eax, count
4   count   equ 2
5   mov eax, count
6   count   equ 3
7   mov eax, count
8   restore count
9   mov eax, count
10  restore count
11  mov eax,count
12  restore count
13  mov eax,count
```

получим:

Assembler Выделить код

```
1  mov eax, count
2  mov eax, 1
3  mov eax, 2
4  mov eax, 3
5  mov eax, 2
6  mov eax, 1
7  mov eax, count
```

Если попытаться выполнить **RESTORE** большее количество раз, чем было сделано **EQU**, никаких предупреждений выдано не будет. Значение идентификатора будет неопределенно.
Например:

Assembler Выделить код

```
1  mov eax, count
2  restore count
3  mov eax, count
```

получим:

Assembler Выделить код

```
1  mov eax, count
2  mov eax, count
```

2

Mikl
Ушел с форума
13980 / 6996 / **810**
Регистрация: 11.11.2010
Сообщений: 12,580

09.09.2014, 12:56 **[ТС]** **3**

4. Простые макросы без аргументов

4.1. Определение простых макросов

Использую **EQU** можно делать наиболее простые замены в исходном тексте при обработке препроцессором. Большими возможностями обладают макросы. Командой **MACRO** можно создавать собственные инструкции.

Синтаксис:

Assembler Выделить код

```
1  macro   name
2  {
3  ; тело макроса
4  }
```

Когда препроцессор находит директиву **macro**, он определяет макрос с именем **name**. Далее, встретив в исходном тексте строку, начинающуюся с **name**, препроцессор заменит **name** на тело макроса - то, что указано в определении между скобками **{** и **}**. Имя макроса может быть любым допустимым идентификатором, а тело макроса - всё, что угодно, за исключением символа **}**, который означает завершение тела макроса.
Например:

Assembler Выделить код

```
1  macro   a
2  {
3      push eax
4  }
5  xor eax, eax
6      a
```

будет заменено на:

Assembler Выделить код

Assembler Выделить код

```
1  xor eax, eax
2  push    eax
```

Или:

Assembler Выделить код

```
1   macro   a
2   {
3   push    eax
4   }
5   macro   b
6   {
7   push    ebx
8   }
9   b
10  a
```

получим:

Assembler Выделить код

```
1  push    ebx
2  push    eax
```

Разумеется, макросы не обязательно оформлять так, как выше, можно делать и так:

Assembler Выделить код

```
1  macro   push5 {push dword 5}
2  push5
```

получим:

Assembler Выделить код

```
1  push    dword 5
```

Или:

Assembler Выделить код

```
1  macro   push5 {push dword 5
2  }
```

с тем же самым результатом. Скобочки можете размещать как хотите.

4.2. Вложенные макросы

Макросы могут быть вложенными один в другой. То есть, если мы переопределим макрос, будет использовано последнее определение. Но если в теле нового определения содержится тот же макрос, то будет использовано предыдущее определение. Посмотрите пример:

Assembler Выделить код

```
1   macro   a { mov ax, 5}
2
3   macro   a
4   {
5       a
6       mov bx, 5
7   }
8
9   macro   a
10  {
11      a
12      mov cx, 5
13  }
14      a
```

в результате получим:

Assembler Выделить код

```
1       mov ax, 5
2       mov bx, 5
3       mov cx, 5
```

Или такой пример:

Assembler Выделить код

Assembler Выделить код

```
1   macro   a {1}
2   a
3
4   macro   a {
5       a
6       2 }
7   a
8
9   macro   a {
10      a
11      3 }
12
13  a
```

получим:

Assembler Выделить код

```
1       1
2
3       1
4       2
5
6       1
7       2
8       3
```

4.3. Директива PURGE. Отмена определения макроса

Как и в случае с директивой **EQU**, можно отменить определение макроса. Для этого используется директива **PURGE** с указанием имени макроса.

Синтаксис:

Assembler Выделить код

```
1  purge   name
```

Пример:

Assembler Выделить код

```
1  a
2  macro   a {1}
3  a
4  macro   a {2}
5  a
6  purge   a
7  a
8  purge   a
9  a
```

получим:

Assembler Выделить код

```
1  a
2  1
3  2
4  1
5  a
```

Если применить PURGE к несуществующему макросу, ничего не произойдёт.

4.4. Поведение макросов

Имя макроса будет заменено его телом не только в том случае, если оно расположено в начале строки. Макрос может находиться в любом месте исходного текста, где допустима мнемоника инструкции (например, **add** или **mov**). Всё потому, что основное предназначение макросов - имитировать инструкции. Единственное исключение из этого правила - макросы недопустимы после префиксов инструкций (**rep**).

Пример:

Assembler Выделить код

```
1  macro   CheckErr
2  {
3      cmp eax, -1
4      jz  error
5  }
6
7      call    Something
8  a:  CheckErr    ; здесь макросу предшествует метка, всё Ок.
```

получим:

Assembler Выделить код

```
1      call    Something
2  a:  cmp eax,-1
3      jz  error
```

Пример #2:

Assembler Выделить код

```
1  macro   stos0
2  {
3      mov al, 0
4      stosb
5  }
6      stos0       ;это место инструкции, будет замена.
7  here:   stos0       ;это тоже место инструкции.
8      db  stos0   ;здесь инструкции не место, замены не будет.
```

получим:

Assembler Выделить код

```
1      mov al, 0
2      stosb
3  here:   mov al, 0
4      stosb
5      db  stos0
```

Возможно переопределять (**overload**) инструкции посредством макросов. Так как препроцессор ничего об инструкциях не знает, он позволяет использовать мнемонику инструкции в качестве имени макроса:

Assembler Выделить код

```
1  macro   pusha
2  {
3      push eax ebx ecx edx ebp esi edi
4  }
5  macro   popa
6  {
7      pop edi esi ebp edx ecx ebx eax
8  }
```

эти две новые инструкции будут экономить по четыре байта в стеке, так как не сохраняют ESP (правда, занимают побольше места, чем реальные инструкции  . Всё же, переопределение инструкций не всегда хорошая идея - кто-нибудь читая Ваш код может быть введён в заблуждение, если он не знает, что инструкция переопределена.
Также, возможно переопределять директивы ассемблера:

Assembler Выделить код

```
1   macro   use32
2   {
3       align   4
4       use32
5   }
6
7   macro   use16
8   {
9       align   2
10      use16
11  }
```

5. Макросы с фиксированным количеством аргументов

5.1. Макросы с одним аргументом

Макросы могут иметь аргумент. Аргумент представляет собой какой-либо идентификатор, который будет повсюду заменён в теле макроса тем, что будет указанно при использовании.

Синтаксис:

Assembler Выделить код

```
1  macro <name> <argument> { <тело макроса> }
```

Например:

Assembler Выделить код

Assembler Выделить код

```
1  macro add5 where
2  {
3      add where, 5
4  }
5
6      add5    ax
7      add5    [variable]
8      add5    ds
9      add5    ds+2
```

получим:

Assembler Выделить код

```
1      add ax, 5
2      add [variable], 5
3      add ds, 5   ;такой инструкции не существует
4              ;но препроцессор это не волнует.
5              ;ошибка появится на стадии ассемблирования.
6      add ds+2,5  ;ошибка синтаксиса, как и ранее
7              ;определится при анализе синтаксиса (parsing).
```

(разумеется, комментарии в результате работы препроцессора не появятся 

5.2. Макросы с несколькими аргументами

У макросов может быть несколько аргументов, разделённых запятыми "**,**":

Assembler Выделить код

```
1  macro movv where, what
2  {
3      push    what
4      pop where
5  }
6
7  movv    ax, bx
8  movv    ds, es
9  movv    [var1], [var2]
```

преобразуется в:

Assembler Выделить код

```
1      push    bx
2      pop ax
3
4      push    es
5      pop ds
6
7      push    [var2]
8      pop [var1]
```

Если несколько аргументов имеют одно и тоже имя, то будет использован первый из них  .

Если при использовании макроса указать меньше аргументов, чем при определении, то значения неуказанных будет пустым:

Assembler Выделить код

```
1  macro   pupush a1, a2, a3, a4
2  {
3      push    a1 a2 a3 a4
4      pop a4 a3 a2 a1
5  }
6
7      pupush  eax, dword [3]
```

получим:

Assembler Выделить код

```
1      push    eax dword [3]
2      pop dword [3] eax
```

Если в аргументе макроса необходимо указать запятую как символ ("**,**"), тогда необходимо аргумент заключить в скобочки из символов **<** и **>**.

Assembler Выделить код

Assembler Выделить код

```
1  macro   safe_declare name, what
2  {
3      if used name
4          name    what
5      end if}
6
7  safe_declare    var1, db 5
8  safe_declare    array5, <dd 1,2,3,4,5>
9  safe_declare    string, <db "привет, я просто строка",0>
```

получим:

Assembler Выделить код

```
1   if used var1
2       var1    db 5
3   end if
4
5   if used array5
6       array5  dd 1,2,3,4,5
7   end if
8
9   if used string
10      string  db "привет, я просто строка",0
11  end if
```

Конечно же, можно использовать символы **<** и **>** и внутри тела макроса:

Assembler Выделить код

```
1  macro   a arg {db arg}
2  macro   b arg1,arg2 {a <arg1,arg2,3>}
3  b   <1,1>,2
4  получим:
5  db  1,1,2,3
```

5.3. Директива "LOCAL"

Возможно, появится необходимость объявить метку внутри тела макроса:

Assembler Выделить код

```
1  macro   pushstr string
2  {
3      call    behind ;помещаем в стек адрес string и переходим к behind
4      db  string, 0
5  behind:
6  }
```

но если использовать такой макрос 2 раза, то и метка **behind** будет объявлена дважды, что приведёт к ошибке. Эта проблема решается объявлением локальной метки **behind**. Это и делает директива **LOCAL**.
Синтаксис:

Assembler Выделить код

```
1  local       label_name
```

Директива должна применяться внутри тела макроса. Все метки **label_name** внутри макроса становятся локальными. Так что, если макрос используется дважды никаких проблем не появляется:

Assembler Выделить код

```
1   macro   pushstr string
2   {
3     local behind
4       call    behind
5       db  string,0
6   behind:
7   }
8
9   pushstr 'aaaaa'
10  pushstr 'bbbbbbbb'
11  call    something
```

На самом деле, **behind** заменяется на **behind?XXXXXXXX**, где **XXXXXXXX** - какой-то шестнадцатеричный номер генерируемый препроцессором. Последний пример может быть преобразован к чему-то вроде:

Assembler Выделить код

Assembler Выделить код

```
1      call    behind?00000001
2      db  'aaaaa', 0
3  behind?00000001:
4      call    behind?00000002
5      db  'bbbbbbbb', 0
6  behind?00000002:
7      call    something
```

Заметьте, Вы не сможете напрямую обратиться к метке содержащей **?**, так как это специальный символ в FASM, поэтому он и используется в локальных метках. К примеру, **aa?bb** рассматривается как идентификатор **aa**, специальный символ **?** и идентификатор **bb**.

Если Вам нужно несколько локальных меток - не проблема, их можно указать в одной директиве LOCAL, разделив запятыми:

Assembler Выделить код

```
1  macro   pushstr string  ;делает то же, что и предыдущий макрос
2  {
3    local addr, behind
4      push    addr
5      jmp behind
6  addr    db  string,0
7  behind:
8  }
```

Всегда хорошо бы начинать все локальные метки макросов с двух точек **..** - это значит, что они не будут менять текущую глобальную метку. К примеру:

Assembler Выделить код

```
1   macro   pushstr string
2   {
3     local behind
4       call    behind
5       db  string, 0
6   behind:
7   }
8
9   MyProc:
10      pushstr 'aaaa'
11  .a:
```

будет преобразовано в:

Assembler Выделить код

```
1  MyProc:
2      call    behind?00000001
3      db  'aaaa', 0
4  behind?00000001:
5  .a:
```

в результате получим метку **behind?00000001.a** вместо **MyProc.a**. Но если в примере выше **behind** заменить на **..behind**, текущая глобальная метка не изменится и будет определена метка **MyProc.a:**

Assembler Выделить код

```
1   macro   pushstr string
2   {
3     local ..behind
4       call    ..behind
5       db  string,0
6   ..behind:
7   }
8
9   MyProc:
10      pushstr 'aaaa'
11  .a:
```

2

Mikl
Ушел с форума

13980 / 6996 / **810**
Регистрация: 11.11.2010
Сообщений: 12,580

09.09.2014, 12:57 **[TC]** **4**

5.4. Оператор объединения

У макроязыка FASMa есть ещё одна возможность - манипуляции с идентификаторами. Делается это оператором **#**, который объединяет два идентификатора в один. К примеру, **a#b** становится **ab**, а **aaa bbb#ccc ddd** -- **aaa bbbccc ddd**.

Оператор **#** может быть использован только внутри тел макросов, а объединение символов происходит после замены аргументов макроса параметрами. Так что его можно использовать для создания новых идентификаторов из переданных

в макрос параметров:

Assembler Выделить код

```
1   macro   string name, data
2   {
3     local ..start
4   ..start:
5   name db data,0
6   sizeof.#name =  $ - ..start
7   }
8
9   string  s1,'нудные макросы'
10  string  s2,<'а вот и я',13,10,'заставлю тебя их видеть во сне'>
```

получим:

Assembler Выделить код

```
1  ..start?00000001:
2  s1  db 'нудные макросы',0
3  sizeof.s1 = $ - ..start?00000001
4
5  ..start?00000002:
6  s2  db 'а вот и я',13,10,'заставлю тебя их видеть во сне',0
7  sizeof.s2 = $ - ..start?00000002
```

так что для всех строк, создаваемых этим макросом будет определён идентификатор **sizeof.имя строки**, равный количеству байт строки.
Оператор **#** способен так же объединять символьные строки:

Assembler Выделить код

```
1  macro   debug name
2  {
3      db 'name: '#b,0
4  }
5  debug   '1'
6  debug   'foobar'
```

будет:

Assembler Выделить код

```
1  db  'name: 1',0
2  db  'name: foobar',0
```

Это полезно при передаче аргументов из макроса в макрос:

Assembler Выделить код

```
1   macro   pushstring string
2   {
3     local ..behind
4       call    ..behind
5       db  string,0
6   ..behind:
7   }
8   macro   debug string
9   {
10      push    MB_OK
11      push 0 ;empty caption
12      pushstring  'debug: '#string    ;принимает один аргумент
13      push    0               ;нет окна-предка
14      call    [MessageBox]
15  }
```

Обратите внимание, нельзя использовать **#** совместно с идентификаторами, определёнными **local**, так как **local** обрабатывается препроцессором раньше, чем **#**. Из-за этого подобный код работать не будет:

Assembler Выделить код

```
1  macro   a arg
2  {
3    local name_#arg
4  }
5  a   foo
```

5.5. Оператор "`"

Существует оператор, преобразующий идентификатор в символьную строку. Он так же может быть использован только внутри макросов:

Assembler Выделить код

Assembler Выделить код

```
1  macro   proc name
2  {
3     name:
4     log  `name   ;log - макрос, принимающий параметр-строку
5  }
6  proc    DummyProc
```

получим:

Assembler Выделить код

```
1  DummyProc:
2      log 'DummyProc'
```

Пример посложнее, с использованием "**#**"

Assembler Выделить код

```
1  macro   proc name
2  {
3       name:
4       log 'начинается подпрограмма: '#`name
5  }
6  proc    DummyProc
7  retn
8  proc    Proc2
9  retn
```

будет:

Assembler Выделить код

```
1  DummyProc:
2  log 'начинается подпрограмма: DummyProc'
3  retn
4  Proc2:
5  log 'начинается подпрограмма: Proc2'
6  retn
```

6. Макросы с групповыми аргументами

6.1. Определение макросов с групповым аргументом

У макросов могут быть так называемые групповые аргументы. Это позволяет использовать переменное количество аргументов. При определении макроса, групповой аргумент заключается в квадратные скобочки "**[**" и "**]**":

Синтаксис:

Assembler Выделить код

```
1  macro   name arg1, arg2, [grouparg]
2  {
3   <тело макроса>
4  }
```

Среди аргументов в определении макроса, групповой аргумент должен быть последним. Групповой аргумент может содержать несколько значений:

Assembler Выделить код

```
1  macro   name arg1,arg2,[grouparg] {}
2  name    1,2,3,4,5,6
```

В этом примере значение **arg1** равно 1, **arg2** равно 2, а **grouparg** равно 3,4,5 и 6

6.2. Директива "*COMMON*"

Для работы с групповыми аргументами применяются специальные директивы препроцессора. Они могут быть использованы только внутри тела макроса имеющего групповой аргумент. Первая такая директива - это "**COMMON**". Она означает, что после нее имя группового аргумента будет замещаться всеми аргументами сразу:

Assembler Выделить код

```
1  macro   string [grp]
2  {
3    common
4    db    grp,0
5  }
6
7  string  'aaaaaa'
8  string  'line1',13,10,'line2'
9  string  1,2,3,4,5
```

получим:

Assembler Выделить код

```
1  db  'aaaaaa',0
2  db  'line1',13,10,'line2',0
3  db  1,2,3,4,5,0
```

6.3. Директива "*FORWARD*"

Аргументы можно обрабатывать и по-отдельности. Для этого служит директива "**FORWARD**". Часть тела макроса после этой директивы обрабатывается препроцессором для каждого аргумента из группы:

Assembler Выделить код

```
1  macro   a arg1,[grparg]
2  {
3  forward
4       db  arg1
5       db  grparg
6  }
7
8  a   1,'a','b','c'
9  a   -1, 10, 20
```

будет:

Assembler Выделить код

```
1   db  1
2   db  'a'
3   db  1
4   db  'b'
5   db  1
6   db  'c'
7
8   db  -1
9   db  10
10  db  -1
11  db  20
```

Директива "**FORWARD**" работает по умолчанию для макросов с групповыми аргументами, так что предыдущий пример можно сделать так:

Assembler Выделить код

```
1  macro   a arg1,[grparg]
2  {
3       db  arg1
4       db  grparg
5  }
```

6.4. Директива "*REVERSE*"

"**REVERSE**" - это аналог "**FORWARD**", но обрабатывает группу аргументов в обратном порядке - от последнего к первому:

Assembler Выделить код

```
1  macro   a arg1,[grparg]
2  {
3  reverse
4       db  arg1
5       db  grparg
6  }
7
8  a   1,'a','b','c'
```

получим:

Assembler Выделить код

```
1  db  1
2  db  'c'
3  db  1
4  db  'b'
5  db  1
6  db  'a'
```

6.5. Комбинирование директив управления группами

Три вышеупомянутые директивы могут разделять тело макроса на блоки. Каждый блок обработается препроцессором после предыдущего. Например:

Assembler Выделить код

Assembler Выделить код

```
1  macro   a [grparg]
2  {
3    forward
4      f_#grparg:  ;оператор объединения
5    common
6      db  grparg
7    reverse
8      r_#grparg:
9  }
10
11 a   1,2,3,4
```

будет:

Assembler Выделить код

```
1  f_1:
2  f_2:
3  f_3:
4  f_4:
5  db  1,2,3,4
6  r_4:
7  r_3:
8  r_2:
9  r_1:
```

6.6. Директива *LOCAL* в макросах с групповыми аргументами

У локальных меток в макросах есть ещё одно полезное свойство. Если директива "**LOCAL**" находится внутри блока "**FORWARD**" или "**REVERSE**", то уникальное имя метки сгенерируется для каждого аргумента из группы, и в последующих блоках "**FORWARD**" и/или "**REVERSE**" для каждого аргумента будет использована соответствующая ему метка:

Assembler Выделить код

```
1  macro   string_table [string]
2  {
3    forward       ;таблица указателей на строки
4      local   addr    ;локальная метка для строки
5      dd  addr        ;указатель на строку
6    forward       ;строки
7      addr    db string,0 ;создаём и завершаем нулём
8  }
9  string_table    'aaaaa','bbbbbb','5'
```

получим:

Assembler Выделить код

```
1  dd  addr?00000001
2  dd  addr?00000002
3  dd  addr?00000003
4  addr?00000001   db 'aaaaa',0
5  addr?00000002   db 'bbbbbb',0
6  addr?00000003   db '5',0
```

Другой пример с блоком "**REVERSE**":

Assembler Выделить код

```
1  macro   a [x]
2  {
3  forward
4    local here
5    here  db x
6  reverse
7    dd    here
8  }
9  a   1,2,3
```

будет:

Assembler Выделить код

```
1  here?00000001   db 1
2  here?00000002   db 2
3  here?00000003   db 3
4  dd  here?00000003
5  dd  here?00000002
6  dd  here?00000001
```

Как видно, метки используется с соответствующими аргументами и в "**FORWARD**"- и в "**REVERSE**"-блоках.

6.7. Макросы с несколькими групповыми аргументами

Возможно использовать и несколько групповых аргументов. В этом случае определение макроса не будет выглядеть как:

Assembler Выделить код

```
1  macro   a [grp1],[grp2]
```

так как тут не ясно какой аргумент какой группе принадлежит. Исходя из этого делают так:

Assembler Выделить код

```
1  macro   a [grp1,grp2]
```

В этом случае каждый нечётный аргумент относится к группе **grp1**, а каждый чётный - к **grp2**:

Assembler Выделить код

```
1  macro   a [grp1,grp2]
2  {
3    forward
4      l_#grp1:
5    forward
6      l_#grp2:
7  }
8
9  a   1,2,3,4,5,6
```

будет:

Assembler Выделить код

```
1  l_1:
2  l_3:
3  l_5:
4  l_2:
5  l_4:
6  l_6:
```

Или ещё:

Assembler Выделить код

```
1  macro   ErrorList [name,value]
2  {
3    forward
4      ERROR_#name = value
5  }
6  ErrorList \
7      NONE,0,\
8      OUTOFMEMORY,10,\
9      INTERNAL,20
```

получим:

Assembler Выделить код

```
1  ERROR_NONE = 0
2  ERROR_OUTOFMEMORY = 10
3  ERROR_INTERNAL = 20
```

Конечно же, может быть больше двух групп аргументов:

Assembler Выделить код

```
1  macro   a [g1,g2,g3]
2  {
3    common
4      db  g1
5      db  g2
6      db  g3
7  }
8  a   1,2,3,4,5,6,7,8,9,10,11
```

будет:

Assembler Выделить код

```
1  db  1,4,7,10
2  db  2,5,8,11
3  db  3,6,9
```

7. Условный препроцессинг

В действительности, FASM не имеет директив для условного препроцессинга. Но директива ассемблера "**if**" может быть использована совместно с возможностями препроцессора для получения тех же результатов, что и при условном

препроцессинге. (Но в этом случае увеличивается расход памяти и времени).
Как известно, оператор "**if**" обрабатывается во время ассемблирования. Это значит, что условие в этом операторе проверяется после обработки исходного текста препроцессором. Именно это обеспечивает работу некоторых логических операций.

Я не буду рассказывать о деталях времени ассемблирования (логических операциях вроде "**&**", "**|**" и тому подобном) - читайте об этом в документации FASM. Я лишь расскажу об операторах проверки условия используемых препроцессором.

2

Mikl
Ушел с форума

13980 / 6996 / **810**
Регистрация: 11.11.2010
Сообщений: 12,580

10.09.2014, 04:59 **[ТС]** **5**

7.1. Оператор "*EQ*"

Простейший логический оператор - это "**EQ**". Он всего лишь сравнивает два идентификатора - одинаковы ли их значение. Значение **abcd eq abcd** - *истина*, а **abcd eq 1** - *ложь* и так далее... Это полезно для сравнения символов, которые будут обработаны препроцессором:

Assembler Выделить код

```
1  STRINGS equ ASCII
2  if STRINGS eq ASCII
3      db  'Oh yeah',0
4  else if STRINGS eq UNICODE
5      du  'Oh yeah',0
6  else
7      display 'unknown string type'
8  end if
```

после обработки препроцессором, это примет вид:

Assembler Выделить код

```
1  if ASCII eq ASCII
2      db  'Oh yeah',0
3  else if ASCII eq UNICODE
4      du  'Oh yeah',0
5  else
6      display 'unknown string type'
7  end if
```

Здесь только первое условие (**ASCII eq ASCII**) выполняется, так что будет ассемблировано только

Assembler Выделить код

```
1  db 'Oh yeah',0
```

Другой вариант:

Assembler Выделить код

```
1  STRINGS equ UNICODE   ;разница здесь, UNICODE вместо ASCII
2  if STRINGS eq ASCII
3      db  'Oh yeah',0
4  else if STRINGS eq UNICODE
5      du  'Oh yeah',0
6  else
7      display 'unknown string type'
8  end if
```

получим:

Assembler Выделить код

```
1  if UNICODE eq ASCII
2      db  'Oh yeah',0
3  else if UNICODE eq UNICODE
4      du  'Oh yeah',0
5  else
6      display 'unknown string type'
7  end if
```

Тут уже первое условие (**UNICODE eq ASCII**) будет ложно, второе (**UNICODE eq UNICODE**) - верно, будет ассемблироваться

Assembler Выделить код

```
1  du 'Oh yeah',0
```

Несколько лучшее применение этого - проверка аргументов макросов, вроде:

Assembler Выделить код

Assembler Выделить код

```
1   macro   item type,value
2   {
3      if type eq BYTE
4       db  value
5      else if type eq WORD
6       dw  value
7      else if type eq DWORD
8       dd  value
9      else if type eq STRING
10      db  value,0
11     end if
12  }
13  item    BYTE,1
14  item    STRING,'aaaaaa'
```

будет:

Assembler Выделить код

```
1   if BYTE eq BYTE
2       db  1
3   else if BYTE eq WORD
4       dw  1
5   else if BYTE eq DWORD
6       dd  1
7   else if BYTE eq STRING
8       db  1,0
9   end if
10  if STRING eq BYTE
11      db  'aaaaaa'
12  else if STRING eq WORD
13      dw  'aaaaaa'
14  else if STRING eq DWORD
15      dd  'aaaaaa'
16  else if STRING eq STRING
17      db  'aaaaaa',0
18  end if
```

ассемблироваться будут только две команды:

Assembler Выделить код

```
1  db  1
2  db  'aaaaaa',0
```

Подобно всем другим операторам препроцессора, "**EQ**" может работать с пустыми аргументами. Это значит, что, например, "**if eq**" верно, а **if 5 eq** - *ложно* и т.п.

Пример макроса:

Assembler Выделить код

```
1  macro   mov dest,src,src2
2  {
3    if src2 eq
4      mov dest, src
5    else
6      mov dest, src
7      mov src, src2
8    end if
9  }
```

здесь, если есть третий аргумент, то будут ассемблироваться две последних команды, если нет - то только первая.

7.2. Оператор "*EQTYPE*"

Ещё один оператор - "**EQTYPE**". Он определяет, одинаков ли тип идентификаторов.
Существующие типы:

- отдельные строки символов, заключённые в кавычки (те, которые не являются частью численных выражений)
- вещественные числа
- любые численные выражения, например, **2+2** (любой неизвестный символ будет рассматриваться как метка, так что он будет считаться подобным выражением)
- адреса - численные выражения в квадратных скобках (учитывая оператор размерности и префикс сегмента)
- мнемоники инструкций
- регистры
- операторы размерности
- операторы **NEAR** и **FAR**
- операторы **USE16** и **USE32**
- пустые аргументы (пробелы, символы табуляции)

Пример макроса, который позволяет использовать переменную в памяти в качестве счётчика в инструкции **SHL** (например **shl ax, [myvar]**):

Assembler Выделить код

```
1   macro   shl dest, count
2   {
3     if count eqtype [0]       ;если count - ячейка памяти
4       push    cx
5       mov cl, count
6       shl dest, cl
7       pop cx
8     else              ;если count другого типа
9       shl dest, count ;просто используем обычную shl
10    end if
11  }
12  shl ax, 5
13  byte_variable   db 5
14  shl ax, [byte_variable]
```

получится:

Assembler Выделить код

```
1   if 5 eqtype [0]
2       push    cx
3       mov cl, 5
4       shl ax, cl
5       pop cx
6   else
7       shl ax, 5
8   end if
9   byte_variable   db 5
10
11  if [byte_variable] eqtype [0]
12      push    cx
13      mov cl, [byte_variable]
14      shl ax, cl
15      pop cx
16  else
17      shl ax, [byte_variable]
18  end if
```

в результате обработки условий конечный результат будет:

Assembler Выделить код

```
1       shl ax, 5
2  byte_variable   db 5
3       push    cx
4       mov cl, [byte variable]
5       shl ax, cl
6       pop cx
```

Заметьте, что **shl ax, byte [myvar]** не будет работать с этим макросом, так как условие **byte [variable] eqtype [0]** не выполняется. Читаем дальше.
Когда мы сравниваем что-то посредством "**EQTYPE**", то это что-то может быть не только единичным идентификатором, но и их комбинацией. В таком случае, результат **eqtype** *истина*, если не только типы, но и порядок идентификаторов совпадают. К примеру, **if eax 4 eqtype ebx name** - верно, так как **name** - это метка, и её тип - численное выражение.

Пример расширенной инструкции **mov**, которая позволяет перемещать данные между двумя ячейками памяти:

Assembler Выделить код

```
1   macro   mov dest,src
2   {
3     if dest src eqtype [0] [0]
4       push    src
5       pop dest
6     else
7       mov dest,src
8     end if
9   }
10
11  mov [var1], 5
12  mov [var1], [var2]
```

преобразуется препроцессором в:

Assembler Выделить код

Assembler Выделить код

```
1   if [var1] 5 eqtype [0] [0]  ;не верно
2       push    5
3       pop [var1]
4   else
5       mov [var1],5
6   end if
7
8   if [var1] [var2] eqtype [0] [0]  ;верно
9       push    [var2]
10      pop [var1]
11  else
12      mov [var1], [var2]
13  end if
```

и будет ассемблировано в:

Assembler Выделить код

```
1       mov [var1], 5
2       push    [var2]
3       pop [var1]
```

Хотя более удобно для восприятия реализовать макрос используя **логический оператор *И*** - **&**:

Assembler Выделить код

```
1  macro   mov dest,src
2  {
3    if (dest eqtype [0]) & (src eqtype [0])
4      push    src
5      pop dest
6    else
7      mov dest, src
8    end if
9  }
```

Пример с использованием "**EQTYPE**" с четырьмя аргументами приведён для демонстрации возможностей, обычно проще использовать в таких случаях "**&**". Кстати, в качестве аргументов, возможно использовать некорректные выражения - достаточно, чтобы лексический анализатор распознал их тип. Но это не является документированным, так что не будем этот обсуждать.

7.3. Оператор "*IN*"

Бывают случаи, когда в условии присутствует слишком много "**EQ**":

Assembler Выделить код

```
1   macro   mov a,b
2   {
3     if (a eq cs) | (a eq ds) | (a eq es) | (a eq fs) | \
4        (a eq gs) | (a eq ss)
5       push    b
6       pop a
7     else
8       mov a, b
9     end if
10  }
```

Вместо применения множества **логических операторов *ИЛИ*** - **|**, можно использовать специальный оператор "**IN**". Он проверяет, присутствует ли идентификатор слева, в списке идентификаторов справа. Список должен быть заключён в скобочки "**<**" и "**>**", а идентификаторы в нём разделяются запятыми:

Assembler Выделить код

```
1  macro   mov a,b
2  {
3    if a in <cs,ds,es,fs,gs,ss>
4      push    b
5      pop a
6    else
7      mov a, b
8    end if
9  }
```

Это так же работает для нескольких идентификаторов (как и "**EQ**"):

Assembler Выделить код

```
1  if dword [eax] in <[eax], dword [eax], ptr eax, dword ptr eax>
```

8. Структуры

В FASM, структуры практически тоже самое, что и макросы. Определяются они посредством директивы **STRUC**:
Синтаксис:

Assembler Выделить код

```
1  struc   <name> <arguments>  { <тело структуры> }
```

Отличие от макросов заключается в том, что в исходном тексте перед структурой должна находиться некое "**имя**" - *имя объекта-структуры*. Например:

Assembler Выделить код

```
1  struc   a {db 5}
2  a
```

это не будет работать. Структуры распознаются только после имен, как здесь:

Assembler Выделить код

```
1  struc   a {db 5}
2  name    a
```

подобно макросу, это преобразуется препроцессором в:

Assembler Выделить код

```
1  db  5
```

Смысл имени в следующем - оно будет добавлена ко всем идентификаторам из тела структуры, которые начинаются с точки. Например:

Assembler Выделить код

```
1  struc   a {.local:}
2  name1   a
3  name2   a
```

будет:

Assembler Выделить код

```
1  name1.local:
2  name2.local:
```

Таким образом можно создавать структуры вроде тех, что есть в языках высокого уровня абстракции:

Assembler Выделить код

```
1  struc   rect left,right,top,bottom  ;аргументы как у макроса
2  {
3      .left   dd left
4      .right  dd right
5      .top    dd top
6      .bottom dd bottom
7  }
8  r1 rect 0,20,10,30
9  r2 rect ?,?,?,?
```

получим:

Assembler Выделить код

```
1  r1.left     dd 0
2  r1.right    dd 20
3  r1.top      dd 10
4  r1.bottom   dd 30
5  r2.left     dd ?
6  r2.right    dd ?
7  r2.top      dd ?
8  r2.bottom   dd ?
```

Поскольку, используемой структуре всегда должно предшествовать имя, препроцессор однозначно отличает их от макросов. Поэтому имя структуры может совпадать с именем макроса - в каждом случае будет выполняться нужная обработка.

Существуют хитрый приём, позволяющий не указывать аргументы, если они равны 0:

Assembler Выделить код

```
1  struc   ymmv arg
2  {
3      .member dd arg+0
4  }
5  y1  ymmv 0xACDC
6  y2  ymmv
```

будет:

Assembler Выделить код

```
1  y1.member   dd 0xACDC+0
2  y2.member   dd +0
```

Как говорилось ранее, если значение аргумента не указанно, то в теле макроса или структуры вместо него ничего не подставляется. В этом примере "плюс" ("**+**") используется или как бинарный оператор (то есть с двумя операндами), или как унарный (с одним операндом) оператор.
ПРИМЕЧАНИЕ: часто используется так же макрос или структура **struct**, которая определяется для расширения возможностей при определении структур. Не путайте **struct** и **struc**.

3

Mikl
Ушел с форума
13980 / 6996 / **810**
Регистрация: 11.11.2010
Сообщений: 12,580

10.09.2014, 04:59 **[ТС]** **6**

9. Оператор *FIX* и макросы внутри макросов

В стародавние времена, в FASMe отсутствовала одна полезная возможность - создавать макросы внутри других макросов. Например, что бы при развёртывании макроса был бы определён новый макрос. Что-то вроде гипотетичного:

Assembler Выделить код

```
1  macro   declare_macro_AAA
2  {
3    macro AAA
4    {
5      db  'AAA',0
6    } ;завершаем определение AAA
7  }   ;завершаем определение declare_macro_AAA
```

Проблема в том, что когда макрос

Assembler Выделить код

```
1  declare_macro_AAA
```

обрабатывается препроцессором, первая найденная скобочка "**}**" считается завершением определения его, а не так как хотелось бы. Так же происходит и с другими символами и/или операторами (например, "**#**", "`", "**forward**", "**local**").

9.1. Explaination of fixes

Но со временем, была добавлена новая директива. Она работает подобно "**EQU**", но обрабатывается до любого другого препроцессинга. (За исключением предварительных операций, про которые говорится в разделе "Общие понятия" - они выполняются как бы до самого препроцессинга, но это уже внутренние детали, не слишком интересные). Директива эта называется **FIX**:

Синтаксис :

Assembler Выделить код

```
1  <name1> <fix name2>
```

Видно, что синтаксис такой же как у "**EQU**", но как я сказал, когда препроцессор обрабатывает часть кода, он смотрит, есть ли "**FIX**", а потом уже делает всё остальное. Например код:

Assembler Выделить код

```
1  a equ 1
2  b equ a
3  a b
```

Then preprocesisng happens like this:

Preprocessing line 1:
a - Preprocessor finds unknown word, skips it.
equ - "equ" is second word of line, so it remembers "a" equals rest of line ("b") and deletes line
Preprocessing line 2:
b - Preprocessor finds unknown word, skips it.
equ - "equ" is second word of line, so it remembers "b" equals rest of line ("a") and deletes line
Preprocessing line 3:
a - Preprocessor replaces "a" with "1"
b - Preprocessor replaces "b" with "a"

So it becomes:

Assembler Выделить код

```
1  1 a
```

But if we have

Assembler Выделить код

Assembler Выделить код

```
1  a fix 1
2  b fix a
3  a b
```

then it looks like:

Fixing line 1: No symbols to be fixed
Preprocessing line 1:
a - Preprocessor finds unknown word, skips it.
fix - "fix" is second word of line, so it remembers "a" is fixed to rest of line ("b") and deletes line
Fixing line 2: "a" is fixed to "1", so line becomes "b fix 1"
Preprocessing line 2:
b - Preprocessor finds unknown word, skips it.
fix - "fix" is second word of line, so it remembers "b" is fixed to rest of line ("1") and deletes line
Fixing line 3: "a" is fixed to "1", "b" is fixed to "1" so line becomes "1 1"
Preprocessing line 3:
1 - Preprocessor finds unknown word, skips it.
1 - Preprocessor finds unknown word, skips it.

This was only example to see how fixing works, usually it isn't used in this manner.

9.2. Using fixes for nested macro declaration

Now back to declaring macro inside macro - First, we need to know how are macros preprocessed. You can quite easily make it out yourself - on macro declaration macro body is saved, and when macro is being expanded preprocessor replaces line with macro usage by macro body and internally declares equates to handle arguments and continues with preprocessing of macro body. (of course it is more complicated but this is enough for understanding fixes).

So where was problem with declaring macro inside macro? First time compiler found "**}**" inside macro body it took it as end of macro body declaration, so there wasn't any way to include "**}**" in macro body. So we can easily fix this

Assembler Выделить код

```
1   macro a
2   {
3     macro b
4     %_
5        display 'Never fix before something really needs to be fixed'
6     _%
7   }
8   %_ fix {
9   _% fix }
10  a
11  b
```

Now preprocessing looks like (simplified)

1. Preprocessor loads declaration of macro "a"
2. Preprocessor loads declaration of fixes "%_" and "_%"
3. Preprocessor expands macro "a"
4. Preprocessor loads macro "b" declaration ("_%" and "%_" are fixed in each line before being handled by rest of preprocessor)
5. Preprocessor expands macro "b"

Here you see how important is placing of declaration of fixes, because macro body is fixed too before it's loaded by preprocessor. For example this won't work:

Assembler Выделить код

```
1   %_ fix {
2   _% fix }
3   macro a
4   {
5     macro b
6     %_
7        display 'Never fix before something really needs to be fixed, here you see it'
8     _%
9   }
10  a
11  b
```

Because "**%_**" and "**_%**" will be fixed before loading macro "**a**", so loading macro body will end at "**_%**" fixed to "**}**" and second "**}**" will remain there.

NOTE: Character "**%**" isn't special character for FASM's preprocessor, so you use it just like any normal character, like "**a**" or "**9**". It has special meaning AFTER preprocessing, and only when it is only char in whole word ("**%**" not "**anything%anything**").

We also need to fix other macro-releated operators:

Assembler Выделить код

Assembler Выделить код

```
1  %_ fix {
2  _% fix }
3  %local fix local
4  %forward fix forward
5  %reverse fix revese
6  %common fix common
7  %tostring fix `
```

Only **#** is special case, you can fix it, but there is a easier way. Every time preprocessor finds multiple **#**s, it removes one, so it is something like (this won't really work)

Assembler Выделить код

```
1  etc...
2  ###### fix #####
3  #####  fix ####
4  ####   fix ###
5  ###    fix ##
6  ##     fix #
```

So instead of using symbol fixed to "**#**" you can just use "**##**" etc.

9.3. Using fixes for moving parts of codes

You can also use fixes to move parts of code. In assembly programming is this useful especially when you break code into modules, but you want to have data and code grouped in separate segment/section, but defined in one file.

Right now this part of tutorial is **TODO**, I hope I will write it soon, for now you can look at JohnFound's Fresh's macro library, file

Assembler Выделить код

```
1  INCLUDE\MACRO\globals.inc
```

Я знаю, **FIX**ы могут смутить, и хорошо бы понимать внутренние детали работы препроцессора, но они предоставляют очень большие возможности. Создатель FASM'а сделал его настолько мощным, на сколько это возможно, даже за счёт некоторого ущерба удобочитаемости.

Assembler Выделить код

```
1  a   equ 10
2  b   fix 10
3  mov ax, a
4  mov bx, b
```

будет преобразован в:

Assembler Выделить код

```
1  mov ax, 10
2  mov bx, 10
```

Но при обработке такого кода:

Assembler Выделить код

```
1   equ fix =
2  a   equ 10
3  mov ax, a
```

в первой строк директива **FIX** скажет препроцессору поменять все **EQU** на **=**. Далее, перед обработкой следующей строки, препроцессор проверит, нет ли там пофиксеных идентификаторов. Так что в нашей второй строке **equ** будет заменено на **=**, и строка примет вид **a = 10**. Так что никакой другой обработки этой строки не будет выполнено. А значит, и третья строка не будет преобразовываться препроцессором, так как идентификатор a не будет определён директивой **EQU**. Результат всего этого будет такой:

Assembler Выделить код

```
1  a   = 10
2  mov ax, a
```

Директива **FIX** может быть использован и для определения макросов в макросах - того, что мы хотели сделать в нашем гипотетичном примере. Делается это подобным образом:

Assembler Выделить код

Assembler Выделить код

```
1   macro   declare_macro_AAA
2   {
3     macro AAA
4     %_
5       db  'aaa',0
6     _%
7   }
8
9   %_ fix {
10  _% fix }
11
12  declare_macro_AAA
```

Здесь, препроцессор найдёт объявление макроса **declare_macro_AAA** и определит его, далее будет два **FIX**, и потом использование макроса **declare_macro_AAA**. Так что он преобразует это в:

Assembler Выделить код

```
1   macro   declare_macro_AAA
2   {
3     macro AAA
4     %_
5       db  'aaa',0
6     _%
7   }
8
9   %_ fix {
10  _% fix }
11
12  macro   AAA
13  %_
14      db  'aaa',0
15  _%
```

и теперь уже содержимое нового макроса будет обработано препроцессором. Далее будут заменены аргументы FIXов, и получится:

Assembler Выделить код

```
1   macro   declare_macro_AAA
2   {
3     macro AAA
4     %_
5       db  'aaa',0
6     _%
7   }
8
9   macro   AAA
10  {
11      db  'aaa',0
12  }
```

как мы и хотели.
Подобным образом можно пофиксить все остальные проблематичные вещи:

Assembler Выделить код

```
1   macro   declare_macro_TEXT
2   {
3     macro TEXT [arg]
4     %_
5       %forward
6       db  %x arg
7     _%
8   }
9
10  %_ fix {
11  _% fix }
12  %forward fix forward
13
14  declare_macro_TEXT
15
16  %x fix `
17
18  TEXT abc,def
```

В этом примере нужно обратить внимание на один момент: строка **%x fix `** должна находиться после **declare_macro_TEXT**. Если б она находилась до, то **%x** было бы пофиксено во время развёртывания макроса, и тогда **`arg** приняло бы вид **'arg'**, следовательно макрос **TEXT** был бы объявлен так:

Assembler Выделить код

Assembler Выделить код

```
1  macro   TEXT [arg]
2  {
3    forward
4      db 'arg'    ;строка не зависит от аргументов
5  }
```

Но, в нашем случае он будет:

Assembler Выделить код

```
1  macro   TEXT [arg]
2  {
3    forward
4      db `arg     ;имена аргументов превращаются в строки
5  }
```

Этот пример показывает, как важно местонахождение **FIX**.
Иногда необходимо фиксить идентификаторы дважды:

Assembler Выделить код

```
1   macro m1
2   {
3     macro m2
4     %_
5       macro m3 [arg]
6       %%_
7         db arg
8       _%%
9     _%
10  }
11
12  %%_ fix %_
13  _%% fix _%
14  %_ fix {
15  %_ fix }
16
17  m1
18  m2
19  m3
```

Символы фиксятся даже во время препроцессинга других **FIX**, так что код выше не будет работать, если порядок будет такой:

Assembler Выделить код

```
1  %_ fix {
2  %_ fix }
3  %%_ fix %_
4  _%% fix _%
```

В этом случае строка

Assembler Выделить код

```
1  %%_ fix %_
```

была бы пофиксена сразу же после

Assembler Выделить код

```
1  %_ fix {
```

, так что все последующие **%%_** сразу же преобразовались бы в **}**. То же самое и для

Assembler Выделить код

```
1  _%% fix _%
```

.

Заключение

Не забывайте читать документацию FASM. Практически всё, что есть в туториале, можно найти там. Может быть написано и немного сложнее для изучения, но лучше подойдёт в качестве справочной информации. Не так сложно запомнить - 99% пользователей FASM научились его использовать по этой документации и при помощи форума.

3

Полный 30h
Эксперт быдлокодинга
2084 / 518 / **68**

18.09.2015, 01:41 **7**

Mikl___, не знаю, в тему ли и быть может об этом тут уже где то писалось, но я не встречал, а поделиться хочется. Если где то уже есть, не сочти за труд удали мою телегу.

Регистрация: 04.11.2010
Сообщений: 1,293

По локальным переменным
В отличие от простых переменных, адрес локальной переменной просто так в регистры не пишется.

Assembler — Выделить код

```
1  locals
2    Var_loc rq 1
3  endl
4
5  mov rax,Var_loc ;выдаст ошибку компиляции
6  lea rax,[Var_loc] ; а так нет
7
8  invoke ReadFile,[hFile],[Adr],[Size], Var_loc,0 ; выдаст ошибку компиляции
9  invoke ReadFile,[hFile],[Adr],[Size], addr Var_loc,0 ; а так нет
```

Второй момент касается 64 битного FASM макроса invoke
Если по какой либо причине (с оказией) удалось запихнуть один (или все первые четыре) параметры в положенные регистры, то при подстановке этих регистров "масла масляного" не происходит.
Т.е. при написании

Assembler — Выделить код

```
1  xor rcx,rcx
2  mov rdx,String
3  xor r8,r8
4  invoke MessageBox, rcx, rdx, r8,  MB_OK
```

Макрос "понимает" где родные регистры и не пытается их загрузить самих в себя тем самым создавая избыточный код.
Кактотаг

1

Полный 30h
Эксперт быдлокодинга
2084 / 518 / **68**
Регистрация: 04.11.2010
Сообщений: 1,293

10.04.2016, 11:01 — **8**

Очередная мелочёвка из цикла "хозяйке на заметку"
В интернетах читал что помимо инициированных данных

Assembler — Выделить код

```
1  ABC db 1
```

имеются так же не инициированные

Assembler — Выделить код

```
1  ABC db ?
```

или вовсе резервирование (или что то типа того)

Assembler — Выделить код

```
1  ABC rb 1
```

По поводу первого типа представления мне всегда всё было менее ясно - исходный код программы несёт в себе какие то данные пользуемые по мере необходимости. По поводу второго и третьего типа интернеты уверяли что дескать программа только декларирует свои намерения. Не занимая при этом лишние байты под этот тип данных. Однако это не так. Вернее не совсем так. Потому что как выяснилось, что интернеты скромно умолчали о том, что неинициированные данные остаются таковыми только в том случае если они находятся в самом конце секции данных и не подпёрты инициированными переменными. В этом не трудно убедится скомпилировав программу следующим образом

Assembler — Выделить код

```
1  section '.data' data readable writeable
2    INI       DB 1
3    MASSIFF_1 RB 1000h
4    MASSIFF_2 DB 1000h dup ?
```

а потом вот так

Assembler — Выделить код

```
1  section '.data' data readable writeable
2    MASSIFF_1 RB 1000h
3    MASSIFF_2 DB 1000h dup ?
4    INI       DB 1
```

И оценить размеры полученных файлов в первом и во втором случаях.

З.Ы. Вполне допускаю что данное наблюдение всем давно известно и прописано во всех мануалах. Однако я человек темный и ленивый, мануалов не читаю, ассемблер учу из под палки. Поэтому вот только дозрел. Короче если инфа не актуальна, то снесите что бы не захламлять ветку.

0