МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ

А.А. Антонов

МЕТОДЫ ПРОГРАММИРОВАНИЯ

**Учебно-методическое пособие
по выполнению лабораторных работ**

**для студентов II курса
специальности 10.05.02
очной формы обучения**

Москва · 2022

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
(РОСАВИАЦИЯ)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра основ радиотехники и защиты информации

А.А. Антонов

МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие
по выполнению лабораторных работ

для студентов II курса
специальности 10.05.02
очной формы обучения

Москва
ИД Академии Жуковского
2022

УДК 004.424
ББК 6Ф7.3
А72

Рецензент:
Петров В.И. – канд. техн. наук, доцент

Антонов А.А.
А72 Методы программирования [Текст] : учебно-методическое пособие по выполнению лабораторных работ / А.А. Антонов. – М.: ИД Академии Жуковского, 2022. – 32 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Методы программирования» по учебному плану специальности 10.05.02 для студентов II курса очной формы обучения.

В учебно-методическом пособии рассматриваются руководства к лабораторным работам, получение практических навыков программирования на языке высокого уровня, применение существующих структур данных и алгоритмов. Рассматриваются методы разработки эффективного программного обеспечения.

Рассмотрено и одобрено на заседаниях кафедры 19.04.2022 г. и методического совета 21.04.2022 г.

УДК 004.424
ББК 6Ф7.3

В авторской редакции

Подписано в печать 03.11.2022 г.
Формат 60x84/16 Печ. л. 2 Усл. печ. л. 1,86
Заказ № 937/1021-УМП05 Тираж 30 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (495) 973-45-68
E-mail: zakaz@itsbook.ru

Лабораторная работа № 1

Решение на ЭВМ задач с использованием программ линейных алгоритмов

1. Цель работы - получение практических навыков в решении задач на ЭВМ общего назначения, с использованием программ линейных алгоритмов, включая редактирование программ в ЭВМ, отладку программ, выполнение расчётов и вывод результатов на периферийные устройства, исследование реакций операционной системы на ошибки в программе.

2. Краткие теоретические сведения

Четыре директивы — это то, что называется секциями. Есть две группы секций: данных и кода.

.DATA - Эта секция содержит инициализированные данные вашей программы.

.DATA? - Эта секция содержит неинициализированные данные вашей программы. Иногда вам нужно только предварительно выделить некоторое количество памяти, но вы не хотите инициализировать ее. Эта секция для этого и предназначается. Преимущество неинициализированных данных следующее: они не занимают места в исполняемом файле.

.CONST - Эта секция содержит объявления констант, используемых программой. Константы не могут быть изменены ей. Это всего лишь *константы*.

Вы не обязаны задействовать все три секции. Объявляйте только те, которые хотите использовать.

Есть только одна секция для кода: .CODE, там где содержится весь код.

Рассмотрим основные команды и директивы ассемблера.

Первая команда - пересылка данных.

Команда:	MOV приемник, источник
Назначение:	Пересылка данных
Процессор:	все

Таблица 1 – Пересылка данных

Базовая команда пересылки данных. Копирует содержимое источника в приемник, источник не изменяется. Команда MOV действует аналогично операторам присваивания из языков высокого уровня, то есть команда

```
mov     ax,bx
```

эквивалентна выражению

```
ax := bx;
```

языка Паскаль или

ax = bx;

языка C, за исключением того, что команда ассемблера позволяет работать не только с переменными в памяти, но и со всеми регистрами процессора (более низкий уровень).

Следующая команда XCHG.

Команда:	XCHG операнд1, операнд2
Назначение:	Обмен операндов между собой
Процессор:	все

Таблица 2 – XCHG

Содержимое операнда 2 копируется в операнд 1, а старое содержимое операнда 1 — в операнд 2. XCHG можно выполнять над двумя регистрами или над регистром и переменной.

Далее представлена следующая директива INCLUDE

INCLUDE имя_файла — директива, вставляющая в текст программы текст файла аналогично команде препроцессора C #include.

INCLUDELIB имя_файла — директива, указывающая компоновщику имя дополнительной библиотеки или объектного файла, который потребуется при составлении данной программы.

Далее процедура (или подпрограмма) – *это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи*. Процедуры являются средством реализации модульного программирования.

Обращение к процедуре происходит по команде CALL [модификатор] имя_процедуры

Windows предоставляет огромное количество ресурсов Windows-программам через Windows API.

Windows API — это коллекция функций, располагающихся непосредственно в операционной системе и готовых для использования программам.

Эти функции находятся в нескольких динамически подгружаемых библиотек (DLLs), таких как kernel32.dll, user32.dll и gdi32.dll.

Kernel32.dll содержит API функции, взаимодействующие с памятью и управляющие процессами.

User32.dll контролирует пользовательский интерфейс.

Gdi32.dll ответственен за графические операции.

Кроме этих трех "основных", существуют также другие dll, которые вы можете использовать, при условии, что обладаете достаточным количеством

информации о нужных API функциях. Windows программы динамически подсоединяется к этим библиотекам, то есть код API функций не включается в исполняемый файл. Информация находится в библиотеках импорта.

Если описание функции располагается в другом программном файле, то объявление этой функции должно быть обязательно записано перед функцией. Для того чтобы функция заработала необходимо в другой функции записать вызов (обращение) к функции.

Когда программа выходит в Windows, ей следует вызвать API функцию ExitProcess.

При выполнении лабораторной работы необходимо использовать сопроцессор - FPU (Floating Point Unit) используется для ускорения и упрощения вычислений с плавающей точкой, использования математических функций.

Рис. 1 – Fpu функции

Сопроцессор (другое название FPU) ориентирован на математические вычисления - в нем отсутствуют операции с битами, зато расширен набор математических функций: тригонометрические, логарифм и т.д.

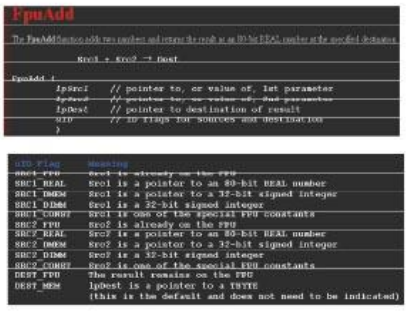

Рис 2. – Функция FpuAdd

Далее представлен пример вычисления математического выражения:
Составить программу для расчёта выражения: (a – d)/b + d при a=12;

b=17; d=2110, результат вывести на экран.

```
   .486
   .model flat, stdcall
   option casemap :none   ; case sensitive
   include \masm32\include\windows.inc
   include \masm32\include\user32.inc
   include \masm32\include\kernel32.inc
   include \masm32\macros\macros.asm
   include \masm32\include\Fpu.inc
   includelib \masm32\lib\user32.lib
   includelib \masm32\lib\kernel32.lib
   includelib \masm32\lib\Fpu.lib
   .data
a db "12",0    ; текстовая запись чисел
b db "17",0
d db "2110",0
a_real dt 0   ; не обязательно - храним преобразованные в вещественные
b_real dt 0   ; можно было обойтись, но так проще и нагляднее
d_real dt 0
result db 32 dup(0)    ; место для результата
res_title  db "Результат",0  ; заголовок окна
   .code
   start:
   invoke FpuAtoFL, ADDR a, ADDR a_real, DEST_MEM      ; преобразовываем
строку в вещественное
   invoke FpuAtoFL, ADDR b, ADDR b_real, DEST_MEM
   invoke FpuAtoFL, ADDR d, ADDR d_real, DEST_MEM
   ; вычитаем из преобразованного преобразованные и запоминаем в числовом
процессоре
; см. Help -> FPU Lib Help -> Available Functions
; источник1 = вещественное, источник2 = вещественное, результат в
сопроцессоре
; 3 параметр поэтому игнорируется
   invoke FpuSub, ADDR a_real, ADDR d_real, 0, SRC1_REAL or SRC2_REAL or
DEST_FPU
; Эти аналогчно, но результат предыдущего в сопроцессоре
   invoke FpuDiv, 0, ADDR b_real, 0, SRC1_FPU or SRC2_REAL or DEST_FPU
   invoke FpuAdd, 0, ADDR d_real, 0, SRC1_FPU or SRC2_REAL or DEST_FPU
; Преобразование результата в сопроцессоре в строку
```

```
; первый параметр игнорируется (в сопроцессоре источник),
; второй - сколько чисел после десятичной точки
; трей - куда сохранить
   invoke FpuFLtoA, 0, 2, ADDR result, SRC1_FPU or SRC2_DIMM
   invoke MessageBox, 0, ADDR result, ADDR res_title, MB_OK
   exit
end start
```

3. Порядок выполнения работы.

1. Получаете индивидуальное задание на выполнение лабораторной работы.
2. Нарисовать блок схему решаемой задачи.
3. Написать программу решения задачи.
4. По окончании занятия представляете работающую программу для решения задачи.

4.Контрольные вопросы

1. Структура программного модуля.
2. Что такое сегмент.
3. Что такое смещение.
4. Что такое стэк.
5. Функция ввода данных.
6. Функции вывода данных.
7. Команды пересылки данных общего назначения
8. Форматы арифметических данных.

5. Содержание отчета

1. Титульный лист.
2. Математическое описание решаемой задачи.
3. Блок-схема алгоритма решаемой задачи.
4. Таблица идентификаторов.
5. Программу решения задачи в masm32.
6. Результаты решения задачи (скриншоты).
7. Вывод

5. Литература

1. Павловская Т.А. C/C++. Программирование на языке высокого уровня. Учебник. – СПб.: Питер, 2009.

2. Подбельский В.В. Стандартный Си++. Учебное пособие. - М.: Финансы и статистика, 2008.

3. Петров В.И., Антонов А.А. Методы и средства программирования. Пособие по выполнению лабораторных работ. – М.: МГТУ ГА, 2010.

4. Матыюк С.П. Безопасность информационно-вычислительных систем воздушного транспорта. Учебное пособие. МГТУГА. 2020.

Лабораторная работа № 2

Решение на ЭВМ задач с использованием программ разветвляющихся алгоритмов

1. Цель работы - получение практических навыков в решении задач на ЭВМ общего назначения, с использованием программ разветвляющихся алгоритмов, включая редактирование программ в ЭВМ, отладку программ, выполнение расчётов и вывод результатов на периферийные устройства, исследование реакций операционной системы на ошибки в программе.

2. Краткие теоретические сведения

Для выполнения разветвляющихся алгоритмов необходимо использовать операции сравнения.

Знак операции	Содержание операции	Группа операций
<	Меньше	Операции отношения
<=	Меньше или равно	
>=	Больше или равно	
==	Равно	
!=	Не равно	
&&	Логическое И	Логические операции
||	Логическое ИЛИ	
!	Логическое НЕ	

Таблица 3 – Операторы

Рассмотрим пример выполнения задачи.

Вычислить величину

$$y=\begin{cases} a+\dfrac{b}{x}, & \text{если } x \le b, \\ b-ax, & \text{если } x > b, \end{cases}$$

где $a = 3$;
$b = 45$;
$x = 31$

Вывести y в зависимости от аргумента x.

```
.486
.model flat, stdcall
option casemap :none   ; case sensitive

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\macros\macros.asm
include \masm32\include\Fpu.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\Fpu.lib
.data
a db "3",0
b db "45",0
x db "31",0

b1 dd 45
x1 dd 46
a_real dt 0
b_real dt 0
x_real dt 0
Y db 32 dup(0)
Y_title  db "Результат",0
.code
start:
invoke FpuAtoFL, ADDR a, ADDR a_real, DEST_MEM
invoke FpuAtoFL, ADDR b, ADDR b_real, DEST_MEM
invoke FpuAtoFL, ADDR x, ADDR x_real, DEST_MEM
   mov eax,x1
  .if eax>b1   ; x>b
 invoke FpuMul, ADDR a_real, ADDR x_real, 0, SRC1_REAL or
SRC2_REAL or DEST_FPU
invoke FpuSub, ADDR b_real, 0, 0, SRC1_REAL or SRC2_FPU or
DEST_FPU
invoke FpuFLtoA, 0, 2, ADDR Y, SRC1_FPU or SRC2_DIMM
invoke MessageBox, 0, ADDR Y, ADDR Y_title, MB_OK
  .else
invoke FpuDiv, ADDR b_real, ADDR x_real, 0, SRC1_REAL or
SRC2_REAL or DEST_FPU
invoke FpuAdd, 0, ADDR a_real, 0, SRC1_FPU or SRC2_REAL or
DEST_FPU
invoke FpuFLtoA, 0, 2, ADDR Y, SRC1_FPU or SRC2_DIMM
```

```
invoke MessageBox, 0, ADDR Y, ADDR Y_title, MB_OK
.endif
exit
end start
```

3. Порядок выполнения работы.

1. Получаете индивидуальное задание на выполнение лабораторной работы.
2. Нарисовать блок схему решаемой задачи.
3. Написать программу решения задачи.
4. По окончании занятия представляете работающую программу для решения задачи

4. Контрольные вопросы

1. Понятие разветвляющегося алгоритма.
2. Типы переменных в языке ассемблера, примеры.
3. Оператор присваивания, порядок его выполнения.
4. Приоритеты выполнения операций.
5. Перечислить библиотеки, подключаемые к программному модулю.
6. Оператор безусловного перехода, синтаксис, порядок выполнения.
7. Оператор switch , синтаксис, порядок выполнения.
8. Оператор блочный if , синтаксис, порядок выполнения.
9. Оператор else –if, синтаксис, порядок выполнения.

5. Содержание отчета

1. Титульный лист.
2. Математическое описание решаемой задачи.
3. Блок-схема алгоритма решаемой задачи.
4. Таблица идентификаторов.
5. Программу решения задачи в masm32.
6. Результаты решения задачи (скриншоты).
7. Вывод

6. Литература

1. Павловская Т.А. С/С++. Программирование на языке высокого уровня. Учебник. – СПб.: Питер, 2009.

2. Подбельский В.В. Стандартный Си++. Учебное пособие. - М.: Финансы и статистика, 2008.

3. Петров В.И., Антонов А.А. Методы и средства программирования. Пособие по выполнению лабораторных работ. – М.: МГТУ ГА, 2010.

4. Матыюк С.П. Безопасность информационно-вычислительных систем воздушного транспорта. Учебное пособие. МГТУГА. 2020.

Лабораторная работа № 3

Решение на ЭВМ задач с использованием программ циклических алгоритмов

1. Цель работы - получение практических навыков в решении задач на ЭВМ общего назначения, с использованием программ циклических алгоритмов, включая редактирование программ в ЭВМ, отладку программ, выполнение расчётов и вывод результатов на периферийные устройства, исследование реакций операционной системы на ошибки в программе.

2. Краткие теоретические сведения

Для организации циклических алгоритмов в ассемблере используют команду безусловного перехода -jmp

Команда	Перевод (с англ.)	Назначение	Процессор
JMP метка	jump - прыжок	Безусловный переход	все

Таблица 4 – Команда безусловного перехода

Безусловный переход в программе на Ассемблере производится по команде JMP. Полный формат команды следующий:

JMP [модификатор] *адрес_перехода*.

Адрес перехода может быть либо меткой, либо адресом области памяти, в которую предварительно помещен указатель перехода.

Фрагмент кода программы представлен ниже. Команда jmp просто переходит на указанную метку в программе, пример:

```
(1) mov ah,9
(2) mov dx,offset Str
(3) int 21h
(4) jmp Label_2
(5)
(6) add cx,12
(7) dec cx
(8) Label_2:
(9) int 20h
```

В результате строки (5) - (7) работать не будут. Программа выведет сообщение на экран, а затем jmp перейдет к строке (8), после чего программа завершится

Кроме того, активно используется оператор LOOP для организации циклов.

Команда	Перевод (с англ.)	Назначение	Процессор
LOOP метка	loop - петля	Организация циклов	все

Таблица 5 – Таблица организации циклов

Количество повторов задается в регистре ECX (счетчик).

Вот как можно использовать этот оператор на практике:

(1) mov cx,3
(2) Label_1:
(3) mov ah,9
(4) mov dx,offset Str
(5) int 21h
(6) loop Label_1
(7) ...

В строке (1) загружаем в CX количество повторов (отсчет будет идти от 3 до 0).

В строке (2) создаем метку (Label - метка).

Далее (строки (3)-(5)) выводим сообщение.

И в строке (6) оператор loop уменьшает на единицу CX и, если он не равен нулю, переходит на метку Label_1 (строка (2)).

Итого строка будет выведена на экран четыре раза. Когда программа перейдет на строку (7), регистр CX будет равен нулю. В результате код программы уменьшается.

Кроме того можно использовать условное ассемблирование. Пример приведен ниже:

```
.WHILE условие
  ...
.ENDW

WHILE EAX<64H
ADD EAX,10
ENDW
```

При условном ассемблировании необходимо помнить о знаках операций и их содержимом.

Знак операции	Содержание операции	Группа операций
<	Меньше	Операции отношения
<=	Меньше или равно	
>=	Больше или равно	
==	Равно	
!=	Не равно	
&&	Логическое И	Логические операции
||	Логическое ИЛИ	
!	Логическое НЕ	

Таблица 6 – Операторы

Пример

Задан массив T=(10, 20, 30, 40, 50, 60, 70, 80); и начальные значения $x_0=0$; $y_0=0$. С помощью операторов цикла ассемблера вычислить $x_i= T_i*T_i$, $y_{i+1}=x_i+y_i$ и вывести значения y_{i+1} на экран для всех значений массива T.

```
.486
.model flat, stdcall
option casemap :none   ; case sensitive
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\macros\macros.asm
include \masm32\include\Fpu.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\Fpu.lib
.data
T_real   dt 10., 20., 30., 40., 50., 60., 70., 80.; точка после числа обязательна
x_real   dt 0
y_real   dt 0
result    db  32 dup(0)    ; место для результата
res_title  db  "Результат",0  ; заголовок окна

.code
start:
mov       ecx,7          ;значение счетчика цикла в cx
mov       esi,0          ;индекс начального элемента в cx
go:                      ;цикл инициализации
push ecx
invoke FpuXexpY, ADDR T_real[esi], 2, ADDR x_real, SRC1_REAL or
SRC2_DIMM or DEST_MEM
invoke FpuAdd, ADDR x_real, ADDR y_real, ADDR y_real, SRC1_REAL or
SRC2_REAL or DEST_MEM
```

```
invoke FpuFLtoA, ADDR y_real, 2, ADDR result, SRC1_REAL or SRC2_DIMM
invoke MessageBox, 0, ADDR result, ADDR res_title, 0
add esi,10 ;продвижение к следующему элементу массива
pop ecx
loop go ; уменьшает ecx на 1
exit
```

3. Порядок выполнения работы.

1. Получаете индивидуальное задание на выполнение лабораторной работы.
2. Нарисовать блок схему решаемой задачи.
3. Написать программу решения задачи.
4. По окончании занятия представляете работающую программу для решения задачи

4. Перечень вопросов для защиты лабораторной работы.

1. Понятие циклического алгоритма.
2. Организация цикла с предусловием while, схема цикла, синтаксис оператора, порядок выполнения оператора.
3. Организация цикла с параметром, схема цикла, синтаксис оператора, порядок выполнения оператора.
4. Оператор break в циклах.
5. Оператор jmp.

5. Содержание отчета

1. Титульный лист.
2. Математическое описание решаемой задачи.
3. Блок-схему алгоритма решаемой задачи.
4. Таблицу идентификаторов.
5. Программу решения задачи в masm32.
6. Результаты решения задачи (скриншоты).
7. Вывод

6. Литература

1. Павловская Т.А. C/C++. Программирование на языке высокого уровня. Учебник. – СПб.: Питер, 2009.

2. Подбельский В.В. Стандартный Си++. Учебное пособие. - М.: Финансы и статистика, 2008.

3. Петров В.И., Антонов А.А. Методы и средства программирования. Пособие по выполнению лабораторных работ. – М.: МГТУ ГА, 2010.

4. Матыюк С.П. Безопасность информационно-вычислительных систем воздушного транспорта. Учебное пособие. МГТУГА. 2020.

Лабораторная работа № 4

Решение на ЭВМ задач с использованием программ сложных циклических алгоритмов.

1. Цель работы - получение практических навыков в решении задач на ЭВМ общего назначения, с использованием программ сложных циклических алгоритмов, включая редактирование программ в ЭВМ, отладку программ, выполнение расчётов и вывод результатов на периферийные устройства, исследование реакций операционной системы на ошибки в программе.

2. Краткие теоретические сведения

Большинство команд ассемблера оперируют байтом, словом или двойным словом. Однако во многих случаях бывает необходимо переслать или сравнить поля данных, которые превышают по размеру слово или байт.

Например, может потребоваться сравнивать описания или имена, чтобы отсортировать
их в определенной последовательности. Элементы такого формата известны как строковые данные и могут иметь как символьный, так и числовой тип. Преимущества ассемблера проявляются и при обработке строк и массивов данных.

Под операциями обработки строк мы будем понимать следующие операции:

- ***сравнение двух строк;***
- ***копирование строки-источника в строку-приемник;***
- ***считывание строк из устройства или файла;***
- ***запись строки в устройство или файл;***
- ***определение размера строки;***
- ***нахождение подстроки в заданной строке;***
- ***объединение двух строк (конкатенация).***

Операции над строками широко используются в языках высокого уровня. Ассемблерная реализация таких операций позволяет существенно повысить быстродействие программ, особенно если требуется обработать большое количество строк и массивов.

Строка символов или чисел, с которыми программа работает как с группой, является обычным типом данных.

Программа пересылает строку из одного места в другое, сравнивает ее с другими строками, ищет в ней заданное значение.

При работе со строками программист сталкивается с необходимостью определить окончание строки, чтобы точно знать, когда заканчивать обработку. Существует два принципиально разных подхода к идентификации строки и ее элементов.

Можно указать размер строки (количество элементов, входящих в строку), записав число элементов в первый байт строки. По общепринятым соглашениям первый элемент строки имеет смещение 0, поэтому можно сказать, что размер строки прописывается в нулевом элементе, а символы строки начинаются с первого элемента. Такой принцип был реализован в языке **Pascal** и в среде программирования **Delphi.** Такие строки называются короткими **(short strings)**, поскольку их размер не превышает 255 байт.

Наибольшее распространение получил второй способ идентификации строки, при котором в конце строки указывается нулевой символ. Такие строки называются строками с завершающим нулем (null-terminated strings).

Например, они используются в языке Си++ в операционных системах Windows.

StringJ DB "NULL-TERMINATED STRING",0

Ничто не мешает использовать и другой вариант в ассемблере, при этом в процессе обработки придется отслеживать конец строки.

```
.data
si DB "TEST STRING",0
.code
lea ESI, si ; адрес первого элемента строки
cmp byte ptr [ESI],0 ; проверка на конец строки
```

Для байтовых массивов действительны те же приемы работы, что и для символьных строк, а вот при работе с элементами размером в слово или двойное слово следует учитывать некоторые особенности, связанные с размерностью элементов.

Команда:	LEA приемник, источник
Назначение:	Вычисление эффективного адреса

Таблица 7 – LEA приемник

Для обработки строковых данных ассемблер имеет пять команд обработки строк:

MOVS - переслать один байт или одно слово из одной области памяти в другую;

LODS - загрузить из памяти один байт в регистр AL или одно слово в регистр AX;
STOS - записать содержимое регистра AL или AX в память;
CMPS - сравнить содержимое двух областей памяти, размером в один байт или в одно слово;
SCAS - сравнить содержимое регистра AL или AX с содержимым памяти.

Следующая команда – цепочечная, может быть использована для многократной обработки одного байта, одного слова или двойного слова. Для этого указывается префикс повторения rep.

В таблице приведены модификации префикса rep для команд строковых примитивов.

• rep — повторять операцию, пока CX не станет равным 0;

• repz, repe — повторять операцию, пока элементы равны, то есть до первого неравенства (флаг ZF установлен в 0). Операция прекращается, если флаг ZF устанавливается в 1 или счетчик в регистре ECX (CX) достигает нуля;

• repne, repnz — повторять операцию, пока элементы не равны, то есть до первого равенства (флаг ZF установлен в 1). Операция прекращается при установке флага ZF в 0 или при достижении значения 0 в регистре ECX (CX).

Несмотря на то, что цепочечные команды имеют отношение к одному байту или одному слову, префикс REP обеспечивает повторение команды несколько раз. Префикс кодируется непосредственно перед цепочечной командой, например, REP MOVSB.

Для использования префикса REP необходимо установить начальное значение в регистре CX. При выполнении цепочечной команды с префиксом REP происходит уменьшение на 1 значения в регистре CX до нуля. Таким образом, можно обрабатывать строки любой длины.

В строковых командах не применяются способы адресации, характерные для остальных команд обработки строк. Строковые команды адресуют операнды комбинациями регистров ESI(SI) или EDI (DI).

Операнды источника используют регистр ESI (SI), а операнды приемника (результата) — регистр EDI (DI).

При использовании команды movsb, movsw или movsd компилятор ассемблера предполагает наличие корректного размера строковых данных и не требует кодирования операндов в команде.

Операцию копирования строк здесь выполняет команда rep movsb, для адресации строк используются 32-разрядные регистры ESI и EDI, поскольку в линейной модели адресации (директива .model flat) сегментные регистры не применяются. Красным выделена сама конструкция.

Операции копирования можно выполнять также и для массивов целых или вещественных чисел.

Команда movs может использоваться для еще одной весьма полезной операции над двумя строками, называемой сцеплением или конкатенацией. При ее выполнении к строке-приемнику добавляются символы строки-источника.

Следующая команда ЗАГРУЗКА СТРОКИ.

Команда LODS загружает из памяти в регистр AL один байт или в регистр AX одно слово. Адрес памяти определяется регистрами DS:SI. В зависимости от значения флага DF происходит увеличение или уменьшение регистра SI.

Поскольку одна команда LODS загружает регистр, то практической пользы от префикса REP в данном случае нет.

Часто простая команда MOV полностью адекватна команде LODS, хотя MOV генерирует три байта машинного кода, а LODS - только один, но требует инициализацию регистра SI.

Можно использовать команду LODS в том случае, когда требуется продвигаться вдоль строки (по байту или по слову), проверяя загружаемый регистр на конкретное значение.

Следующая команда ЗАПИСЬ СТРОКИ.

Команда STOS записывает (сохраняет) содержимое регистра AL или AX в байте или в слове памяти. Адрес памяти всегда представляется регистрами ES:DI. В зависимости от флага DF команда STOS также увеличивает или уменьшает адрес в регистре DI на 1 для байта или на 2 для слова.

Практическая польза команды STOS с префиксом REP - инициализация области данных конкретным значением, например, очистка дисплейного буфера пробелами. Длина области (в байтах или в словах) загружается в регистр AX.

Следующая команда CMPS: СРАВНЕНИЕ СТРОК

Команда CMPS сравнивает содержимое одной области памяти (адресуемой регистрами DS:SI) с содержимыми другой области (адресуемой как ES:DI). В зависимости от флага DF команда CMPS также увеличивает или уменьшает адреса в регистрах SI и DI на 1 для байта или на 2 для слова.

Команда CMPS устанавливает флаги AF, CF, OF, PF, SF и ZF. При использовании префикса REP в регистре CX должна находиться длина сравниваемых полей. Команда CMPS может сравнивать любое число байт или слов.

Следующая команда SCAS: СКАНИРОВАНИЕ СТРОК.

Команда SCAS отличается от команды CMPS тем, что сканирует (просматривает) строку на определенное значение байта или слова. Команда SCAS сравнивает содержимое области памяти (адресуемой регистрами ES:DI) с содержимым регистра AL или AX.

В зависимости от значения флага DF команда SCAS также увеличивает или уменьшает адрес в регистре DI на 1 для байта или на 2 для слова. Команда SCAS устанавливает флаги AF, CF, OF, PF, SF и ZF. При использовании префикса REP и значения длины в регистре CX команда SCAS может сканировать строки любой длины.

Команда SCAS особенно полезна, например, в текстовых редакторах, где программа должна сканировать строки, выполняя поиск знаков пунктуации: точек, запятых и пробелов.

ОСНОВНЫЕ ПОЛОЖЕНИЯ НА ПАМЯТЬ

для цепочечных команд MOVS, STOS, CMPS и SCAS не забывайте инициализировать регистр ES;

сбрасывайте (CLD) или устанавливайте (STD) флаг направления в соответствии с направлением обработки;

не забывайте устанавливать в регистрах DI и SI необходимые значения. Например, команда MOVS предполагает операнды DI,SI, а команда CMPS - SI,DI;

инициализируйте регистр CX в соответствии с количеством байтов или слов, участвующих в процессе обработки;

для обычной обработки используйте префикс REP для команд MOVS и STOS и модифицированный префикс (REPE или REPNE) для команд CMPS и SCAS;

помните об обратной последовательности байтов в сравниваемых словах при выполнении команд CMPSW и SCASW.

при обработке справа налево устанавливайте начальные адреса на последний байт обрабатываемой области. Если, например, поле NAME1 имеет длину 10 байтов, то для побайтовой обработки данных в этой области справа налево начальный адрес, загружаемый командой LEA, должен быть NAME1+9. Для обработки слов начальный адрес в этом случае - NAME1+8.

Теперь еще раз акцентируем внимание на основных элементах лекции.

Большинство команд ассемблера оперируют байтом, словом или двойным словом. Однако во многих случаях бывает необходимо переслать или сравнить поля данных, которые превышают по размеру слово или байт.

Например, может потребоваться сравнивать описания или имена, чтобы отсортировать
их в определенной последовательности. Элементы такого формата известны как строковые данные и могут иметь как символьный, так и числовой тип. Преимущества ассемблера проявляются и при обработке строк
и массивов данных.

Под операциями обработки строк мы будем понимать следующие операции:

• ***сравнение двух строк;***

- ***копирование строки-источника в строку-приемник;***
- ***считывание строк из устройства или файла;***
- ***запись строки в устройство или файл;***
- ***определение размера строки;***
- ***нахождение подстроки в заданной строке;***
- ***объединение двух строк (конкатенация).***

Операции над строками широко используются в языках высокого уровня. Ассемблерная реализация таких операций позволяет существенно повысить быстродействие программ, особенно если требуется обработать большое количество строк и массивов.

Строка символов или чисел, с которыми программа работает как с группой, является обычным типом данных.

Программа пересылает строку из одного места в другое, сравнивает ее с другими строками, ищет в ней заданное значение.

При работе со строками программист сталкивается с необходимостью определить окончание строки, чтобы точно знать, когда заканчивать обработку. Существует два принципиально разных подхода к идентификации строки и ее элементов.

Можно указать размер строки (количество элементов, входящих в строку), записав число элементов в первый байт строки. По общепринятым соглашениям первый элемент строки имеет смещение 0, поэтому можно сказать, что размер строки прописывается в нулевом элементе, а символы строки начинаются с первого элемента. Такой принцип был реализован в языке **Pascal** и в среде программирования **Delphi.** Такие строки называются короткими **(short strings)**, поскольку их размер не превышает 255 байт.

Наибольшее распространение получил второй способ идентификации строки, при котором в конце строки указывается нулевой символ. Такие строки называются строками с завершающим нулем (null-terminated strings).

ОСНОВНЫЕ ПОЛОЖЕНИЯ НА ПАМЯТЬ

для цепочечных команд MOVS, STOS, CMPS и SCAS не забывайте инициализировать регистр ES;

сбрасывайте (CLD) или устанавливайте (STD) флаг направления в соответствии с направлением обработки;

не забывайте устанавливать в регистрах DI и SI необходимые значения. Например, команда MOVS предполагает операнды DI,SI, а команда CMPS - SI,DI;

инициализируйте регистр CX в соответствии с количеством байтов или слов, участвующих в процессе обработки;

для обычной обработки используйте префикс REP для команд MOVS и STOS и модифицированный префикс (REPE или REPNE) для команд CMPS и SCAS;

помните об обратной последовательности байтов в сравниваемых словах при выполнении команд CMPSW и SCASW.

при обработке справа налево устанавливайте начальные адреса на последний байт обрабатываемой области. Если, например, поле NAME1 имеет длину 10 байтов, то для побайтовой обработки данных в этой области справа налево начальный адрес, загружаемый командой LEA, должен быть NAME1+9. Для обработки слов начальный адрес в этом случае - NAME1+8.

С представлением *одномерных* массивов в программе на ассемблере и организацией их обработки все достаточно просто. А как быть если программа должна обрабатывать *двухмерный* массив? Все проблемы возникают по-прежнему из-за того, что специальных средств для описания такого типа данных в ассемблере нет. *Двухмерный* массив нужно моделировать. На описании самих данных это почти никак не отражается — память под массив выделяется с помощью директив резервирования и инициализации памяти.

Непосредственно моделирование обработки массива производится в сегменте кода, где программист, описывая алгоритм обработки ассемблеру, определяет, что некоторую область памяти необходимо трактовать как двухмерный массив. При этом вы вольны в выборе того, как понимать расположение элементов двухмерного массива в памяти: по строкам или по столбцам.

Таким образом, последовательность однотипных элементов в памяти трактуется как двухмерный массив, расположенный по строкам, то адрес элемента **(i, j)** вычисляется по формуле:

(база + количество_элементов_в_строке * размер_элемента * i+j)

Здесь **i = 0...n–1** указывает *номер строки*, а **j = 0...m–1** указывает *номер столбца.*

3. Порядок выполнения работы

1. Получаете индивидуальное задание на выполнение лабораторной работы.
2. Нарисовать блок схему решаемой задачи.
3. Написать программу решения задачи.
4. По окончании занятия представляете работающую программу для решения задачи

4. Перечень вопросов для защиты лабораторной работы

1. Понятие циклического алгоритма.
2. Организация цикла с предусловием while.
3. Организация цикла с пост условием do-while.
4. Организация цикла с параметром.
5. Оператор break в циклах.
6. Оператор continue в циклах.

5. Содержание отчета

1. Титульный лист.
2. Математическое описание решаемой задачи.
3. Блок-схему алгоритма решаемой задачи.
4. Таблицу идентификаторов.
5. Программу решения задачи в masm32.
6. Результаты решения задачи (скриншоты).
7. Вывод

6. Литература

1. Павловская Т.А. C/C++. Программирование на языке высокого уровня. Учебник. – СПб.: Питер, 2009.

2. Подбельский В.В. Стандартный Си++. Учебное пособие. - М.: Финансы и статистика, 2008.

3. Петров В.И., Антонов А.А. Методы и средства программирования. Пособие по выполнению лабораторных работ. – М.: МГТУ ГА, 2010.

4. Матыюк С.П. Безопасность информационно-вычислительных систем воздушного транспорта. Учебное пособие. МГТУГА. 2020.

Лабораторная работа № 5

Решение на ЭВМ задач с использованием программ дочерних элементов управления в операционной системе

1. Цель работы - получение практических навыков в решении задач на ЭВМ общего назначения, с использованием дочерних элементов управления в операционной системе, включая редактирование программ в ЭВМ, отладку программ, выполнение расчётов и вывод результатов на периферийные устройства, исследование реакций операционной системы на ошибки в программе.

2. Краткие теоретические сведения

Операционная система Windows предоставляет несколько предопределенных классов окон, которые мы можем сразу же использовать в своих программах.

Рассмотрим эту же задачу с применением дочерних элементов управления операционной системы.

Составить программу с использованием дочерних элементов управления операционной системы для расчёта выражения: (a – d)/b + d при a=12; b=17; d=2110, результат вывести на экран, предусмотреть ввод данных с клавиатуры.

Изначально необходимо написать код программы с тремя полями ввода данных и кнопкой, при нажатии на которую происходит вычисление.

В дальнейшем ввести необходимые значения - начальные условия.

Затем при нажатии на кнопку происходит вычисление и вывод значения на экран диалоговым окном.

Начало программы стандартное, подключение различных библиотек.

.386 говорит MASM'у, что намереваемся использовать набор инструкций процессора 80386 в этой программе.

Вторая строка .model flat, stdcall говорит MASM'у, что наша программа будет использовать плоскую модель памяти. Также мы использовать передачу параметров типа STDCALL по умолчанию.

Следом идет прототип функции WinMain. Перед тем, как мы вызовем в дальнейшем эту функцию, мы должны сначала определить ее прототип.

Мы должны подключить windows.inc в начале кода. Он содержит важные структуры и константы, которые потребуются нашей программе. Этот файл всего лишь текстовый файл.

Наша программа вызывает API функции, находящиеся в user32.dll (CreateWindowEx, RegisterWindowClassEx, например) и kernel32.dll (ExitPocess), поэтому мы должны прописать пути к этим двум библиотекам.

В секции .DATA, мы объявляем оканчивающиеся NULL'ом строки.

ClassName - имя нашего класса окна и AppName - имя нашего окна. Обе переменные проинициализированны.

Далее идут имена класса относящего к полю ввода и кнопке.

Затем идут надпись на кнопке и текстовые поля трёх параметров, а также название окна результата.

Резервируем место для результата вычислений.

Далее определяем три наши вводимые параметра .

В секции .DATA? объявлены переменные: hInstance (хэндл нашей программы) и CommandLine (командная строка нашей программы).

Затем объявляются хэндлы трёх полей ввода и кнопки, а определяется буфер для хранения промежуточных переменных.

В дальнейшем объявляется секция констант. Значение equ, означает равнозначно 2, 3 или 1. Зависит соответственно какая цифра после выражения.

Обратите внимание, что все переменные в этой секции не инициализированны, так как они не должны содержать какое-то определенное значение при загрузке программа, но мы хотим зарезервировать место на будущее.

Следующая секция .CODE содержит все инструкции.

Наш код располагается между <стартовая start>: и end < start >. Имя метки несущественно. Вы можете назвать ее как пожелаете, до тех пор, пока оно уникально и не нарушает правила именования в MASM'е.

Первая инструкция - вызов GetModuleHandle, чтобы получить хэндл нашей программы. Еще раз напомпинаю, что хэндл программы - это ее линейный адрес в памяти. По возвращению из Win32 функции, возвращаемое ею значение находится в eax. Все другие значения возвращаются через переменные, переданные в параметрах функции.

Затем мы запоминаем наш линейный адрес нашей программы в заранее инициализированную нами переменную - hInstance.

Вызов GetCommandLine не нужен, если ваша программа не обрабатывает командные строки, но для общего представления необходим. Также запоминаем дескриптор в заранее инициализированную нами переменную - CommandLine.

Далее идет вызов WinMain. Она получает четыре параметра:

- *хэндл программы,*
- *хэндл предыдущего экземпляра программы,*
- *коммандную строку*
- *и состояние окна при первом появлении.*

Под Win32 нет такого понятия, как предыдущий экземпляр программы. Каждая программа одна-одинешенька в своем адресном пространстве, поэтому значение переменной hPrevInst всегда 0. Это пережиток времен Win16, когда все экземпляры программы запускались в одном и том же адресном пространстве, и экземпляр мог узнать, был ли запущены еще копии этой программы. Под Win16, если hPrevInst равен NULL, тогда этот экземпляр является первым.

По возвращению из WinMain, регистр EAX заполняется значением кода выхода. Мы передаем код выхода как параметр функции ExitProcess, которая завершает нашу программу.

WinMain proc

В данной строке объявление функции WinMain. Обратите внимание на параметры. Вы можете обращаться к этим параметрам, вместо того, чтобы манипулировать со стеком. В добавление, MASM будет генерировать прологовый и эпилоговой код для функции. Так что мы не должны беспокоиться о состоянии стека при входе и выходе из функции.

Директива LOCAL резервирует память из стека для локальных переменных, использованных в функции. Все директивы LOCAL должны следовать непосредственно после директивы PROC. После LOCAL сразу идет <имя_переменной>:<тип переменной>.

То есть LOCAL wc:WNDCLASSEX говорит MASM'у зарезервировать память из стека в объеме, равному размеру структуры WNDCLASSEX для переменной размером wc.

Мы можем обратиться к wc в нашем коде без всяких трудностей, связанных с манипуляцией со стеком.

Обратной стороной этого является то, что локальные переменные не могут быть использованы вне функции, в которой они были созданы и будут автоматически уничтожены функцией по возвращении управления вызывающему.

Другим недостатком является то, что вы не можете инициализировать локальные переменные автоматически, потому что они всего лишь стековая память, динамически зарезервированная, когда функция была созданна. Вы должны вручную присвоить им значения.

Некоторое отступление к классам.

Каждое окно принадлежит определенному классу окна. Прикладная программа должна зарегистрировать класс окна перед созданием любого окна этого класса. Класс окна (**window class**) определяет большинство аспектов внешнего вида и поведения. Класс окна - это спецификации будущего окна. Он определяет характеристики окна, такие как иконка, курсор, функцию, ответственную за окно и так далее. Вы создаете окно из класса окна.

Если вы определите ваше собственное окно, вы должны заполнить желаемые характеристики в структуре WNDCLASSEX или WNDCLASSEX и вызвать RegisterClass или RegisterClassEx, прежде чем в сможете создать ваше окно. Вы только должны один раз зарегистрировать класс окна для каждой их разновидности, из которых вы будете создавать окна.

Самый важный член WNDCLASSEX - это lpfnWndProc. lpfn означает дальний указатель на функцию. Каждому классу окна должен быть сопоставлена процедура окна, которая ответственна за обработку сообщения всех окон этого класса. Windows будут слать сообщения процедуре окна, чтобы уведомить его о важных событий, касающихся окон, за которые ответственна эта процедура, например о вводе с клавиатуры или перемещении мыши. Процедура окна должна выборочно реагировать на получаемые ей сообщения. Вы будете тратить большую часть вашего времени на написания обработчиков событий.

cbSize: Размерр структуры WDNCLASSEX в байтах. Мы можем использовать оператор SIZEOF, чтобы получить это значение.

style: Стиль окон, создаваемых из этого класса. Вы можете комбинировать несколько стилей вместе, используя оператор "or".

cbClsExtra: Количество дополнительных байтов, которые нужно зарезервировать (они будут следовать за самой структурой).

hInstance: Хэндл модуля.

hIcon: Хэндл иконки. Получите его функцией LoadIcon.

hCursor: Хэндл курсора. Получите его функцией LoadCursor.

hbrBackground: Цвет фона

lpszMenuName: Хэндл меню для окон, созданных из класса по умолчанию.

lpszClassName: Имя класса окна.

hIconSm: Хэндл маленькой иконки, которая сопоставляется классу окна. Если этот член равен NULL'у, система ищет иконку, определенную для члена hIcon, чтобы использовать ее как маленькую иконку.

Таким образом, в следующем фрагменте кода мы задаем параметры нашего окна.

По завершении данного фрагмента кода происходит регистрация класса окна.

После регистрации класса окна, мы должны вызвать функцию CreateWindowEx, чтобы создать наше окно, основанное на этом классе.

CreateWindowExA proto dwExStyle:DWORD,\ lpClassName:DWORD,\ lpWindowName:DWORD,\ dwStyle:DWORD,\ X:DWORD,\ Y:DWORD,\ nWidth:DWORD,\ nHeight:DWORD,\ hWndParent:DWORD ,\ hMenu:DWORD,\ hInstance:DWORD,\ lpParam:DWORD

Рассмотрим описание каждого параметра:

dwExStyle: Дополнительные стили окна. Вы можете использовать NULL, если вам не нужны дополнительные стили.

lpClassName: (Обязательный параметр). Адрес ASCIIZ строки, содержащую имя класса окна, которое вы хотите использовать как шаблон для этого окна. Это может быть ваш собственный зарегистрированный класс или один из предопределенных классов.

lpWindowName: Адрес ASCIIZ строки, содержащей имя окна. Оно будет показано на title bar'е окно. Если этот параметр будет равен NULL'у, он будет пуст.

dwStyle: Стили окна. Вы можете определить появление окна здесь. Можно передать NULL без проблем, тогда у окна не будет кнопок изменения резмеров, закрытия и системного меню.

X, Y: Координаты вернего левого угла окна. Обычно эти значения равны CW_USEDEFAULT, что позволяет Windows решить, куда поместить окно.

nWidth, nHeight: Ширина и высота окна в пикселях. Вы можете также использовать CW_USEDEFAULT, чтобы позволить Windows выбрать соответствующую ширину и высоту для вас.

hWndParent: Хэндл родительского окна (если существует). Этот параметр говорит Windows является ли это окно дочерним (подчиненным) другого окна, и, если так, кто родитель окна. Так как в нашем примере всего лишь одно окно, мы устанавливаем этот параметр в NULL.

hMenu: Хэндл меню окна. NULL - если будет использоваться меню, определенное в классе окна.

hInstance: Хэндл программного модуля, создающего окно.

lpParam: Опциональный указатель на структуру данных, передаваемых окну. Обычно этот параметр установлен в NULL, означая, что никаких данных не передается через CreateWindow().

После успешного возвращения из CreateWindowsEx, хэндл окна находится в eax. Мы должны сохранить это значение, так как будем использовать его в будущем.

Окно, которое мы только что создали, не покажется на экране автоматически. Вы должны вызвать ShowWindow, передав ему хэндл окна и желаемый тип отображения на экране, чтобы оно появилось на рабочем столе.

Затем вы должны вызвать UpdateWindow для того, чтобы окно перерисовало свою клиентскую область. Эта Функция полезна, когда вы хотите обновить содержимое клиентской области. Вы Тем не менее, вы можете пренебречь вызовом этой функции.

Теперь окно на экране. Но оно не может получать ввод из внешнего мира. Поэтому мы должны проинформировать его о соответствующих событиях. Мы достигаем этого с помощью цикла сообщений. В каждом модуле есть только один цикл сообщений.

В нем функцией GetMessage последовательно проверяется, есть ли сообщения от Windows.

GetMessage передает указатель на на MSG структуру Windows.

GetMessage возвращает FALSE, если было получено сообщение WM_QUIT, что прерывает цикл обработки сообщений и происходит выход из программы.

TranslateMessage - это вспомогательная функция, которая обрабатывает ввод с клавиатуры и генерирует новое сообщение (WM_CHAR), помещающееся в очередь сообщений. Вы можете не использовать эту функцию, если ваша программа не обрабатывает ввод с клавиатуры.

DispatchMessage пересылает сообщение процедуре соответствующего окна. Если цикл обработки сообщений прерывается, код выхода сохраняется в члене MSG структуры wParam. Вы можете сохранить этот код выхода в eax, чтобы возвратить его Windows.

Далее идет процедура окна.

Первый параметр, hWnd, это хэндл окна, которому предназначается сообщение.

uMsg - сообщение. Отметьте, что uMsg - это не MSG структура. Это всего лишь число.

wParam и lParam всего лишь дополнительные параметры, исспользующиеся некоторыми сообщениями.

Ваш код должен проверить сообщение, чтобы убедиться, что это именно то, которое вам нужно. Если это так, сделайте все, что вы хотите сделать в качестве реакции на это сообщение, а затем возвратитесь, оставив в eax ноль.

Единственное сообщение, которое вы ОБЯЗАНЫ обработать - это WM_DESTROY. Это сообщение посылается вашему окну, когда оно закрывается.

Если же это не то сообщение, которое вас интересует, вы должны вызвать DefWindowProc, передав ей все параметры, которые вы до этого получили. Это API функция, обрабатывающая сообщения, которыми ваша программа не интересуется.

Относительно WM_DESTROY - после выполнения необходимых вам действий, вы должны вызвать PostQuitMessage, который пошлет сообщение WM_QUIT, что вынудит GetMessage вернуть нулевое значение в eax, что в свою очередь, повлечет выход из цикла обработки сообщений, а значит из программы.

Вы можете послать сообщение WM-DESTROY вашей собственной процедуре окна, вызвав функцию DestroyWindow.

На этом можно считать, что пустое окно создано, далее сформируем на нем надписи, поля ввода и кнопку.

Опять директивой LOCAL резервируем память из стека для локальных переменных, использованных в функции, только функция теперь WndProc.

Когда программе нужно отрисовать что-нибудь, она должна получить хэндл контекста устройства.

- *Вызовите BeginPaint в ответ на сообщение WM_PAINT.*
- *Вызовите GetDC в ответ на другие сообщения.*
- *Вызовите CreateDC, чтобы создать ваш собственный контекст устройства.*

Windows посылает сообщение WM_PAINT окну, чтобы уведомить его о том, что настало время для перерисовки клиентской области.

Вы должны выполнить следующие шаги, обрабатывая сообщение WM_PAINT:

- Получить хэндл контекста устройства с помощью BeginPaint.
- Отрисовать клиентскую область.

- Освободить хэндл функцией EndPaint.

Это несколько переменных, использующихся в нашей секции WM_PAINT.

Переменная **hdc** используется для сохранения хэндла контекста устройства, возвращенного функцией BeginPaint.

ps - это структура PAINTSTRUCT. Обычно вам не нужны значения этой структуры. Она передается функции BeginPaint и Windows заполняет ее подходящими значениями. Затем вы передаете ps функции EndPaint, когда заканчиваете отрисовку клиентской области.

rect - это структура RECT, определенная следующим образом:

Left и top - это координаты вернего левого угла прямоугольника.

Right и bottom - это координаты нижнего правого угла. Начала координатных осей находятся в левом верхнем углу клиентской области, поэтому точка y=10 НИЖЕ, чем точка y=0.

В ответ на сообщение WM_PAINT, вы вызываете BeginPaint, передавая ей хэндл окна, в котором вы хотите рисовать и неинициализированную структуру типа PAINTSTRUCT в качестве параметров.

После успешного вызова, eax содержит хэндл контекста устройства.

После вы вызываете GetClientRect, чтобы получить размеры клиентской области. Размеры возвращаются в переменной rect, которую вы передаете функции DrawText как один из параметров. DrawText - это высокоуровневая API функция вывода текста. Она берет на себя такие вещи как перенос слов, центровка и т.п., так что вы можете сконцентрироваться на строке, которую вы хотите нарисовать.

DrawText подгоняет строку под прямоугольник. Давайте посмотрим на ее параметры:

hdc - хэндл контекста устройства

lpString - указатель на строку, которую вы хотите нарисовать в прямоугольнике. Строка должна заканчиваться NULL'ом, или же вам придется указывать ее длину в следующем параметре, nCount.

nCount - количество символов для вывода. Если строка заканчивается NULL'ом, nCount должен быть равен -1. В противоположном случае, nCount должен содержать количество символов в строке.

lpRect - указатель на прямоугольник (структура типа RECT), в котором вы хотите рисовать строку. Заметьте, что прямоугольник ограничен, то есть вы не можете нарисовать строку за его пределами.

uFormat - значение, определяющее как строка отображается в прямоугольнике. Мы используем три значение, скомбинированные оператором "or":

DT_SINGLELINE указывает, что текст будет располагаться в одну линию

DT_CENTER центрирует текст по горизонтали

DT_VCNTER центрирует тест по вертикали. Должен использоваться вместе с DT_SINGLELINE.

После того, как вы отрисовали клиентскую область, вы должны вызвать функцию EndPaint, чтобы освободить хэндл устройства контекста.

В данном случае у вас получилось окно с тремя надписями. Далее добавим поля ввода и кнопку для начала вычислений.

Примерами предопределенных классов окон являются кнопки, списки, checkbox'ы и т.д.

Если вы хотите создать поля ввода, вы должны использовать функцию CreateWindowsEx. Другие параметры, которые вы должны указать - это хэндл родительского окна и ID поля ввода. ID поля ввода должно быть уникальным. Вы используете его для того, чтобы отличать данное поле ввода от других.

Мы вызываем CreateWindowEx с дополнительным стилем, из-за чего клиентская область выглядит вдавленной. Имя каждого контрола предопределенно - "edit" для edit-контрола.

Затем мы указываем стили дочерних окон. Например, стили кнопок начинаются с "BS_", стили edit'a - с "ES_".

SetFocus вызывается для того, чтобы направить фокус ввода на edit box, чтобы пользователь мог сразу начать вводить в него текст.

В данном случае у нас получилось окно с тремя надписями и тремя полями ввода. Далее добавим кнопку на данное окно.

Если вы хотите создать кнопку, вы должны указать "button" в качестве имени класса в CreateWindowsEx. Другие параметры этой функции мы рассмотрели при определении поля ввода.

После того, как кнопка была создана, он посылает сообщение, уведомляющие родительское окно об изменении своего состояния.

Данная кнопка посылает сообщение WM_COMMAND родительскому окну со своим ID в нижнем слове WParam'a, код уведомления в верхнем слове wParam'a, а ее хэндл в lParam'e.

Обратите внимание, что меню также шлет сообщение WM_COMMAND, как и кнопка.

Как мы можем повести различие между сообщениями WM_COMMAND, исходящими от меню и контролов (кнопок, полей ввода)? Вот ответ:

Вы можете видеть, что вы должны проверить lParam. Если он равен нулю, текущее сообщение WM_COMMAND было послано меню. Вы не можете использовать wParam, чтобы различать меню и контрол, так как ID меню и ID контрола могут быть идентичными и код уведомления должен быть равен нулю.

Вы можете поместить текстовую строку в edit box с помощью вызова SetWindowText. Вы очищаете содержимое edit box'a с помощью вызова SetWindowText, передавая ей NULL. SetWindowText - это функция общего

назначения. Вы можете использовать ее, чтобы изменить заголовок окна или текст на кнопке.

Чтобы получить текст в edit box'e (поле ввода), вы можете использовать GetWindowText.

```
invoke
GetWindowText,hwndEdit1,ADDR buffer,512
```

Далее получаем текст из полей ввода и преобразовываем его в наши переменные, а потом проводим вычисление. В завершении выводим результат.

В конце программы нам необходимо написать код обработке нажатия на кнопку.

Сначала он проверяет нижнее слово wParam'a, чтобы убедиться, что ID контрола принадлежит кнопке.

Если это так, он проверяется верхнее слово wParam'a, чтобы убедиться, что был послан код уведомления BN_CLICKED, то есть кнопка была нажата.

После этого идет собственно обработка нажатия на клавиш и программа завершена.

3. Порядок выполнения работы

1. Получаете индивидуальное задание на выполнение лабораторной работы.
2. Нарисовать блок схему решаемой задачи.
3. Написать программу решения задачи.
4. По окончании занятия представляете работающую программу для решения задачи

4. Перечень вопросов для защиты лабораторной работы

1. Структуры и классы(объекты) в ассемблере
2. Создание графического объекта, окна
3. Отображение текста
4. Элементы ввода и вывода программ
5. Создание меню
6. Открытие файлов
7. Редактирование файлов
8. Memory mapped файлы

5. Содержание отчета

1. Титульный лист.
2. Математическое описание решаемой задачи.
3. Блок-схему алгоритма решаемой задачи.
4. Таблицу идентификаторов.
5. Программу решения задачи в masm32.
6. Результаты решения задачи (скриншоты).
7. Вывод

6. Литература

1. Павловская Т.А. С/С++. Программирование на языке высокого уровня. Учебник. – СПб.: Питер, 2009.

2. Подбельский В.В. Стандартный Си++. Учебное пособие. - М.: Финансы и статистика, 2008.

3. Петров В.И., Антонов А.А. Методы и средства программирования. Пособие по выполнению лабораторных работ. – М.: МГТУ ГА, 2010.