Учебный курс. Часть 29. Макросы PROC и ENDP

Автор: xrnd | Рубрика: Учебный курс | 06-12-2010 |  **Распечатать запись**

Наверно, вы заметили, что довольно неудобно обращаться к параметрам и локальным переменным, указывая смещения относительно регистра BP. Такой способ подходит только для совсем простых и маленьких процедур. Поэтому в таких ассемблерах, как TASM и MASM, существуют специальные директивы, позволяющие создавать процедуры быстро и удобно. В FASM таких директив нет! Но они и не нужны — то же самое можно сделать с помощью макросов.

Для начала, нам потребуется заголовочный файл с макросами. Стандартный пакет FASM для Windows, к сожалению, не включает в себя макросы для 16-битных процедур. Однако такие макросы можно найти на официальном форуме FASM или скачать здесь: PROC16.INC. Это переделанная версия файла PROC32.INC с точно таким же синтаксисом.

Заголовочный файл необходимо будет включить в программу с помощью директивы *include*:

```
1 include 'proc16.inc'
```

Базовый синтаксис объявления процедуры

Для создания процедуры используется следующий синтаксис:

```
proc < имя_процедуры>[,][< список_параметров>]
    ...
    ret
endp
```

После *proc* указывается имя процедуры. Далее через запятую список параметров. Между именем процедуры и списком параметров запятую ставить не обязательно (можно просто поставить пробел). Для возврата из процедуры следует использовать команду RET без операндов. Завершается процедура макросом *endp*. Например, объявим процедуру с тремя параметрами:

```
proc myproc,a,b,c
    mov ax,[b]
    ...
    ret
endp
```

Внутри процедуры обращаться к параметрам можно как к простым переменным — с помощью квадратных скобок! При вызове процедуры параметры должны помещаться в стек, начиная с последнего. Если запустить программу в отладчике, то можно увидеть сгенерированный макросами код (вместо 3-х точек я поставил команду NOP):

```
cs:012B 55              push    bp
cs:012C 89E5            mov     bp,sp
cs:012E 8B4606          mov     ax,[bp+06]
cs:0131 90              nop
cs:0132 C9              leave
cs:0133 C20600          ret     0006
```

Макросы создали нужный пролог и эпилог процедуры. Команда RET дополнительно выталкивает 6 байт из стека (это как раз 3 параметра-слова).

Указание размеров параметров

По умолчанию размер параметров считается равным ширине стека, то есть в нашем случае 16-бит. Если требуется передавать процедуре байт или двойное слово, то нужно дополнительно указать размер. Это можно сделать, поставив двоеточие после имени параметра. Возможные операторы размера перечислены в таблице:

Оператор	Размер в байтах
BYTE	1
WORD	2

DWORD	4
PWORD	6
QWORD	8
TBYTE	10
DQWORD	16

Пусть первый параметр процедуры будет иметь размер байт, второй — слово, третий — двойное слово:

```
proc myproc,a:BYTE,b:WORD,c:DWORD
    mov cl,[a]
    mov bx,[b]
    mov ax,word[c]
    mov dx,word[c+2]
    ...
    ret
endp
```

В результате будет сгенерирован такой код:

```
cs:012B 55              push    bp
cs:012C 89E5            mov     bp,sp
cs:012E 8A4E04          mov     cl,[bp+04]
cs:0131 8B5E06          mov     bx,[bp+06]
cs:0134 8B4608          mov     ax,[bp+08]
cs:0137 8B560A          mov     dx,[bp+0A]
cs:013A 90              nop
cs:013B C9              leave
cs:013C C20800          ret     0008
```

Первый параметр будет занимать в стеке одну ячейку, хотя использоваться будет только младший байт. Третий параметр займёт 2 смежные ячейки стека. Для избежания путаницы, я рекомендую вам всегда использовать только параметры размером слово (или двойное слово в 32-битном ассемблере). Дело в том, что поместить отдельный байт в стек всё равно не получится 🙂 А *DWORD* придётся заталкивать двумя командами, поэтому лучше разделить его на два слова, явно указав младшую и старшую часть.

Указание соглашений вызова

Дополнительно для процедуры можно указать соглашения вызова. Чтобы это сделать, надо добавить специальное ключевое слово после имени процедуры. Доступно всего 2 варианта:

Ключевое слово/ соглашения вызова	Порядок помещения параметров в стек	Очищает стек
stdcall	обратный	вызываемая процедура
c	обратный	вызывающий код

Буква *c* здесь обозначает соглашения вызова для языка Си. По умолчанию используется *stdcall*. Например, пусть наша процедура не очищает стек:

```
proc myproc c,a:BYTE,b:WORD,c:DWORD
    mov cl,[a]
    mov bx,[b]
    mov ax,word[c]
    mov dx,word[c+2]
    ...
    ret
endp
```

В результате получится следующий код (сравните с предыдущим примером и найдите одно отличие!):

```
cs:012F 55              push    bp
cs:0130 89E5            mov     bp,sp
cs:0132 8A4E04          mov     cl,[bp+04]
cs:0135 8B5E06          mov     bx,[bp+06]
cs:0138 8B4608          mov     ax,[bp+08]
cs:013B 8B560A          mov     dx,[bp+0A]
cs:013E 90              nop
cs:013F C9              leave
cs:0140 C3              ret
```

Сохранение и восстановление используемых регистров

Макросы PROC и ENDP позволяют также организовать сохранение и восстановление регистров, используемых кодом процедуры. Для этого после имени процедуры нужно указать ключевое слово *uses* и список регистров через пробел. Регистры будут помещены в стек при входе в процедуру (в порядке их записи) и восстановлены перед

возвратом. Например, добавим сохранение регистров к нашей процедуре:

```
proc myproc c uses ax bx cx dx,a:BYTE,b:WORD,c:DWORD
    mov cl,[a]
    mov bx,[b]
    mov ax,word[c]
    mov dx,word[c+2]
    ...
    ret
endp
```

Если объявление процедуры получается слишком длинным, можно продолжить его на следующей строке, добавив символ \ в конец первой строки (это работает и с любыми другими макросами):

```
proc myproc4 c uses ax bx cx dx,\
           a:BYTE,b:WORD,c:DWORD
```

Результат:

```
 cs:012F 55              push   bp
 cs:0130 89E5            mov    bp,sp
 cs:0132 50              push   ax
 cs:0133 53              push   bx
 cs:0134 51              push   cx
 cs:0135 52              push   dx
 cs:0136 8A4E04          mov    cl,[bp+04]
 cs:0139 8B5E06          mov    bx,[bp+06]
 cs:013C 8B4608          mov    ax,[bp+08]
 cs:013F 8B560A          mov    dx,[bp+0A]
 cs:0142 90              nop
 cs:0143 5A              pop    dx
 cs:0144 59              pop    cx
 cs:0145 5B              pop    bx
 cs:0146 58              pop    ax
 cs:0147 C9              leave
 cs:0148 C3              ret
```

Объявление локальных переменных

Для объявления локальных переменных существует 3 варианта синтаксиса:

1. *local* + директивы объявления данных

Макрос *local* предназначен для создания локальных переменных. После слова *local* локальные переменные объявляются обычными директивами. Можно использовать как инициализированные, так и неинициализированные переменные. Можно объявлять несколько переменных в одной строке через запятую (однако, не получится объявить массив, перечислив значения). Например:

```
proc myproc c uses ax bx cx dx,\
           a:BYTE,b:WORD,c:DWORD
local i db 5
local j dw ?, k rd 1
    mov cl,[a]
    mov bx,[b]
    mov ax,word[c]
    mov dx,word[c+2]
    add cl,[i]
    mov [j],bx
    mov word[k],ax
    mov word[k+2],dx
    ret
endp
```

После объявления обращаться к локальным переменным можно также, как к параметрам и глобальным переменным. Теперь посмотрим на код:

```
 cs:012F 55              push   bp
 cs:0130 89E5            mov    bp,sp
 cs:0132 83EC08          sub    sp,0008
 cs:0135 50              push   ax
 cs:0136 53              push   bx
 cs:0137 51              push   cx
 cs:0138 52              push   dx
 cs:0139 C646F805        mov    byte ptr [bp-08],05
 cs:013D 8A4E04          mov    cl,[bp+04]
 cs:0140 8B5E06          mov    bx,[bp+06]
 cs:0143 8B4608          mov    ax,[bp+08]
 cs:0146 8B560A          mov    dx,[bp+0A]
 cs:0149 024EF8          add    cl,[bp-08]
 cs:014C 895EF9          mov    [bp-07],bx
 cs:014F 8946FB          mov    [bp-05],ax
 cs:0152 8956FD          mov    [bp-03],dx
 cs:0155 5A              pop    dx
 cs:0156 59              pop    cx
 cs:0157 5B              pop    bx
 cs:0158 58              pop    ax
 cs:0159 C9              leave
 cs:015A C3              ret
```

В стеке для локальных переменных выделяется 8 байт памяти. Первая переменная находится по самому младшему адресу (ближе к вершине стека). Сгенерировалась также команда, присваивающая значение первой переменной (сразу после сохранения регистров).

2. Альтернативный синтаксис *local*

Альтернативный синтаксис похож на синтаксис размеров параметров. После имени переменной ставится двоеточие и оператор размера. Вместо оператора размера может быть также имя структуры (это удобно при программировании для Windows).

```
local i:BYTE
local j:WORD, k:DWORD
```

Инициализация переменных в этом варианте синтаксиса не предусмотрена, её придётся делать вручную. Зато можно легко объявлять массивы — обозначение очень напоминает языки высокого уровня:

```
local buffer[256]:BYTE      ;Локальный буфер размером 256 байт
```

3. *locals* и *endl*

Третий вариант — объявление локальных переменных в виде блока *locals* — *endl*. Используются обычные директивы объявления данных:

```
proc myproc c uses ax bx cx dx,\
            a:BYTE,b:WORD,c:DWORD
locals
  i db 5
  j dw ?
  k rd 1
endl
    mov cl,[a]
    mov bx,[b]
    mov ax,word[c]
    mov dx,word[c+2]
    add cl,[i]
    mov [j],bx
    mov word[k],ax
    mov word[k+2],dx
```

```
    ret
endp
```

Этот способ хорошо подходит, если в процедуре много локальных переменных.

Макросы для вызова процедур

В файле PROC16.INC объявлены также макросы для удобного вызова процедур. Эти макросы избавляют от необходимости писать несколько команд PUSH для помещения параметров в стек. Вместо этого достаточно написать одну строку, в которой указывается имя процедуры и список параметров через запятую. Например:

```
    stdcall <имя_процедуры>[,<список_параметров>]
```

Всего существует 4 разных макроса, они перечислены в таблице. Два последних макроса чаще используются в программировании для Windows. Они выполняют вызов процедуры, адрес которой находится в переменной.

Макрос	Описание
stdcall	Вызов процедуры с соглашениями вызова *stdcall*
ccall	Вызов процедуры с соглашениями вызова *c*
invoke	То же самое, что stdcall [<имя_переменной>]
cinvoke	То же самое, что ccall [<имя_переменной>]

Так как наша процедура использует соглашения вызова *c*, то вызывать её надо макросом *ccall*:

```
    ccall myproc,5,0,ax,dx      ;3-й параметр находится в DX:AX
```

Макрос дополнительно генерирует код для восстановления указателя стека:

```
cs:0119 52              push    dx
cs:011A 50              push    ax
cs:011B 6A00            push    0000
cs:011D 6A05            push    0005
cs:011F E81300          call    0135
cs:0122 83C408          add     sp,0008
```

Важная особенность

Если вы объявили процедуру, но ни разу не вызывали, то её код не будет добавлен в исполняемый файл! Это позволяет создать свою библиотеку полезных процедур, сохранить их в отдельном файле и добавлять во все проекты, где они нужны. При этом в исполняемом файле окажутся только те процедуры, которые использует ваша программа.

Упражнение

В этот раз упражнение на дизассемблирование 🙂 Проанализируйте процедуру и напишите для неё ассемблерный код с использованием макросов PROC16. Что делает эта процедура? Подумайте, какие недостатки у неё есть и как их можно исправить.

```
cs:010A 55              push    bp
cs:010B 89E5            mov     bp,sp
cs:010D 83EC04          sub     sp,0004
cs:0110 51              push    cx
cs:0111 8B4E04          mov     cx,[bp+04]
cs:0114 E31F            jcxz    0135
cs:0116 49              dec     cx
cs:0117 7417            je      0130
cs:0119 51              push    cx
cs:011A E8EDFF          call    010A
cs:011D 8946FC          mov     [bp-04],ax
cs:0120 8956FE          mov     [bp-02],dx
cs:0123 49              dec     cx
cs:0124 51              push    cx
cs:0125 E8E2FF          call    010A
cs:0128 0346FC          add     ax,[bp-04]
cs:012B 1356FE          adc     dx,[bp-02]
cs:012E EB08            jmp     0138
cs:0130 B80100          mov     ax,0001
cs:0133 EB02            jmp     0137
cs:0135 31C0            xor     ax,ax
cs:0137 99              cwd
cs:0138 59              pop     cx
cs:0139 C9              leave
cs:013A C20200          ret     0002
```

Следующая часть »

Комментарии:

argir
08-01-2011 19:39

Ух и задали задачку… прокачали меня по стеку. Едва вылез оттуда, мало что поняв, собрал из кода программку и расписал таблицу результатов, показал дочке, она узнала последовательность Фибоначчи. Потом оказались важны места запятых в объявлении и вызове «процедурного макроса», потом… вот, что поучилось:

```
include ‘proc16.inc’

use16
org 100h

stdcall fib,12

mov ax,4C00h
int 21h
;——————————
proc fib uses bx cx,c:BYTE; в с номер элемента последовательности, выход ax
local a_1 dw 1
local a_2 dw 0
movzx cx,[c]
jcxz .p1
dec cx
jcxz .p2
@@: mov ax,[a_2]
mov bx,[a_1]
add ax,bx;суммирование предыдущих элементов
mov [a_1],ax;запись новых элементов
mov [a_2],bx; на место прежних
loop @b
jmp @f
.p2:mov ax,[a_1]
jmp @f
.p1:xor ax,ax
@@:cwd
```

ret
endp
Основной недостаток процедуры — рост стека при увеличении номера элемента.

[Ответить]

xrnd
11-01-2011 01:44

Хаха. Да, хорошее упражнение получилось 🙂

Я думал, вы реализуете с помощью макросов мою же процедуру, в смысле точно такую же.

Этот вариант тоже хороший, но диапазон чисел в 2 раза меньше. Зато без рекурсии, что намного увеличит скорость 🙂 В моём коде очень неоптимальная «двойная» рекурсия.

[Ответить]

argir
09-01-2011 11:05

Здравствуйте!
У меня ощущение, что мы сильно скакнули вперед и как бы потерялась земля под ногами. Большинство команд которые используются для создания макросов в файле proc16.inc мне не известны. Конечно эти макросы сокращают написание программ, но потерялась уверенность, что я знаю как это работает.:)

[Ответить]

xrnd
11-01-2011 01:34

Признаюсь, я тоже не до конца понимаю, как эти макросы работают 😀
Там много сложных конструкций, которые очень кратко описаны в документации к FASM или не описаны вообще.
Однако, макросы эти очень удобны. А для больших, сложных

процедур просто незаменимы. Чтобы пользоваться этими макросами, достаточно представлять, какой код они сгенерируют.

[Ответить]

Гость
28-02-2011 13:16

А что делает эта команда
mov bate ptr [bp-08] , 05
как я понимаю делает это local i db 5.

А что делает ptr — ?
И зачем добавляется , 05
[bp-08] — это последней элемент , а зачем туда 5 помешать ?

[Ответить]

Гость
28-02-2011 13:20

а ясна это не массив был ,)
а проста переменная ,запутался чуток.

[Ответить]

xrnd
01-03-2011 15:38

byte ptr — это синтаксис TASM.
в FASM «ptr» писать не нужно. Обозначает эта запись, что в квадратных скобках адрес байта.

[Ответить]

Гость
28-02-2011 17:00

Что та я совсем запутался в задание.
Например.
используется рекурсивный вызов процедуры по адресу 010A

Прерывают её 2 условия
je ; переход по адресу 0130
JCXZ ; переход по адресу 135
Но вот второй вызов рекурсивной процедуры расположен по адресу 0125
Как он может быть вызван я так и не понял.

А в самом теле процедуры используется mov cx , [bp+4]
Sp при каждом вызове -8
И при втором вызови он указывает на ту область которая будет создано sub sp,4
В этом месте я окончательна запутался .

Хотелось бы понять принцип действий этой рекурсивный процедуры. Она видима хитрая какая то.

[Ответить]

xrnd
01-03-2011 15:42

Второй рекурсивный вызов выполняется после возврата из первого.

Попробуй разобраться с параметрами и локальными переменными, заменив их адреса какими-нибудь именами. Тогда будет понятнее, что тут происходит 🙂

[Ответить]

Гость
02-03-2011 13:47

Дела не в значения а рекурсивном вызове.
Первый вызов , вывозит саму себя , потом ещё рас и так датех пор пока условие не станет ложью.
На последнем уровне вложений условие лож , происходит возврат из функции
на пред последнем уровне выполнении начнётся с ледущего оператора после call
и в регистры ax dx , будит помешено значение и после это происходит снова

рекурсивный вызов.
В котором я запутался , слишком много вложений и действий.

[Ответить]

xrnd
03-03-2011 13:32

Странно, что эта рекурсия вообще работает 😉

[Ответить]

RoverWWWorm
26-04-2011 21:47

;сложная рекурсия, вернее рекурсии, не разобрался, побоялся в отладчик взглянуть :), но вот такой получился код
[code]
include ‘c:\FASM\INCLUDE\MACRO\PROC16.INC’

use16
org 100h
;————————-
stdcall myproc,8
mov ax,4C00h
int 21h

;————————-

proc myproc uses cx,a:WORD
local p1:WORD,p2:WORD

mov cx,[a]
jcxz label1

dec cx

je label2

stdcall myproc,cx

mov [p1],ax
mov [p2],dx

dec cx

stdcall myproc,cx

add ax,[p1]
adc dx,[p2]

jmp label3

label2:
mov ax,1

jmp label4
label1:
xor ax,ax
label4:
cwd
label3:
pop cx
ret
endp
[/code]

[Ответить]

xrnd
06-05-2011 00:10

Правильно получилось.

Вот только локальных переменных не две WORD, а одна DWORD
Не случайно сложение выполняется так:

```
    add ax,[p1]
    adc dx,[p2]
```

[Ответить]

fufel
26-05-2011 20:01

Здравствуте!

```
include ‘mymacro.inc’
include ‘proc16.inc’

use16
org 100h
start:
stdcall myproc,3
exit

proc myproc,a
local i dw ?

push cx
mov cx,[a]
jcxz metka_1
dec cx
je metka_2
push cx
call myproc
mov [i],ax
mov [i+2],dx
dec cx
push cx
call myproc
add ax,[i]
adc dx,[i+2]
jmp metka_3
metka_2:
mov ax,1
jmp metka_4
metka_1:
xor ax,ax
metka_4:
cwd
metka_3:
pop cx
```

ret
endp
Вычисляет арифметическую прогрессию?

[Ответить]

xrnd
23-06-2011 15:09

Почти угадал. Похоже на арифметическую прогрессию, но не совсем то.

Локальная переменная имеет размер 4 байта (двойное слово), так как

```
    sub sp,0004
```

[Ответить]

fufel
29-08-2011 12:27

Да-а, ну и задачка! Если убрать вторую рекурсию, можно считать что я её решил?

include ‘mymacro.inc’
include ‘proc16.inc’

use16
org 100h
start:
stdcall myproc,7
exit

proc myproc,a
local i dw ?

push cx
mov cx,[a]
jcxz metka_1
dec cx

```
je metka_2
push cx
call myproc
mov [bp+8],ax
mov [bp+10],dx
add ax,[i]
adc dx,[i+2]

metka_1:
ret
endp

metka_2:
xor ax,ax
cwd
mov [bp+8],ax
mov [bp+10],dx
mov ax,1
jmp metka_1
```

[Ответить]

xrnd
17-09-2011 21:47

Да, как-то сложно получилось. Я сам уже запутался.
Ты перепутал metka_1 и metka_2.
Вот мой исходный код. Процедура вычисляет числа Фибоначчи 🙂

```
; Процедура вычисления n-го числа Фибоначчи
; n - номер числа
; DX:AX - число
proc fibonacci uses cx,n
local fn_2 dd ?
   mov cx,[n]
   jcxz .ret0
   dec cx
   jz .ret1
   stdcall fibonacci,cx          ;F (n - 2)
   mov word[fn_2],ax
   mov word[fn_2+2],dx
   dec cx
   stdcall fibonacci,cx          ;F (n - 1)
```

```
   add ax,word[fn_2]
   adc dx,word[fn_2+2]
   jmp .return
.ret1:
   mov ax,1
   jmp @f
.ret0:
   xor ax,ax
@@:
   cwd
.return:
   ret
endp
```

[Ответить]

andrew.NET
10-06-2012 16:29

В чём ошибка?[asm]
format MZ
entry main:begin
stack 0x300
include ‘sdk16.inc’
;————————————————————-
segment var
string db «Hello world$»
;————————————————————-
segment extra
proc print stdcall uses dx ax, string
mov dx, string
mov ah, 0x9
int 0x21
ret
endp

proc waitkey stdcall uses ax
mov ah, 0x08
int 0x21
ret
endp

proc exit stdcall uses ax
mov ah, 0x4C
mov al, result
int 0x21
ret
endp
;————————————————————-
segment main
begin:
invoke print, string
invoke waitkey
invoke exit[/asm]
Я не могу использовать параметры в процедуре. Ошибка возникает в
mov dx, string
Ошибка: invalid value

[Ответить]

Сергей
16-11-2016 21:54

А как написать макрос который будет на определять не числовое а строковое значение и на основании значения аргумента подставлять уже команду.
например если аргумент на входе al то делаем один набор команд если eax то другой?
что то типа:
Macro mov arg
{
if arg=al
;первый набор команд
elseif arg=eax
;второй набор команд
endif
}

[Ответить]

Ваш комментарий

Имя *

Почта (скрыта) *

Сайт

Добавить

Уведомить меня о новых комментариях по email.

Уведомлять меня о новых записях почтой.