Учебный курс. Часть 21. Простые процедуры

Автор: xrnd | Рубрика: Учебный курс | 08-07-2010 | Распечатать запись

В этой части учебного курса мы рассмотрим основы создания процедур. Процедура представляет собой код, который может выполняться многократно и к которому можно обращаться из разных частей программы. Обычно процедуры предназначены для выполнения каких-то отдельных, законченных действий программы и поэтому их иногда называют подпрограммами. В других языках программирования процедуры могут называться функциями или методами, но по сути это всё одно и то же 🙂

Команды CALL и RET

Для работы с процедурами предназначены команды CALL и RET. С помощью команды CALL выполняется *вызов* процедуры. Эта команда работает почти также, как команда безусловного перехода (JMP), но с одним отличием — одновременно в стек сохраняется текущее значение регистра IP. Это позволяет потом вернуться к тому месту в коде, откуда была вызвана процедура. В качестве операнда указывается адрес перехода, который может быть непосредственным значением (меткой), 16-разрядным регистром (кроме сегментных) или ячейкой памяти, содержащей адрес.

Возврат из процедуры выполняется командой RET. Эта команда восстанавливает значение из вершины стека в регистр IP. Таким образом, выполнение программы продолжается с команды, следующей сразу после команды CALL. Обычно код процедуры заканчивается этой командой. Команды CALL и RET не изменяют значения флагов (кроме некоторых особых случаев в защищенном режиме). Небольшой пример разных способов вызова процедуры:

```
1  use16                   ;Генерировать 16-битный код
2  org 100h                ;Программа начинается с адреса 100h
3
4      mov ax,myproc
5      mov bx,myproc_addr
6      xor si,si
7
8      call myproc         ;Вызов процедуры (адрес перехода - myproc)
9      call ax             ;Вызов процедуры по адресу в AX
10     call [myproc_addr]  ;Вызов процедуры по адресу в переменной
11     call word [bx+si]   ;Более сложный способ задания адреса ;)
12
13     mov ax,4C00h        ;\
14     int 21h             ;/ Завершение программы
15
16 ;---------------------------------------------------------------------
17 ;Процедура, которая ничего не делает
18 myproc:
19     nop                 ;Код процедуры
20     ret                 ;Возврат из процедуры
21 ;---------------------------------------------------------------------
22 myproc_addr dw myproc   ;Переменная с адресом процедуры
```

Ближние и дальние вызовы процедур

Существует 2 типа вызовов процедур. *Ближним* называется вызов процедуры, которая находится в текущем сегменте кода. *Дальний* вызов — это вызов процедуры в другом

сегменте. Соответственно существуют 2 вида команды RET — для ближнего и дальнего возврата. Компилятор FASM автоматически определяет нужный тип машинной команды, поэтому в большинстве случаев не нужно об этом беспокоиться.

В учебном курсе мы будем использовать только ближние вызовы процедур.

Передача параметров

Очень часто возникает необходимость передать процедуре какие-либо параметры. Например, если вы пишете процедуру для вычисления суммы элементов массива, удобно в качестве параметров передавать ей адрес массива и его размер. В таком случае одну и ту же процедуру можно будет использовать для разных массивов в вашей программе. Самый простой способ передать параметры — это поместить их в регистры перед вызовом процедуры.

Возвращаемое значение

Кроме передачи параметров часто нужно получить какое-то значение из процедуры. Например, если процедура что-то вычисляет, хотелось бы получить результат вычисления 🙂 А если процедура что-то делает, то полезно узнать, завершилось действие успешно или возникла ошибка. Существуют разные способы возврата значения из процедуры, но самый часто используемый — это поместить значение в один из регистров. Обычно для этой цели используют регистры AL и AX. Хотя вы можете делать так, как вам больше нравится.

Сохранение регистров

Хорошим приёмом является сохранение регистров, которые процедура изменяет в ходе своего выполнения. Это позволяет вызывать процедуру из любой части кода и не беспокоиться, что значения в регистрах будут испорчены. Обычно регистры сохраняются в стеке с помощью команды PUSH, а перед возвратом из процедуры восстанавливаются командой POP. Естественно, восстанавливать их надо в обратном порядке. Примерно вот так:

```
myproc:
    push bx             ;Сохранение регистров
    push cx
    push si
    ...                 ;Код процедуры
    pop si              ;Восстановление регистров
    pop cx
    pop bx
    ret                 ;Возврат из процедуры
```

Пример

Для примера напишем процедуру для вывода собщения в рамке и протестируем её работу, выведя несколько сообщений. В качестве параметра ей будет передаватся адрес строки в регистре BX. Строка должна заканчиваться символом ‘$’. Для упрощения процедуры можно разбить задачу на подзадачи и написать соответствующие процедуры. Прежде всего нужно вычислить длину строки, чтобы знать ширину рамки. Процедура *get_length* вычисляет длину строки (адрес передаётся также в BX) и возвращает её в регистре AX.

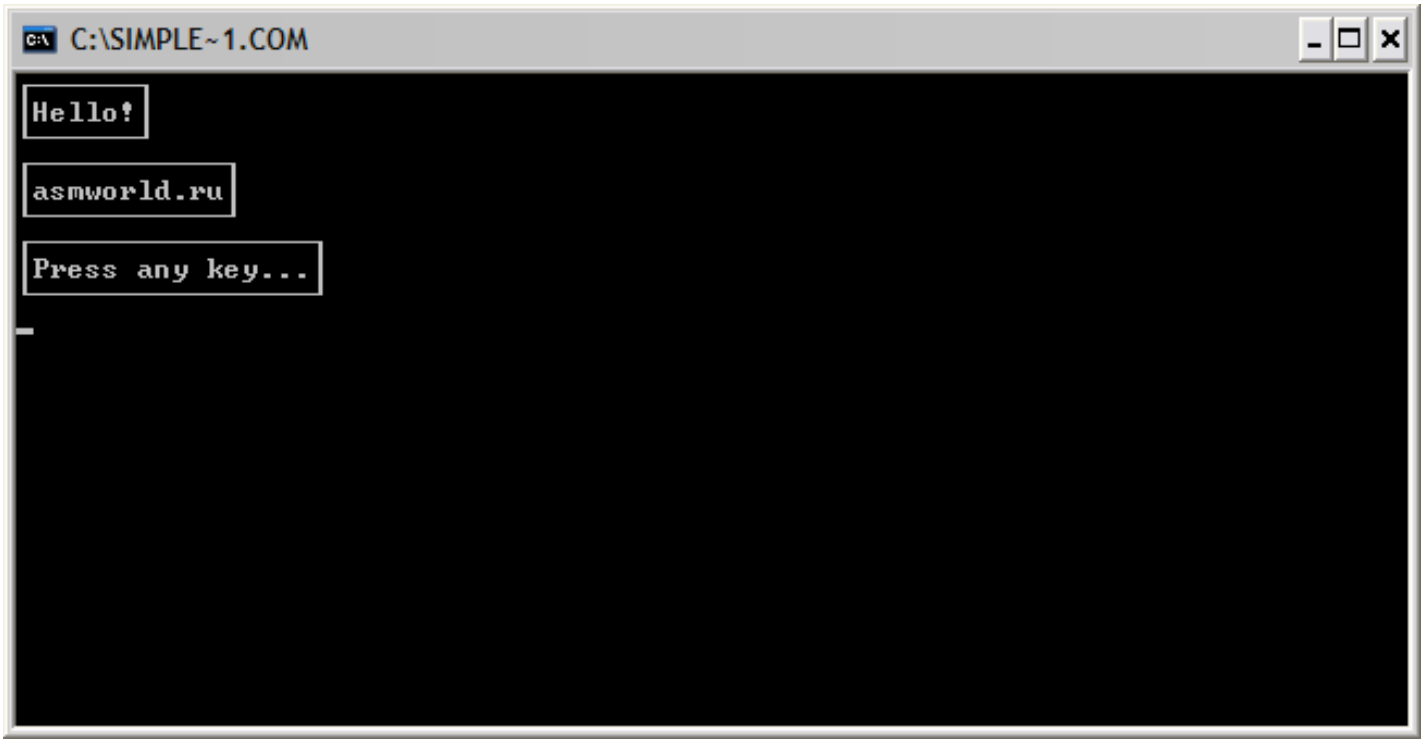
Для рисования горизонтальной линии из символов предназначена процедура *draw_line*. В DL передаётся код символа, а в CX — количество символов, которое необходимо вывести на экран. Эта процедура не возвращает никакого значения. Для вывода 2-х символов конца строки написана процедура *print_endline*. Она вызывается без параметров и тоже не возвращает никакого значения. Коды символов для рисования рамок можно узнать с

помощью таблицы символов кодировки 866 или можно воспользоваться стандартной программой Windows «Таблица символов», выбрав шрифт Terminal.

```
use16                   ;Генерировать 16-битный код
org 100h                ;Программа начинается с адреса 100h
    jmp start           ;Переход на метку start
;-----------------------------------------------------------------------
msg1    db 'Hello!$'
msg2    db 'asmworld.ru$'
msg3    db 'Press any key...$'
;-----------------------------------------------------------------------
start:
    mov bx,msg1
    call print_message  ;Вывод первого сообщения
    mov bx,msg2
    call print_message  ;Вывод второго сообщения
    mov bx,msg3
    call print_message  ;Вывод третьего сообщения

    mov ah,8            ;Ввод символа без эха
    int 21h

    mov ax,4C00h        ;\
    int 21h             ;/ Завершение программы

;-----------------------------------------------------------------------
;Процедура вывода сообщения в рамке
;В BX передаётся адрес строки
print_message:
    push ax             ;Сохранение регистров
    push cx
    push dx

    call get_length     ;Вызов процедуры вычисления длины строки
    mov cx,ax           ;Копируем длину строки в CX
    mov ah,2            ;Функция DOS 02h - вывод символа
    mov dl,0xDA         ;Левый верхний угол
    int 21h
    mov dl,0xC4         ;Горизонтальная линия
    call draw_line      ;Вызов процедуры рисования линии
    mov dl,0xBF         ;Правый верхний угол
    int 21h
    call print_endline  ;Вызов процедуры вывода конца строки

    mov dl,0xB3         ;Вертикальная линия
    int 21h
    mov ah,9            ;Функция DOS 09h - вывод строки
    mov dx,bx           ;Адрес строки в DX
    int 21h
    mov ah,2            ;Функция DOS 02h - вывод символа
    mov dl,0xB3         ;Вертикальная линия
    int 21h
    call print_endline  ;Вызов процедуры вывода конца строки

    mov dl,0xC0         ;Левый нижний угол
    int 21h
    mov dl,0xC4         ;Горизонтальная линия
    call draw_line
    mov dl,0xD9         ;Правый нижний угол
    int 21h
    call print_endline  ;Вызов процедуры вывода конца строки

    pop dx              ;Восстановление регистров
```

```
61      pop cx
62      pop ax
63      ret                 ;Возврат из процедуры
64
65  ;-------------------------------------------------------------------
66  ;Процедура вычисления длины строки (конец строки - символ '$').
67  ;В BX передаётся адрес строки.
68  ;Возвращает длину строки в регистре AX.
69  get_length:
70      push bx             ;Сохранение регистра BX
71      xor ax,ax           ;Обнуление AX
72  str_loop:
73      cmp byte[bx],'$'    ;Проверка конца строки
74      je str_end          ;Если конец строки, то выход из процедуры
75      inc ax              ;Инкремент длины строки
76      inc bx              ;Инкремент адреса
77      jmp str_loop        ;Переход к началу цикла
78  str_end:
79      pop bx              ;Восстановление регистра BX
80      ret                 ;Возврат из процедуры
81
82  ;-------------------------------------------------------------------
83  ;Процедура рисования линии из символов.
84  ;В DL - символ, в CX - длина линии (кол-во символов)
85  draw_line:
86      push ax             ;Сохранение регистров
87      push cx
88      mov ah,2            ;Функция DOS 02h - вывод символа
89  drl_loop:
90      int 21h             ;Обращение к функции DOS
91      loop drl_loop       ;Команда цикла
92      pop cx              ;Восстановление регистров
93      pop ax
94      ret                 ;Возврат из процедуры
95
96  ;-------------------------------------------------------------------
97  ;Процедура вывода конца строки (CR+LF)
98  print_endline:
99      push ax             ;Сохранение регистров
100     push dx
101     mov ah,2            ;Функция DOS 02h - вывод символа
102     mov dl,13           ;Символ CR
103     int 21h
104     mov dl,10           ;Символ LF
105     int 21h
106     pop dx              ;Восстановление регистров
107     pop ax
108     ret                 ;Возврат из процедуры
```

Результат работы программы выглядит вот так:



Отладчик Turbo Debugger

Небольшое замечание по поводу использования отладчика. В Turbo Debugger нажимайте F7 (*«Trace into»*), чтобы перейти к коду вызываемой процедуры. При нажатии F8(*«Step over»*) процедура будет выполнена сразу целиком.

Упражнение

Объявите в программе 2-3 массива слов без знака. Количество элементов каждого массива должно быть разным и храниться в отдельной 16-битной переменной без знака. Напишите процедуру для вычисления среднего арифметического массива чисел. В качестве параметров ей будет передаваться адрес массива и количество элементов, а возвращать она будет вычисленное значение. С помощью процедуры вычислите среднее арифметическое каждого массива и сохраните где-нибудь в памяти. Выводить числа на экран не нужно, этим мы займемся в следующей части 🙂 Результаты можете писать в комментариях или на форуме.

Следующая часть »

Комментарии:

fufel
12-07-2010 20:52

Вот, кажется работает. Только вот какая фигня: при делении получается целое в ax и остаток — целое в dx, а ср. арифметическое — целое+дробь. А как такое получить не знаю.

```
use16
org 100h
jmp start
;_____________
array1 dw 1,2,3,4,5,6,6
array2 dw 2,5,6,8,9
array3 dw 5,5,7,6,8,8
n1 dw 7
```

```
n2 dw 5
n3 dw 6
sr1 rw 1
sr2 rw 1
sr3 rw 1
;————————————-
start:
mov cx,[n1]
mov bx,array1
call sr_arifm
mov [sr1],ax

mov cx,[n2]
mov bx,array2
call sr_arifm
mov [sr2],ax

mov cx,[n3]
mov bx,array3
call sr_arifm
mov [sr3],ax
jmp quit

sr_arifm:
xor di,di
xor si,si
xor ax,ax
xor dx,dx
mov si,cx

lp:
add ax,[bx+di]
add di,2
jcxz return
loop lp
return:
div si
ret

quit:
mov ax,4c00h
int 21h
```

[Ответить]

xrnd
13-07-2010 20:06

Ты все правильно считаешь 🙂 Дробная часть отбрасывается при делении целых чисел, так что остаток можно не учитывать.
Программа написана правильно. Единственное что, можно не обнулять регистр SI в процедуре (но я знаю, тебе так удобнее). И команда JCXZ здесь лишняя, перехода никогда не будет, потому что внутри цикла CX не равно нулю.

[Ответить]

fufel
13-07-2010 21:06

Да, с jcxz перемудрил))) по окончании цикла все равно выполнится деление.
Спасибо ждём следующий урок.

[Ответить]

IgorKing
22-09-2010 17:12

А у меня почему-то при втором вхождении в функцию во время деления ошибка и программа завершается.

```
use16
org 100h
mov bx,array_1
mov cx,word[len_1]
call Function
mov dx,array_2
mov cx,word[len_2]
call Function
mov ax,4c00h
int 21h

Function: ; bx:адрес массива, cx:длинна массива, ax:результат
xor si,si
push cx
Sum:
add ax,word[bx+si]
add si,2
loop Sum
pop cx
div cx
ret

;_____________
array_1 dw 1111,2222,3333
array_2 dw 111,222,333,444
len_1 dw 3
len_2 dw 4
```

[Ответить]

xrnd
23-09-2010 13:52

Хорошая программа, но есть несколько ошибок:

```
mov dx,array_2
```

Здесь должен быть регистр BX.

Дальше внутри функции надо обнулить регистр AX. Иначе сумма считается неправильно. А при втором вызове элементы массива будут прибавляться к среднему арифметическому первого массива.

Ошибка возникает из-за того, что результат деления больше 65536, так как делится 32 бита на 16 бит. AX= (DX:AX)/CX, в DX старшая часть делимого, а у тебя там ошибочно оказывается адрес второго массива. Так как числа без знака, нужно перед делением обнулить регистр DX.

[Ответить]

argir
20-12-2010 23:43

В предыдущих примерах не рассмотрена проблема размерности суммы элементов массива слов, которая ограничена 65 535*65 535 — т.е. двойным словом.
И еще интересно было засовывать результаты в стек.

```
use16
org 100h
jmp start

array1 dw 1,26,789,666,564,0,1000,998,33
array2 dw 2300,7070,234,890,0,105,9999,467,9876,15000,876,43000
array3 dw
32000,62786,56789,1,345,60000,59003,34234,45789,31765,8890,456,7654,55555,23400
arl1 dw 9
arl2 dw 12
arl3 dw 15

start:
mov bx,array1
mov cx,[arl1]
call sr_ar
mov bx,array2
mov cx,[arl2]
call sr_ar
mov bx,array3
mov cx,[arl3]
call sr_ar

mov ax,4C00h
int 21h

sr_ar:
push cx ;ещё пригодиться
xor ax,ax
xor si,si
xor dx,dx ;обнуляем регистры
sym: add ax,[bx+si];суммируем элементы
jnc kon;если нет переноса суммируем дальше
inc dx ; если есть, то добавляем его в dx
kon: inc si
inc si
loop sym
pop cx
```

div cx;пригодилось
pop cx;чтоб положить что-то,сначала надо достать кое-что
push ax
push cx;возвращаем адрес на место
ret

[Ответить]

xrnd
21-12-2010 21:22

Вроде всё правильно 🙂
У тебя получается, что процедура возвращает результат через стек. А не проще было просто в AX оставить?

Красивый кусок кода:

```
    add ax,[bx+si];суммируем элементы
    jnc kon;если нет переноса суммируем дальше
    inc dx ; если есть, то добавляем его в dx
```

но можно было обойтись сложением с переносом

```
    add ax,[bx+si]
    adc dx,0
```

[Ответить]

argir
23-12-2010 08:55

Проще, но тогда я не обратил бы внимание, что использовать стек в подпрограмме надо повнимательнее, а то можно и не вернуться.
Ваш вариант кода логичней, но оба варианта занимают 3 байта, по моему мой выполнится быстрее?

[Ответить]

xrnd
23-12-2010 20:01

Врядли тут большое различие в скорости. К тому же это будет зависеть от конкретной модели процессора.
Обычно, чем меньше переходов, тем быстрее код 🙂

[Ответить]

Гость
19-01-2011 00:21

use16
org 100h
jmp start
;————————————

```
array1 dw 1,2,3,4,5,6,7
array2 dw 1,2,3,4,5
n1 dw 7
n2 dw 5
;————————————-
start:
mov bx,array1
mov ax ,[n1]
call prozidura
mov [n1],ax ; сохраняем результат
;———-2 вызов———
mov bx,array2
mov ax ,[n2]
mov [n2],ax
call prozidura
mov ax,4c00h
int 21h
;————————————
prozidura: ; BX адрес начала даных AX количество элементов al результат
mov si,0;смешение
mov cx,ax
mov dx,ax ; dl=ax
xor ax,ax
zicol:
add ax,word[bx+si]
add si,2
loop zicol
div dl
ret
```

[Ответить]

xrnd
22-01-2011 21:51

Извиняюсь, пропустил этот комментарий.

Хорошая программа, но есть один недочет.

Если результат возвращается в AL, то сохранять целиком AX — неправильно. В AH будет остаток от деления.
Нужно добавить

```
    xor ah,ah
```

перед каждым сохранением результата. Так как число без знака. Либо добавить в конец процедуры, тогда результат будет в AX.

Ещё можно одну команду MOV убрать из процедуры, если количество элементов передавать в CX или DX.

[Ответить]

Knight212
23-02-2011 20:21

```
use16
org 100h
jmp start

array1 dw 12, 8456, 0
length1 dw 3
array2 dw 56875, 1546, 154, 84, 6
length2 dw 5
array3 dw 156, 8974, 1548, 7895
length3 dw 4

start:
mov bx, array1
mov cx, [length1]
call sr_arifm
mov bx, array2
mov cx, [length2]
call sr_arifm
mov bx, array3
mov cx, [length3]
call sr_arifm

mov ax, 4C00h
int 21h

sr_arifm:
push cx
xor si, si
xor ax, ax
lp:
add ax, [bx+si]
add si, 2
loop lp
pop cx
div cx
ret
```

Считает правильно только первый раз.
Не подскажешь, в чем ошибка?

[Ответить]

xrnd
25-02-2011 22:45

Нужно обнулить DX перед командой DIV.
Так как делитель — 16-битный регистр, то делимое в DX:AX.
Во время первого деления в DX ноль, а во время второго и третьего — там оказывается остаток от предыдущего деления.
В остальном всё правильно 🙂

[Ответить]

Knight212
01-03-2011 15:23

Спасибо

[Ответить]

plan4ik
06-04-2011 00:50

[code]
```
use16
org 100h
jmp Start

;=========================[data]==
arrA dw 0xA0, 0xA1, 0xA2, 0xA3, 0xA4, 0xA5
arrAlen db 6

arrB dw 0xB0, 0xB1, 0xB2, 0xB3, 0xB4
arrBlen db 5

arrC dw 0xC0, 0xC1, 0xC2, 0xC3
arrClen db 4

average rw 3 ; arrA = 0xA2, arrB = 0xB2, arrC = 0xC1
;================================
Start:
xor si, si

mov bx, arrA
movzx cx, [arrAlen]
call get_average
mov [average+si], ax

inc si
mov bx, arrB
movzx cx, [arrBlen]
call get_average
mov [average+si], ax

inc si
mov bx, arrC
movzx cx, [arrClen]
call get_average
mov [average+si], ax

mov ax, 4c00h
int 21h

;_____________________
get_average: ; bx == arrN, cx == count, ax == return result of average
xor ax, ax
push cx
```

```
calc:
add ax, [bx]
add bx, 2
loop calc
pop cx
call divide
ret

divide: ; ax == divided, cx == divisor, ax == result
cwd
div cx
ret
[/code]
```

[Ответить]

xrnd
08-04-2011 20:21

Ошибок нет, но divide зря сделал отдельной процедурой.

Можно просто убрать команду call divide и ret после неё — работать будет также 🙂

[Ответить]

NimRoen
15-06-2011 14:05

```
use16
org 100h

   jmp main
;=======================================;
arr1 dw 834,536,1744,6890,241
arr1_length db 5
arr2 dw 486,6574,19,684
arr2_length db 4
arr3 dw 8577,985,911,648,6578,9894,245
arr3_length db 7
;=======================================;
main:
   xor cx,cx
   cld
   mov si,arr1
   mov cl,[arr1_length]
   call getAvrg ;ax — среднее арифметическое arr1
   mov si,arr2
   mov cl,[arr2_length]
   call getAvrg ;ax — среднее арифметическое arr2
   mov si,arr3
   mov cl,[arr3_length]
   call getAvrg ;ax — среднее арифметическое arr3
```

```
    mov ax,4c00h
    int 21h
;=====================================;
;si — адрес массива
;cl — длина массива
getAvrg:
    xor ax,ax
    mov bx,ax
    mov dx,ax
getAvrg_loop:
    lodsw
    add bx,ax
    adc dx,0
getAvrg_loop_end:
    loop getAvrg_loop
getAvrg_end:
    lodsb
    xor ah,ah
    xchg ax,bx
    div bx
    ret
;=====================================;
```

[Ответить]

xrnd
23-06-2011 16:10

Хорошая процедура и команды lodsw, lodsb использованы уместно.
Вот только есть привязка к тому, что размер массива находится в байте после самого массива. Если об этом забыть, можно получить трудно находимую ошибку.
Я бы лучше сохранил cx в начале командой POP, потом восстановил бы командой PUSH перед делением.

[Ответить]

NimRoen
24-06-2011 10:11

Согласен. И еще cld добавить в начало процедуры

[Ответить]

NimRoen
24-06-2011 10:13

сам себя обманул 🙂 cld уже стоит в начале программы

[Ответить]

zipfer
19-06-2011 14:46

А как все тоже самое делать под linux, так надоело для этого виртуальную машину грузить?

[Ответить]

xrnd
23-06-2011 16:17

Под линукс надо писать на 32-битном ассемблере и использовать функции этой системы. В двух словах трудно объяснить.

[Ответить]

zipfer
26-06-2011 22:56

А где можно посмотреть ман для этого, пусть даже на английском, но тока на доступном, как тут?

[Ответить]

xrnd
02-07-2011 02:00

Поищи на wasm.ru, ещё можно посмотреть примеры к FASM для linux.

[Ответить]

алекс
25-03-2012 03:23

```
use16
org 100h
jmp start
array1 dw 15,25,35,65,55
array2 dw 10,20,30,40,50,60
array3 dw 42,34,76,28,44,12,88
len1 dw 5
len2 dw 6
len3 dw 7
aver1 dw ?
aver2 dw ?
aver3 dw ?
start:
mov bx,array1
mov cx,[len1]
call ar_average
mov [aver1],bx

mov bx,array2
mov cx,[len2]
call ar_average
mov [aver2],bx

mov bx,array3
mov cx,[len3]
```

```
call ar_average
mov [aver3],bx

mov ax,4c00h
int 21h
;_____________________________________________
ar_average:
push ax
push dx
push si
xor ax,ax
xor dx,dx
mov si,cx
sum_loop:
add ax,[bx]
adc dx,0
add bx,2
loop sum_loop
div si
mov bx,ax
pop si
pop dx
pop ax
ret
```

[Ответить]

Ваш комментарий

Имя *

Почта (скрыта) *

Сайт

Добавить

Уведомить меня о новых комментариях по email.

Уведомлять меня о новых записях почтой.