

ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ
им. проф. М. А. БОНЧ-БРУЕВИЧА»
(спбгут)

С. И. Штеренберг
А. В. Красов
В. Е. Радынская

АССЕМБЛЕР В ЗАДАЧАХ ЗАЩИТЫ ИНФОРМАЦИИ

УЧЕБНОЕ ПОСОБИЕ

СПб ГУТ)))

САНКТ-ПЕТЕРБУРГ
2019

УДК 004.4'424(078.5)
ББК 32.973-018.2я73
Ш 90

Рецензенты:

кандидат технических наук, доцент кафедры
информационной безопасности компьютерных систем СПбПУ

Д. С. Лаврова,

кандидат педагогических наук, доцент,
директор института военного образования СПбГУТ

А. А. Лубянников

*Утверждено редакционно-издательским советом СПбГУТ
в качестве учебного пособия*

Штеренберг, С. И.

Ш 90 Ассемблер в задачах защиты информации : учебное пособие /
С. И. Штеренберг, А. В. Красов, В. Е. Радынская ; СПбГУТ. – СПб.,
2019. – 82 с.

Дано представление об использовании особенностей низко-
уровневого языка программирования Ассемблер в средах обеспече-
ния защиты информации. Приведены примеры применения средств
дизассемблирования приложений, разбора уязвимостей программного
обеспечения и различной отладки.

Предназначено для подготовки бакалавров по направлению 10.03.01
«Информационная безопасность».

**УДК 004.4'424(078.5)
ББК 32.973-018.2я73**

© Штеренберг С. И., Красов А. В., Радынская В. Е., 2019
© Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский государственный университет
телекоммуникаций им. проф. М. А. Бонч-Бруевича», 2019

СОДЕРЖАНИЕ

1. О НИЗКОУРОВНЕВОМ ЯЗЫКЕ АССЕМБЛЕРА И РЕГИСТРАХ ПАМЯТИ	4
2. ПОСТРОЕНИЕ ПРОГРАММЫ НА АССЕМБЛЕРЕ	10
3. АРИФМЕТИЧЕСКИЕ КОМАНДЫ АССЕМБЛЕРА	20
4. ЛОГИЧЕСКИЕ КОМАНДЫ АССЕМБЛЕРА	26
5. ЦИКЛЫ В АССАМБЛЕРЕ	34
6. РЕВЕРС-ИНЖИНИРИНГ	37
6.1. Некоторые инструменты статического анализа	38
6.2. Некоторые инструменты динамического анализа	40
6.3. Специализированные инструменты для продвинутого анализа	41
7. СТРУКТУРА РЕ-ФОРМАТА	44
8. МЕТОДЫ ЗАЩИТЫ ПРИЛОЖЕНИЙ ОТ АНАЛИЗА И ВЗЛОМА	53
9. ПРАКТИЧЕСКИЕ РАБОТЫ ПО РЕВЕРСУ	60
<i>Практическая работа 1. Динамический анализ кода исполняемого файла на примере CrackMe</i>	60
<i>Практическая работа 2. Ручная распаковка вирусного приложения</i>	63
10. ПРОГРАММИРОВАНИЕ В СРЕДЕ GNU ASSEMBLER	68
11. КОМПОНОВКА АССЕМБЛЕРНЫХ ФУНКЦИЙ С ПРОГРАММАМИ, НАПИСАННЫМИ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ	74
12. ВЗАИМОДЕЙСТВИЕ С МИКРОКОНТРОЛЛЕРА ЧЕРЕЗ ATMEL STUDIO НА ЯЗЫКЕ АССЕМБЛЕРА	76
СПИСОК ЛИТЕРАТУРЫ	81

1. О НИЗКОУРОВНЕВОМ ЯЗЫКЕ АССЕМБЛЕРА И РЕГИСТРАХ ПАМЯТИ

Прежде всего стоит отметить, что такое язык Ассемблера. **Язык Ассемблера** – система обозначений, используемая для представления в удобочитаемой (мнемонической) форме программ, записанных в машинном коде. Он уникален для каждого семейства компьютеров и зависит от архитектуры компьютера. В данном пособии рассматривается программирование на языке Ассемблера для процессоров Intel. К числу архитектурных особенностей процессоров Intel относятся следующие принципы.

1. Принцип хранимой программы – программа и ее данные находятся в одном адресном пространстве, с точки зрения процессора нет принципиальной разницы между данными и командами.

2. Принцип микропрограммирования – каждой команде Ассемблера соответствует набор действий в блоке микропрограммного управления;

3. Линейное пространство памяти – адреса всех ячеек памяти имеют последовательную нумерацию.

Процессоры, начиная с i486 и выше, включают в себя скалярную архитектуру, позволяющую организовать конвейерную обработку команд. Пятиступенчатый конвейер имеет следующие этапы:

- 1) выборка команды из оперативной памяти;
- 2) декодирование команды;
- 3) вычисление адреса операндов;
- 4) выполнение операции арифметико-логическом устройством (далее – АЛУ);
- 5) запись результата.

Дальнейшим развитием скалярной архитектуры является суперскалярная архитектура, реализованная в процессорах Pentium.

Суперскалярная архитектура имеет следующие особенности:

- 1) раздельное кэширование кода и данных;
- 2) предсказание правильного адреса перехода (сохраняется 256 последних переходов, вероятность правильного перехода доходит до 80 %);
- 3) усовершенствованный блок вычислений с плавающей точкой.

Регистры – это специальные ячейки памяти, расположенные непосредственно в процессоре. Работа с регистрами выполняется намного быстрее, чем с ячейками оперативной памяти, поэтому регистры активно используются как в программах на языке Ассемблера, так и компиляторами языков высокого уровня (рис. 1).

Программная модель микропроцессора содержит 32 регистра, которые можно разделить на 2 группы:

- 1) 16 пользовательских регистров;

2) 16 системных регистров.

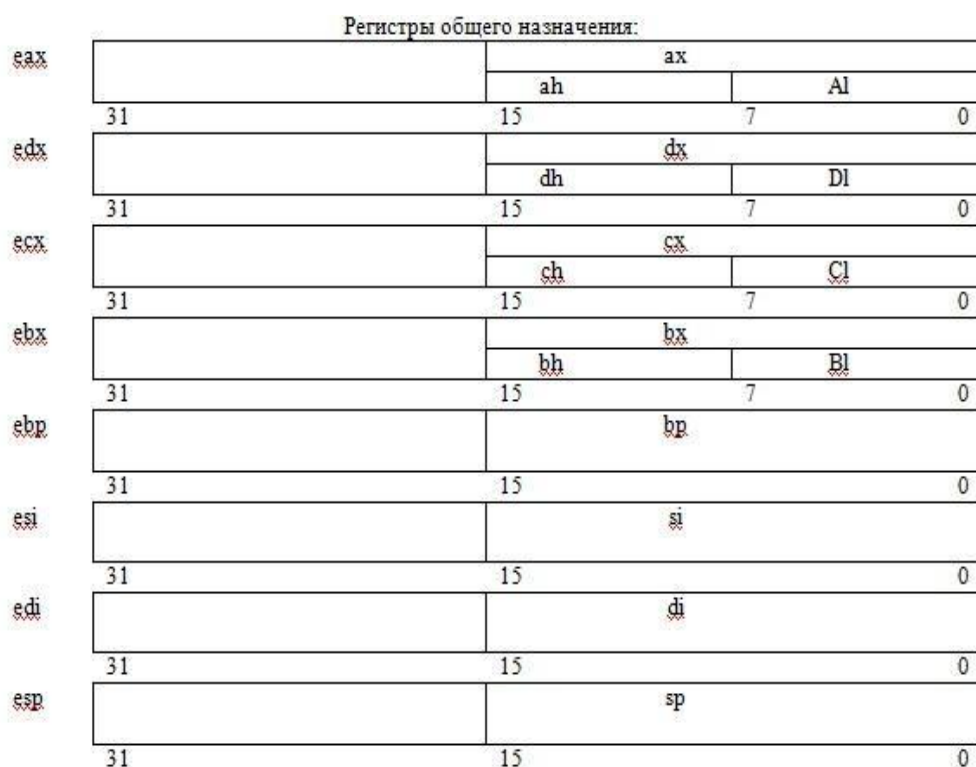


Рис. 1. Структура регистров общего назначения

Особенностями работы с регистрами является то, что для совместимости с младшими моделями процессоров программисту для самостоятельной работы предоставляются только младшие 16- и 8-битные части этих регистров. Несмотря на существующую специализацию все регистры можно использовать в любых машинных операциях. Однако надо учитывать тот факт, что некоторые команды работают только с определенными регистрами (табл. 1).

Таблица 1

Характеристика регистров общего назначения

eax/ax/ah/al	Аккумулятор – применяется для хранения промежуточных данных
ebx/bx/bh/bl	Базовый регистр – применяется для хранения базового адреса объекта в памяти
ecx/cx/ch/cl	Регистр-счетчик – применяется в командах организации циклов
edx/dx/dh/dl	Регистр данных – применяется для хранения промежуточных данных
Регистры для поддержки цепочных операций	
esi/si	Индекс источника – содержит текущий адрес элемента в цепочке-источнике
edi/di	Индекс приемника – содержит текущий адрес в цепочке-приемнике

Дополнительно в архитектуре Intel имеются сегментные регистры (рис. 2 и табл. 2) и регистры состояния (рис. 3 и табл. 3). К этим регистрам относятся: регистр флагов eflags/flags (рис. 4) и регистр указателя команды eip/ip.

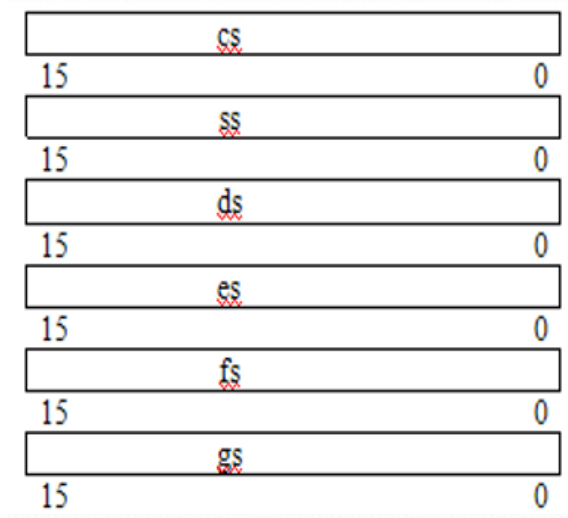


Рис. 2. Сегментные регистры

Таблица 2

Характеристика сегментных регистров

Сегмент кода	Содержит команды программ. Адрес сегмента хранится в регистре cs
Сегмент данных	Содержит данные программы. Адрес сегмента хранится в регистре ds
Сегмент стека	Содержит стек. Адрес сегмента хранится в регистре ss
Дополнительный сегмент данных	Если программе, для размещения своих данных мало одного сегмента, то можно использовать дополнительные сегменты, адреса которых записываются в es, gs, fs

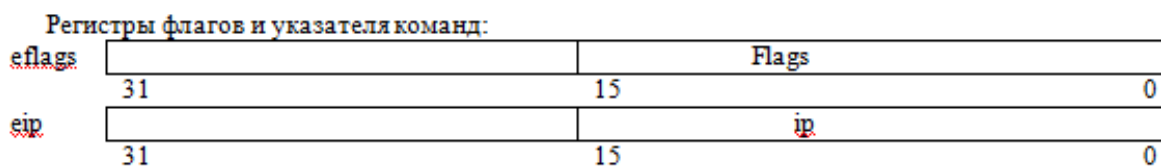


Рис. 3. Регистры состояния

Таблица 3

Характеристика регистров для работы со стеком

esp/sp	Указатель стека – содержит указатель на вершину стека
ebp/br	Указатель базы кадре стека – предназначен для организации доступа к данным из стека

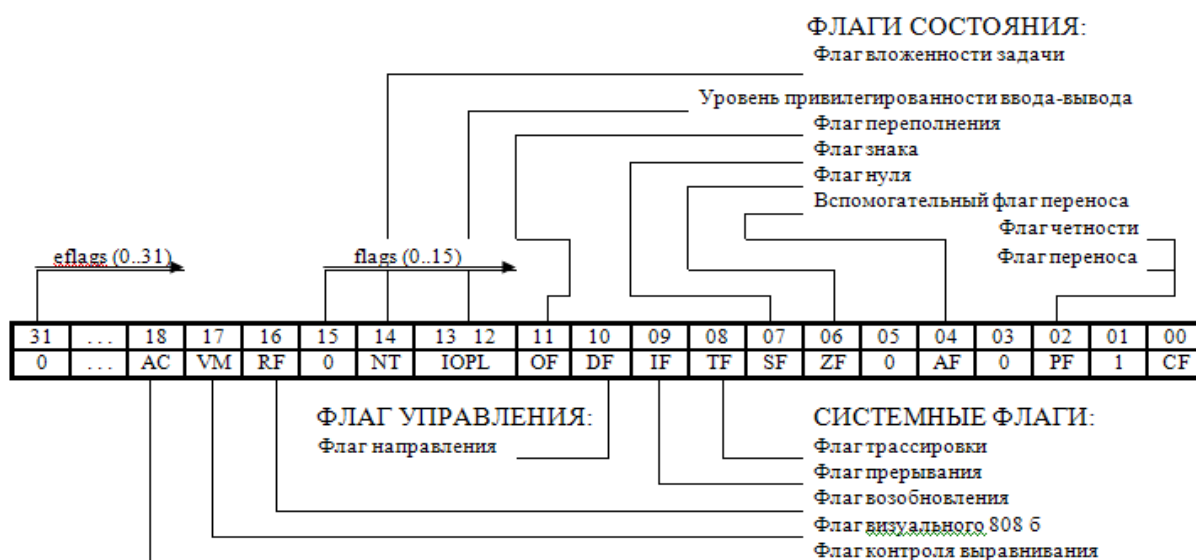


Рис. 4. Флаги состояния

Все флаги регистра флагов можно разделить на следующие три группы.

1. 8 флагов состояния. Данные флаги отражают результат исполнения арифметических или логических команд процессора.

2. 1 флаг управления. Данный флаг используется цепочными командами. Значение флага, обозначаемого как *ds*, определяет направление поэлементной обработки. Если *df* = 0 обработка производится в прямом порядке, а если *df* = 1 то в обратном. Работа с данным флагом возможна с помощью специальных команд (*cld* и *std*).

3. 5 системных флагов. Системные флаги предназначены для управления вводом/выводом, системой прерываний, режимом отладки, переключением задач. Без особой нужды модифицировать значение этих флагов нецелесообразно.

Процессор (рис. 5) поддерживает несколько режимов работы с оперативной памятью:

- 1) реальный режим – режим в котором работал процессор i8086, сохраняемый для преемственности с ранними моделями;
- 2) защищенный режим – использование всех возможностей процессора;
- 3) режим виртуального 8086 – предназначен для работы программ, созданных с использованием реального режима адресации памяти, в защищенном режиме.

Сегментация – механизм адресации, обеспечивающий существование нескольких независимых адресных пространств.

Сегмент – независимый, поддерживаемый на аппаратном уровне блок памяти.

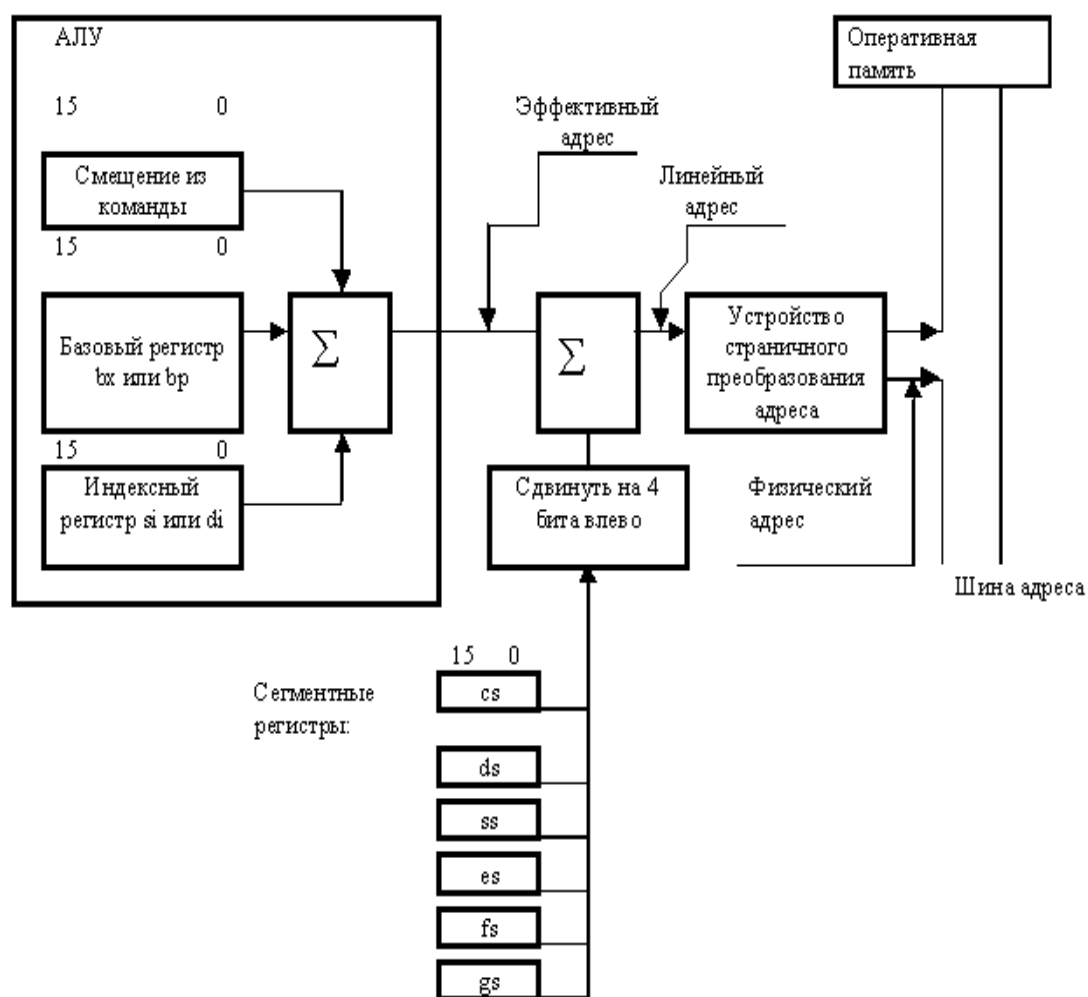


Рис. 5. Механизм формирования физического адреса

Аппаратно поддерживаемые процессором типы данных (рис. 6) имеют свой диапазон значений на каждый из разрядов (табл. 4):

- 1) байт;
- 2) слово;
- 3) двойное слово;
- 4) учетверенное слово.

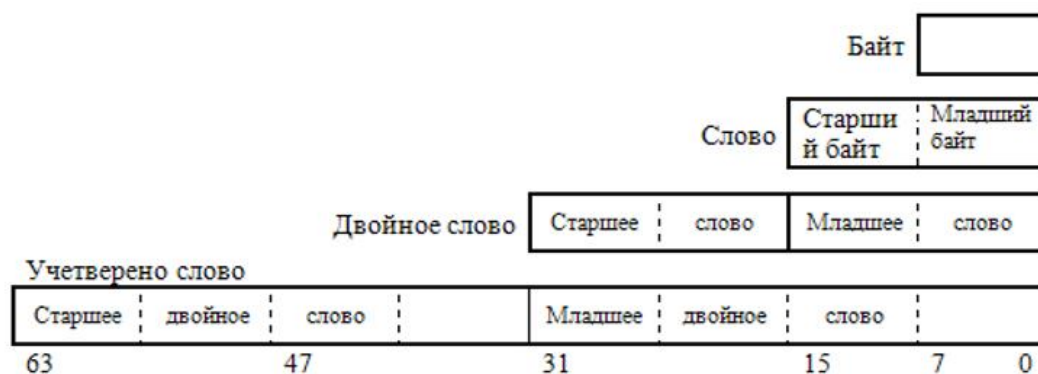


Рис. 6. Типы данных в регистрах

Таблица 4

Диапазон значений

Количество разрядов	Диапазон значений	
	Целое со знаком	Целое без знака
8	-128...127	0...255
16	-32768...32767	0...65535
32	$-2^{31} \dots +2^{31}-1$	$0 \dots +2^{32}-1$

Указатель на память бывает двух типов (рис. 7):

- 1) ближний тип – 32 разряда, отсчитываемый от начала сегмента;
- 2) дальний тип – 48 (16 разрядов – адрес сегмента, 32 разряда – адрес смещения).

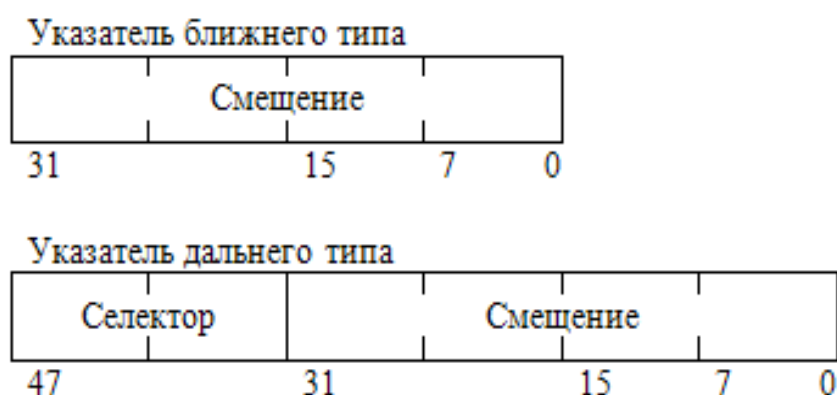


Рис. 7. Указатель на память

Машинная команда процессора имеет следующую структуру:

- 1) поле префиксов;
- 2) поле кода операции;
- 3) поле операндов.

Поле префиксов – элемент команды, который модифицирует действие этой команды, например: замена сегмента, изменение размерности адреса, изменение размерности операнда, циклическое выполнение команды.

Поле кода операции – числовой код команды.

Поле операндов – определяет, с какими ячейками работает команда и куда помещает результат. Поле операндов может содержать от 0 до 2 операндов. Возможны следующие сочетания операндов в команде:

- 1) регистр – регистр;
- 2) регистр – память;
- 3) память – регистр;
- 4) значение – регистр;
- 5) значение – память.

2. ПОСТРОЕНИЕ ПРОГРАММЫ НА АССЕМБЛЕРЕ

Для запуска программы на Ассемблере потребуются: Tasm.exe, Tlink.exe, <имя программы>.asm. Исходный текст программы набирается в любом текстовом редакторе, а затем файлу с программой присваивается формат <asm>. Для трансляции используется программа tasm.exe (рис. 8).

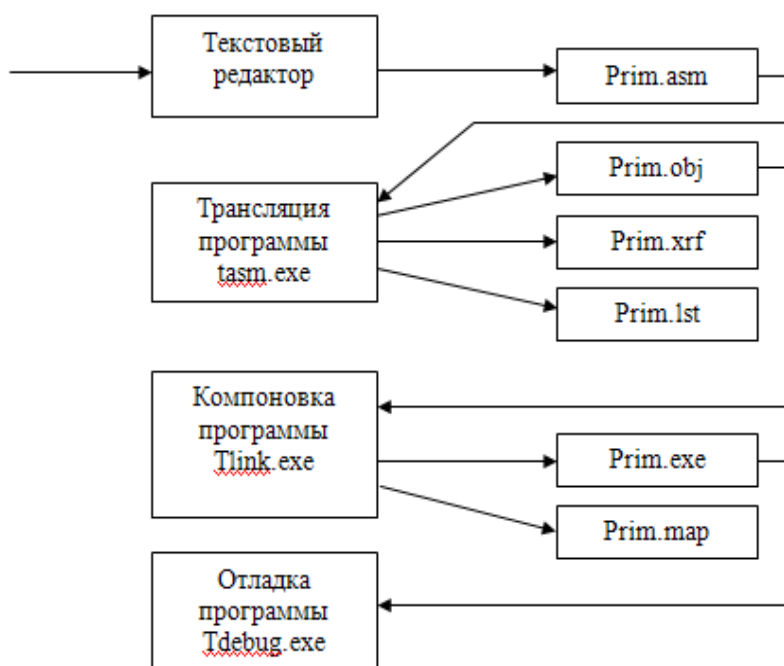


Рис. 8. Построение программы на Ассемблере

Однако для работы на Windows потребуется эмулятор DOSBox (<https://www.dosbox.com/>), поскольку tasm относится к устаревшим программам, запускаемым только под среду MS-DOS. В эмуляторе DOSBox осуществляется следующая последовательность команд:

- 1) mount c e: ; эмуляция диска E: в Windows как диск C: в MS-DOS;
- 2) c: ; переход в эмулированный диск, который будет отображать содержимое диска E:;
- 3) cd <имя директории, содержащей Tasm.exe, Tlink.exe, <имя программы>.asm > ; переход к директории, содержащей файл с текстом программы;
- 4) tasm <имя программы> ; компилирование файла <имя программы>.asm > в <имя программы>.obj для интерпретатор tlink;
- 5) tlink <имя программы>.obj ; преобразование <имя программы>.obj в приложение <имя программы>.exe;
- 6) <имя программы>.exe ; запуск программы. Работа программы будет выводиться на экран эмулятора DOSBox.

Синтаксические диаграммы задают все правила формирования программы на языке Ассемблера (рис. 9). Непосредственным операндом называется число, строка или выражение, имеющие фиксированное значение, оно может быть задано конкретным значением в поле операнда или определено через `equ` или «`=`».

Например:

`r equ 13`

`e = r - 2`

`mov al, r`

`mov al, 13`

Значения `13`, `r`, `e` в приведенном фрагменте являются непосредственными операндами.

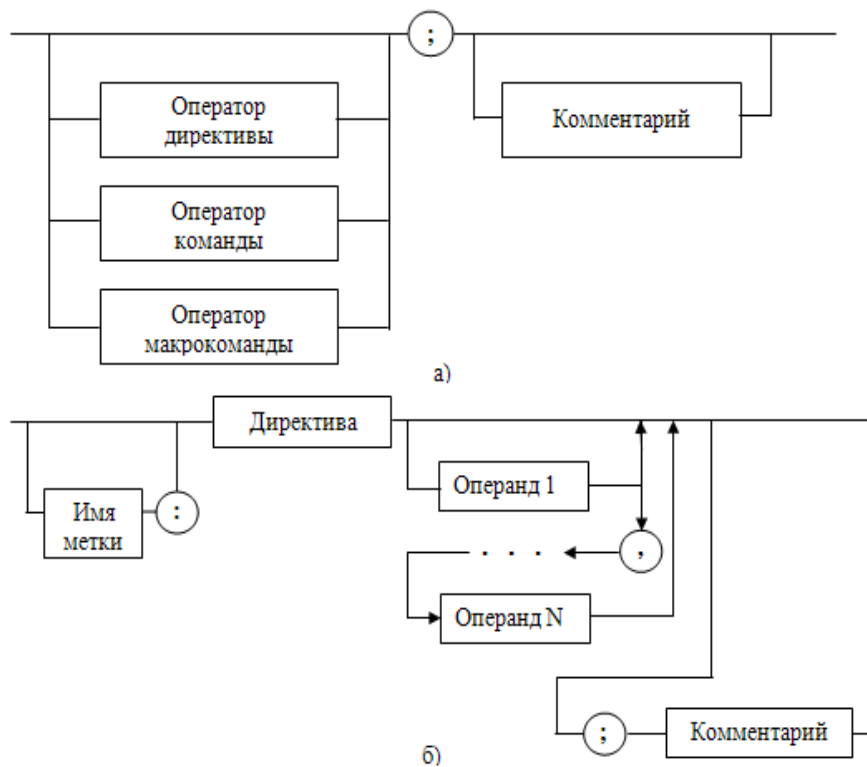


Рис. 9. Структура программы на Ассемблере:

а) с использованием основных операторов директив, макрокоманд и команд;

б) с использованием имен меток и операторов

Адресные операнды задают физическое расположение операнда в памяти (рис. 10).

Например:

`mov ax, ss:0013h`

Приведенный оператор записывает слово из регистра `ax` по адресу, старшая часть которого хранится в регистре `ss`, а младшая имеет значение `0013h`.

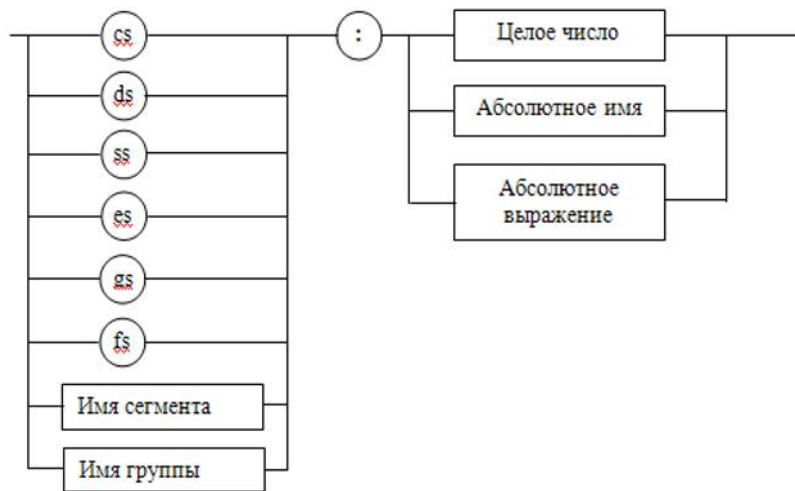


Рис. 10. Адресные операторы

Перемещаемые операнды являются именами переменных или метками инструкций. В отличие от адресных операндов их значение изменяется в зависимости от значения сегментной составляющей адреса.

Например:

data segment

prim dw 25 dup (0)

...

code segment

...

lea si, prim

Конкретное физическое значение физического адреса переменной *prim* будет известно только после загрузки программы.

Счетчик адреса позволяет задавать относительные адреса. Для обозначения текущего значения счетчика адреса используется символ \$.

Например:

jmp \$+3

cld

mov al, 2

В приведенном фрагменте управление передается на оператор пересылки данных *mov*, минуя оператор установки флага управления *cld*, имеющий длину 1 байт.

Операторы сдвига выполняют сдвиг числа на указанное количество разрядов влево или вправо (рис. 11).

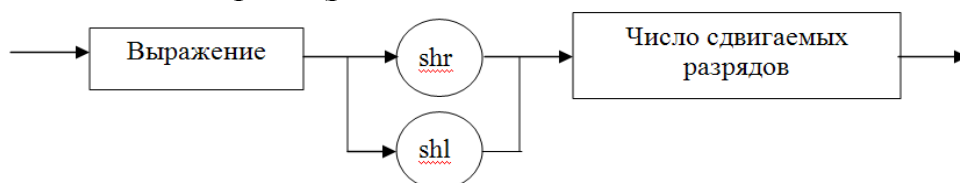


Рис. 11. Оператор сдвига

Оператор сравнения предназначен для формирования логических выражений. Значение «Да» соответствует числу 1, «Нет» – числу 0 (рис. 12). Логические операторы выполняют над аргументами побитовые операции (рис. 13).

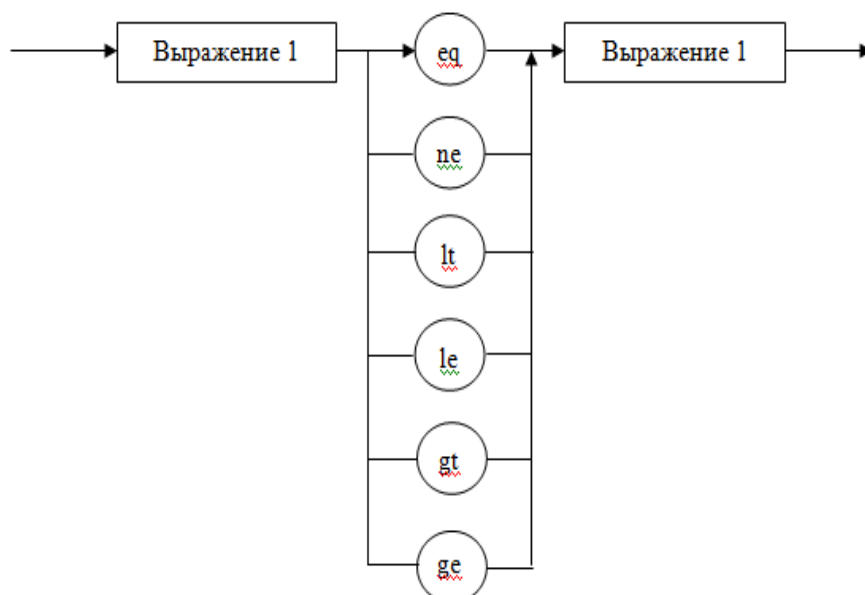


Рис. 12. Оператор сравнения

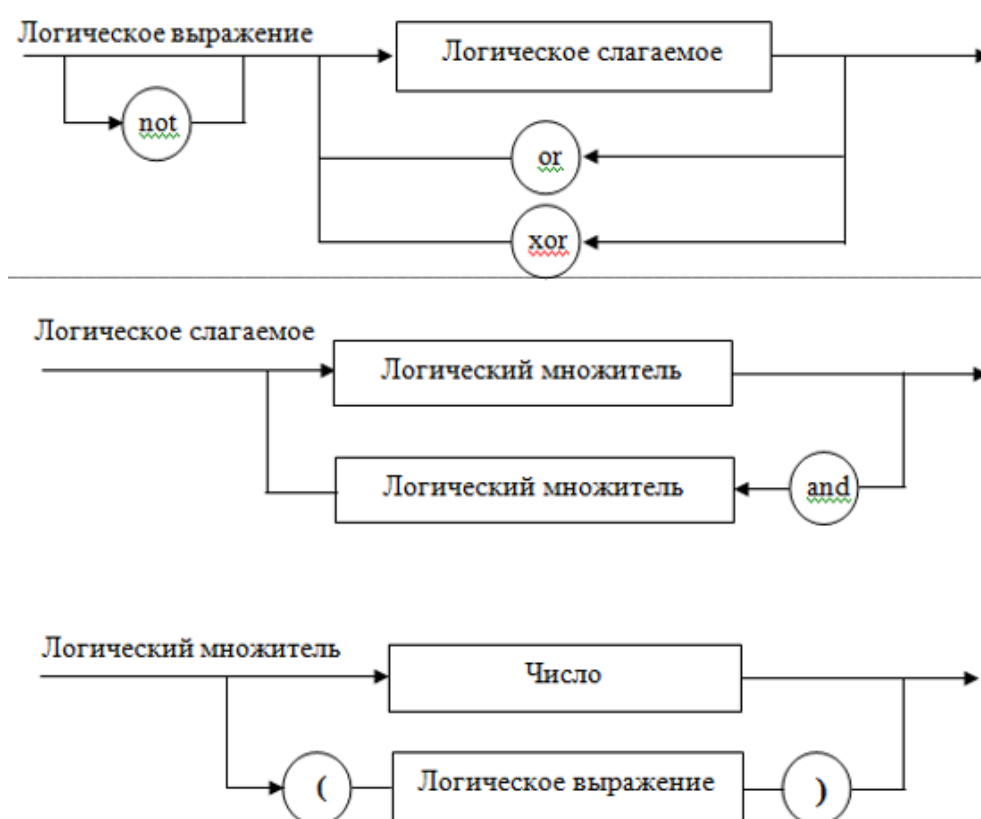


Рис. 13. Логические операторы

Индексный оператор позволяет организовать работу с массивами. В операции используются данные, размещенные по адресу, заданному именем переменной плюс смещение, заданное в квадратных скобках (рис. 14). Оператор *ptr* позволяет преобразовать тип переменной или тип адресации. Возможно использование следующих значений типов: *byte*, *word*, *dword*, *qword*, *tbyte* и два указателя на способ адресации: *near*, *far* (рис. 15).

Например:

mov al, byte ptr d_wrd+1 ; пересылка второго байта из двойного слова

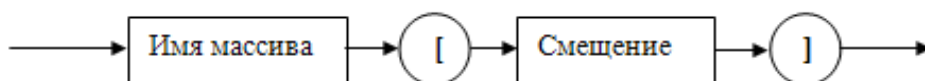


Рис. 14. Индексный оператор

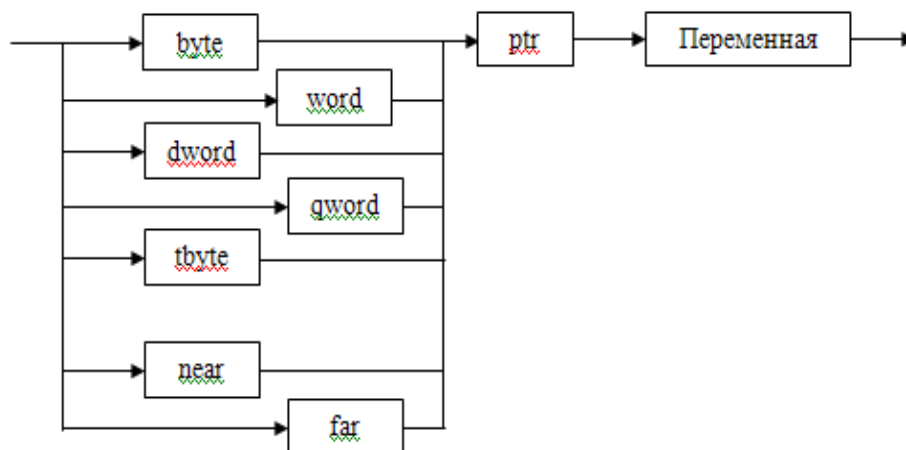


Рис. 15. Оператор преобразования типа

Оператор *SEG* позволяет получить значение сегмента, а *offset* – смещения для указанного адреса.

Например:

mov ex, seg prim
mov dx, offset prim

После выполнения данных операторов в паре регистров *ex:dx* будет полный адрес переменный *prim*.

Программа на Ассемблере может работать с шестью сегментами: кода, сетки и четырех сегментов данных. Для простых программ, содержащих только один сегмент кода, сетки и данных, возможно применение упрощенной модели сегментации. Назначение директив приведено в табл. 5. Назначение переменных идентификатора *model* представлено в табл. 6. Общая организация модели памяти представлена в табл. 7.

Простая структура программы:

model small

.stack (размер)

.data

описание переменных:

.code

main proc

тело программы

main endproc

end

Таблица 5

Назначение директив сегментации

Директива	Описание
model	Выбор модели памяти. Поддерживаются следующие модели памяти: tiny, small, medium, compact, large
.data	Описание переменных программы
.stack	Описание стека
.code	Сегмент для размещения операторов программы
.fardata	Описание переменных, за пределами сегмента

Таблица 6

Служебные переменные

Директива	Описание
@code	Физический адрес сегмента кода
@data	Физический адрес сегмента данных
@stack	Физический адрес сегмента стека

Таблица 7

Модели памяти

Модель памяти	Тип кода	Описание
Tiny	Near	Модель памяти, используемая в .com файлах. Сегмент кода совпадает с сегментом данных
Small	Near	Наиболее часто используемая модель памяти. Все данные объединены в один сегмент
Medium	Far	Каждый программный модуль размещается в своем сегменте, при этом используются длинные адреса. На данные выделяется один сегмент
Compact	Near	Программа размещается в одном сегменте. Данные могут размещаться в различных сегментах, при этом используются длинные адреса
Large	Far	Каждый программный модуль размещается в своем сегменте, везде используются длинные адреса

Для описания простых данных используются директивы резервирования памяти (табл. 8).

Таблица 8

Описание простых типов данных

Имя типа	db	dw	dd	dp	df	dq	dt
Количества памяти	1 байт	2 байта	4 байта	6 байт	6 байт	8 байт	10 байт

При работе с переменными необходимо учитывать следующее – младший байт размещается всегда по младшему адресу (рис. 16).

Например:

model small

.stack 100h

.data

test1 db 12h

test2 dw 10

test3 db 10 dup (' '

test4 db 10 dup (?)

srt1 db 'строка\$'

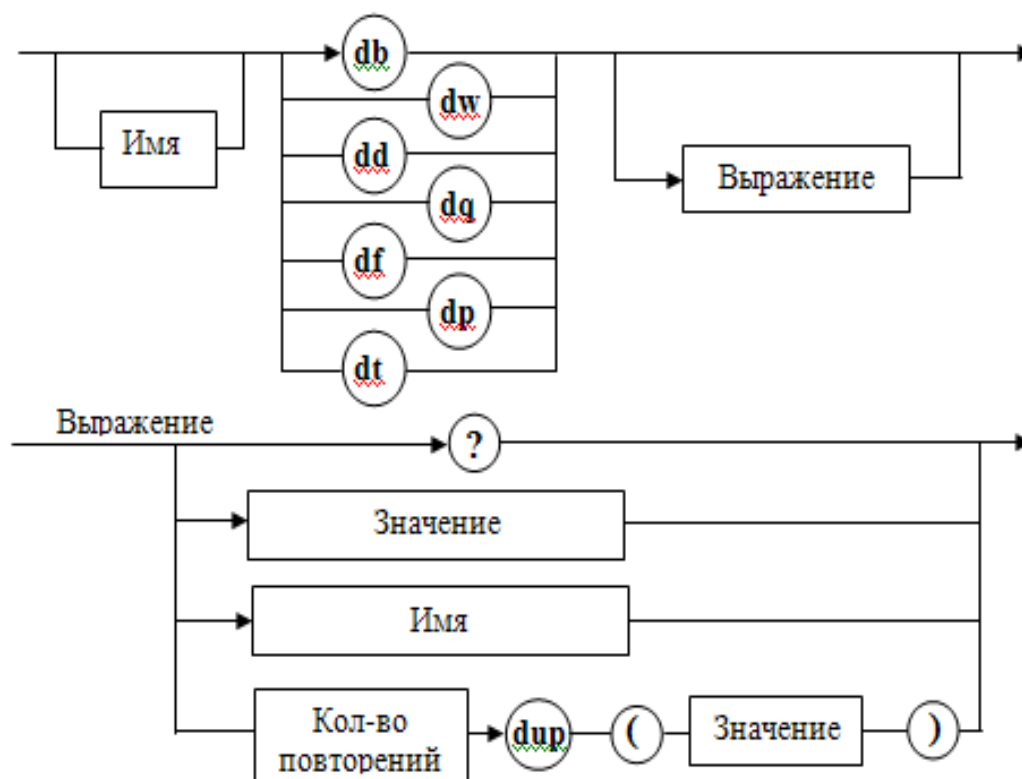


Рис. 16. Работа с переменными

Команда пересылки данных – *mov*. Формат команды:

mov <операнд назначения> <операнд источник>

Особенности команды *mov*:

1) команда *mov* не может непосредственно пересылать данные из одной области памяти в другую, для подобной пересылки необходимо воспользоваться одним из регистров;

2) нельзя загрузить в сегментный регистр значение непосредственно из памяти;

3) нельзя переслать содержимое одного сегментного регистра в другой;

4) нельзя использовать сегментный регистр *cs* в качестве операнда назначения (в данном регистре содержится адрес следующей команды, изменение его значения приведет к сбою работы программы).

При пересылке данных необходимо не забывать о типе переменных, для преобразования типов используется оператор *ptr*.

Для вывода на экран сообщения используется прерывание 21h.

Вывод строки на экран:

mov ah, 09h; поместить в регистр *ah* номер функции прерывания 21h
mov dx, offset str1 ; в регистр *dx* помещается указатель на строку
int 21h ; вызов прерывания 21h

Вывод символа на экран (выводимый символ находится в регистре *dl*):

mov ah, 02h ; поместить в регистр *ah* номер функции прерывания 21h
int 21h ; вызов прерывания 21h

Ввод символа с клавиатуры:

mov ah, 01h ; поместить в регистр *ah* номер функции прерывания 21h
int 21h ; вызов прерывания 21h

Введенный символ находится в регистре *al*.

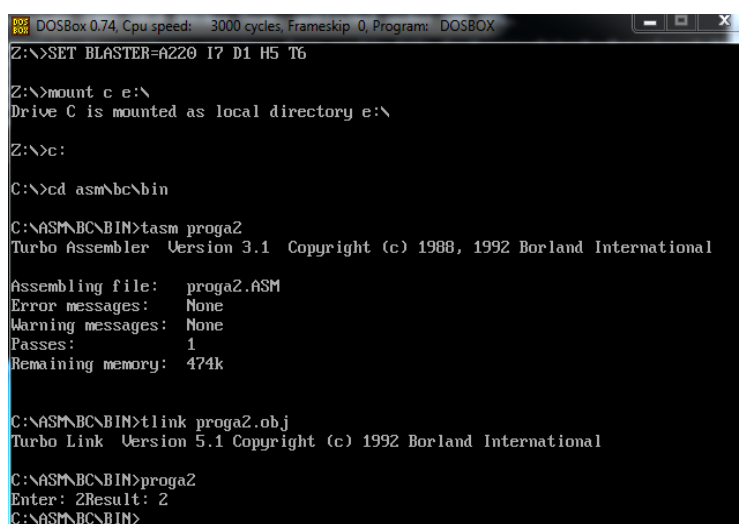
Контролирующая программа сгенерировала следующее задание: необходимо считать символ с клавиатуры и вывести его на консоль. Для создания программы понадобится реализовать операции пересылки и ввода/вывода (рис. 17).

```
model small ; задание модели памяти
.stack 100h ; размер выделенной памяти
.data ; начало блока описания переменных
str1 db 'Enter: $'
str2 db 'Result: $'
.code ; начало блока тела программы
start:
mov ax, @data ; идентификатор доступа к регистру данных ds
mov ds, ax
; вывод строки Enter :
```

```

mov ah,09h ; поместить в регистр ah номер функции прерывания 21h
lea dx,str1 ; в регистр dx помещается указатель на строку
int 21h ; вызов прерывания 21h
; ВВОД СИМВОЛА
mov ah,01h ; помещение в регистр ah номера функции прерывания 21h
int 21h
mov ah,09h ; помещение в регистр ah номера функции прерывания 21h
lea dx,str2 ; в регистр dx помещается указатель на строку
int 21h
mov dl,al ; перевод данных из al в dl для последующего вывода
mov ah,02h ; помещение в регистр ah номера функции прерывания 21h
int 21h
;завершение программы
mov ah,4ch
int 21h
end start

```



```

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX
Z:\>SET BLASTER=A220 I7 D1 H5 T6
Z:\>mount c e:\
Drive C is mounted as local directory e:\
Z:\>c:
C:\>cd asm\bc\bin
C:\ASM\BC\BIN>tasm proga2
Turbo Assembler Version 3.1 Copyright (c) 1988, 1992 Borland International
Assembling file: proga2.ASM
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 474k
C:\ASM\BC\BIN>tlink proga2.obj
Turbo Link Version 5.1 Copyright (c) 1992 Borland International
C:\ASM\BC\BIN>proga2
Enter: 2Result: 2
C:\ASM\BC\BIN>_

```

Рис. 17. Пример выполнения простой программы вывода в DOSBox



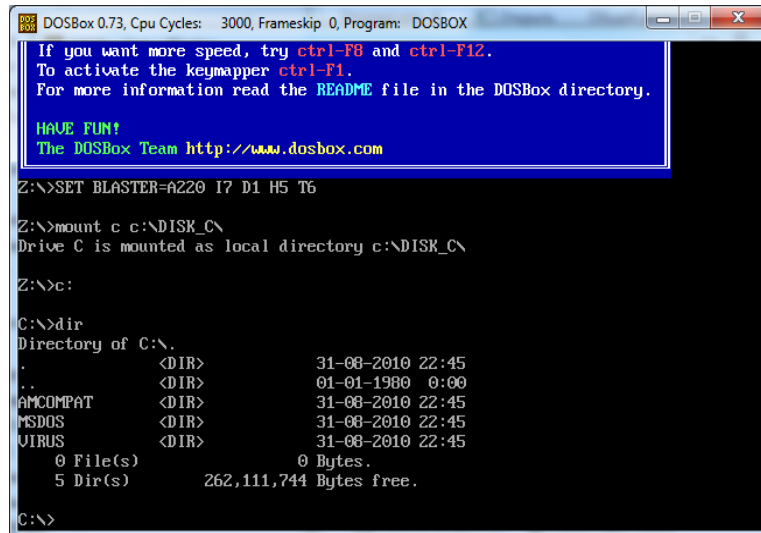
Рис 18. Вид ярлыка DOSBox

Если в качестве эмулятора MS DOS используется DOSBox (рис. 18), то для установки выполните следующие действия:

- 1) запустите DOSBox0.74-win32-installer.exe;
 - 2) в появившемся окне нажмите 2 раза кнопку Next;
 - 3) нажмите кнопку Install. Подождите несколько секунд;
 - 4) после окончания нажмите кнопку Close.
- Для запуска DOSBox щелкните по его ярлыку на рабочем столе.

Для использования в эмуляторе файлов с компьютера необходимо примонтировать какую-либо папку к эмулятору. При этом этой папке на жестком диске ставится в соответствие некоторый диск эмулятора. Для монтирования папки используется команда mount (рис. 19):

mount <Буква диска в эмуляторе> <Папка на диске>



```
DOSBox 0.73, Cpu Cycles: 3000, Frameskip 0, Program: DOSBOX
If you want more speed, try ctrl-F8 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.
HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c c:\DISK_C\
Drive C is mounted as local directory c:\DISK_C\

Z:\>c:

C:\>dir
Directory of C:\.
.                <DIR>                31-08-2010 22:45
..               <DIR>                01-01-1980  0:00
AMCOMPAT        <DIR>                31-08-2010 22:45
MSDOS           <DIR>                31-08-2010 22:45
VIRUS           <DIR>                31-08-2010 22:45
 0 File(s)      0 Bytes.
 5 Dir(s)       262,111,744 Bytes free.

C:\>
```

Рис. 19. Вид окна DOSBox с введенными командами

3. АРИФМЕТИЧЕСКИЕ КОМАНДЫ АССЕМБЛЕРА

Все арифметические целочисленные команды работают с целыми числами двух типов: двоичными и десятичными. Разрядность целого двоичного числа может быть 8, 16 или 32 разряда.

Неупакованный двоично-десятичный тип. Данный тип представляет собой двоичное представление десятичных чисел. Старшие разряды в этом случае всегда равны 0. *Упакованный двоично-десятичный тип.* Данный тип размещает две десятичные цифры в одном байте. На основе этого арифметические команды бывают имеют соответствующую классификацию, представленную на рис. 20.

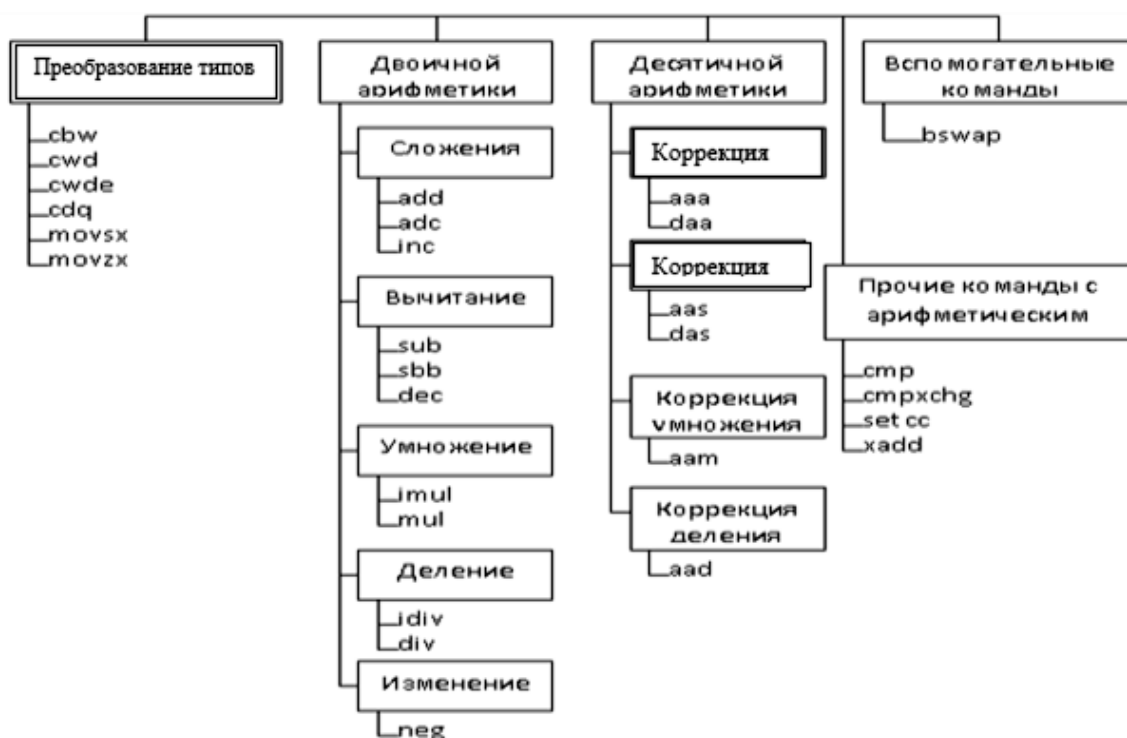


Рис. 20. Целочисленные арифметические команды

Соответственно для команд двоичной арифметики имеется следующее представление (табл. 9).

Таблица 9

Команды двоичной арифметики

Команда	Описание
Сложение двоичных чисел без знака	
inc A1	инкремент операнда A1 (увеличение значения на 1)
add A1, A2	сложение: $A1 = A1 + A2$
adc A1, A2	сложение с учетом флага переноса cf: $A1 = A1 + A2 + cf$

Команда	Описание			
Вычитание двоичных чисел без знака				
dec A1	декремент операнда A1 (увеличение значения на 1)			
sub A1, A2	вычитание: $A1=A1-A2$			
sbb A1, A2	вычитание с учетом флага переноса cf: $A1=A1-A2-cf$			
Умножение двоичных чисел				
mul a1	Умножение двоичных чисел без знака. Умножение операнда A1 на значение регистра a1 (ax, eax). В зависимости от типа a1 получается следующие действия			
	Тип A1	Второй операнд	Результат	
	Байт	al	ah:al= $A1 * al$, 16 бит, в al – младший байт, ah – старший байт	
	Слово	ax	dx:ax = $A1 * ax$, 32 бита, в ax – младшее слово, в dx – старшее слово	
	Двойное слово	eax	edx:eax= $A1 * eax$, 64 бита, в eax – младшее двойное слово, в edx – старшее двойное слово	
imul A1	Команда аналогична команде mul, отличия связаны с формированием знака			
Деление двоичных чисел				
div A1	Деление двоичных чисел без знака. В зависимости от типа делителя (A1) получаются следующие выражения			
	Тип A1	Делимое	Результат	
			Частное	Остаток
	Байт	ax (16 бит)	al (8 бит)	ah (8 бит)
	Слово	dx:ax (32 бита)	ax (16 бит)	dx (16 бит)
	Двойное слово	edx:eax (64 бита)	eax (32 бита)	edx (32 бита)
idiv A1	Команда аналогична команде div, отличия связаны с формированием знака			
neg A1	Смена знака			
xadd A1, A2	Обмен местами и сложение. Реализуется действие $A1=A1+A2$			

Команды преобразования чисел используются в том случае если в арифметических операциях участвуют данные различных типов и их необходимо преобразовать к одному типу (табл. 10).

Таблица 10

Команды преобразования чисел

Команда	Описание
cbw	Преобразование байта в регистре al в слово в регистре ax
cwd	Преобразование слова в регистре ax в двойное слово в регистрах dx:ax
cwde	Преобразование слова в регистре ax в двойное слово в регистре eax
cdq	Преобразование двойного слова в регистре eax в учетверенное слово в регистрах edx:eax
movsx A1, A2	Переслать с преобразованием. Значение A2 (8 или 16 разрядов) пересылается в регистр A1 (16- или 32-разрядный)
Movzx A1, A2	Переслать с преобразованием и очисткой старших разрядов. Значение A2 (8 или 16 разрядов) пересылается в регистр A1 (16- или 32-разрядный). При этом старшие разряды заполняются значением 0. Команда удобна для работы с беззнаковыми данными

Специальных арифметических команд для двоично-десятичных чисел процессор не содержит. Для выполнения арифметических операций с двоично-десятичными числами используются команды двоичной арифметики, результат исполнения которых корректируется с помощью специальных функций – команд десятично-двоичной арифметики (табл. 11). Наличие двоично-десятичных чисел и действий с ними позволяет решить проблему работы с длинными числами.

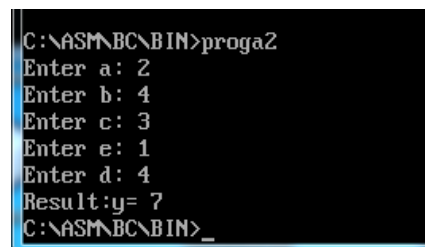
Таблица 11

Команды десятично-двоичной арифметики

Команда	Описание
Действия с неупакованными двоично-десятичными числами	
aaa	Коррекция результата сложения. Аргумент команды содержится в регистре al. Если значение регистра больше, чем 9, то производится корректировка и устанавливается в значение 1 флага переноса cf
aas	Коррекция результата вычитания. Аргумент команды содержится в регистре al. Если значение регистра больше 9, то производится корректировка и устанавливается в значение 1 флага cf, фиксируя заем из предыдущего разряда. Для организации поразрядного вычитания целесообразно использовать команду sbb, учитывающую заем из старшего разряда
amm	Коррекция результата умножения. Команда работает с регистром ax (в этот регистр автоматически помещается результат после выполнения команды mul). Содержимое регистра ax делится на 10, и результат помещается в регистр al, а остаток от деления в регистр ah
aad	Коррекция результата деления. Команда преобразует двухзначное неупакованное число в регистре ax в двоичное число, помещаемое в регистр al. После этого можно воспользоваться командой div

Команда	Описание
Действия с упакованными двоично-десятичными числами	
daa	Коррекция результата сложения упакованных двоично-десятичных чисел. Команда преобразует число в регистре al в две упакованные десятичные цифры. Если результат превышает 99, то устанавливается значение флага переноса cf в 1
das	Коррекция результата вычитания упакованных двоично-десятичных чисел. Команда das преобразует содержимое регистра al в две упакованные десятичные цифры

Приведем пример простой программы с арифметическими операциями. Контролирующая программа сгенерировала следующее задание: необходимо вычислить $(A + B)/C + D + E$. Для создания программы понадобится реализовать операции ввода/вывода и математические операции (рис. 21).



```

C:\ASM\BC\BIN>proga2
Enter a: 2
Enter b: 4
Enter c: 3
Enter e: 1
Enter d: 4
Result:y= 7
C:\ASM\BC\BIN>_

```

Рис. 21. Пример работы программы с арифметическими операциями

```

model small          ; задание модели памяти
.stack 100h          ; размер выделенной памяти
.data                ; начало блока описания переменных
str0 db 13,10,'$'    ; строка переноса
str1 db 'Enter a: $'
str2 db 'Enter b: $'
str3 db 'Enter c: $'
str4 db 'Enter e: $'
str5 db 'Enter d: $'
str6 db 'Result:y= $'
a db ?
b db ?
c db ?
d db ?
e db ?
.code                ; начало блока тела программы
start:
mov ax,@data         ; идентификатор доступа к регистру данных ds
mov ds,ax
; вывод строки "Enter a:"
mov ah,09h           ; поместить в регистр ah номер функции прерывания 21h
lea dx,str1           ; в регистр dx помещается указатель на строку

```

```

int 21h          ; вызов прерывания 21h
; ввод числа a
mov ah,01h
int 21h
sub al,48d       ; преобразование символа в соответствующее десятичное
число
mov a,al         ; перенос введенного значения в переменную a
;перенос строки
mov ah,09h
lea dx,str0
int 21h
; вывод строки "Enter b:"
mov ah,09h
lea dx,str2
int 21h
; ввод числа b
mov ah,01h
int 21h
sub al,48d       ; преобразование символа в соответствующее число
mov b,al         ; перенос введенного значения в переменную b
[ ----- перенос строки ----- ]
; вывод строки "Enter c:"
mov ah,09h
lea dx,str3
int 21h
; ввод числа c
mov ah,01h
int 21h
sub al,48d
mov c,al;
[ ----- перенос строки ----- ]
; вывод строки "Enter d:"
mov ah,09h
lea dx,str4
int 21h
; ввод числа d
mov ah,01h
int 21h
sub al,48d
mov d,al
[ ----- перенос строки ----- ]

```



```

; вывод строки "Enter e:"
mov ah,09h
lea dx,str5
int 21h
; ввод числа e
mov ah,01h
int 21h
sub al,48d
mov e,al
[ ----- перенос строки ----- ]
; начало блока арифметических действий
mov al,a          ; Загружаем значение a в регистр al
add al,b          ; al=al+b, т. е складываем a+b
mov ah,0          ; обнуление ah, регистр остатка от деления должен быть
пуст
div c             ; al=al/c, т. е. (a+b)/c и результат деления записывается в al
add al,d          ; al=al+d, т. е. (a+b)/c+d
add al,e          ; al=al+e, т. е. (a+b)/c+a+e
; вывод строки "Result:y="
mov ah,09h
lea dx,str6
int 21h

mov dl,al         ; перенос значения из al в dl для последующего вывода
add dl,48d        ; преобразование числа в соответствующий десятичный
символ
;вывод результата на экран
mov ah,02h
int 21h
;завершение программы
mov ah,4ch
int 21h
end start

```

4. ЛОГИЧЕСКИЕ КОМАНДЫ АССЕМБЛЕРА

Для начала следует привести основные логические команды Ассемблера, которые могут использоваться в данном языке (рис. 22 и табл. 12). Кроме приведенных команд, процессор поддерживает команды сдвига двойной точности влево и вправо – *shld* и *shrd*.

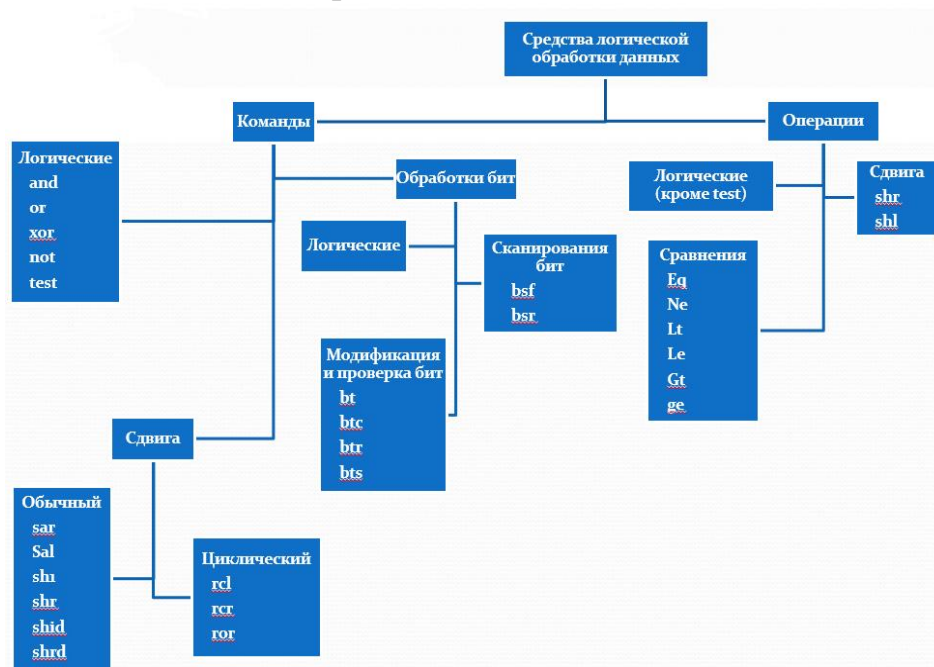


Рис. 22. Основные логические команды Ассемблера

Таблица 12

Команды логической обработки данных

Команда	Описание
Логические команды	
and	Логическая порязрядная операция И. Конъюнкция
or	Логическая порязрядная операция ИЛИ. Дизъюнкция
xor	Логическая порязрядная операция искл. ИЛИ. Сильная дизъюнкция
test	Проверка битов, без изменения значения операндов
not	Логическая порязрядная операция НЕТ. Отрицание
Команды сканирования бит	
bsf	Сканирование битов вперед
bsr	Сканирование бит в обратном порядке
shl	Логический сдвиг влево
shr	Логический сдвиг вправо
sal	Арифметический сдвиг влево
sar	Арифметический сдвиг вправо
rol	Циклический сдвиг битов влево
ror	Циклический сдвиг битов вправо
rcl	Циклический сдвиг влево, через перенос
rcr	Циклический сдвиг вправо, через перенос

Формат команд:

shld A1, A2, кол-во сдвигов,

shrd A1, A2, кол-во сдвигов.

Команды сдвигают биты аргумента A1 (влево или вправо), на указанное количество разрядов (от 1 до 31). При этом значения битов, сдвигающих операнд A1 берутся из операнда A2. Значение операнда A2 не изменяется. Количество сдвигов может быть задано непосредственным операндом или содержаться в регистре *cl*.

В качестве примера целесообразно выполнить следующее задание. Дано поле координат размером $y \in (0;9)$, $x \in (0;9)$, а также находящиеся в этом поле фигуры: прямая ($x = 2$), круг (центр в точке с координатами (4;5), $R = 3$) и прямоугольник ($x \in (5;8)$, $y \in (0;5)$). Требуется написать программу, в которой:

- 1) с клавиатуры вводятся координаты точки;
- 2) анализируется в какую область попала точка (попала ли в фигуры, если попала, то в какие) (рис. 23);

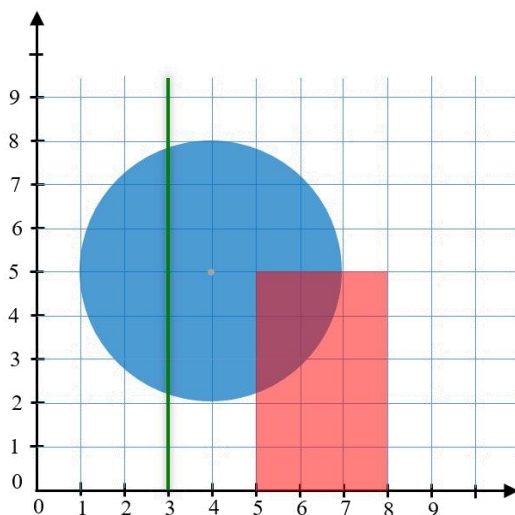


Рис. 23. Задание
для описания точки в области

- 3) выводится на экран (рис. 24) соответствующее сообщение:

$$(x - 4)^2 + (y - 5)^2 = 3^2. \quad (1)$$

Пример точки в области:

model small ; задание модели памяти

.stack 100h ; размер выделенной памяти

.data ; начало блока описания переменных

str1 db 'Enter X: \$'

str2 db 'Enter Y: \$'

str00 db 'Your point got in an empty field! \$' ; точка попала в пустую область

str10 db 'Your point got in the line! \$' ; точка попала на линию

str11 db 'Your point got in the line and cycle! \$' ; точка попала на пересечение линии и круга

str12 db 'Your point got in the cycle! \$' ; точка попала в круг

str13 db 'Your point got in the rectangle! \$' ; точка попала в прямоугольник

str14 db 'Your point got in the rectangle and cycle! \$' ; точка попала в пересечение круга и прямоугольника

str0 db 13,10,'\$' ; строка переноса

X db ?

Y db ?

.code ; начало блока тела программы

start:

;ПЕРЕДАЕМ ИНФОРМАЦИЮ ИЗ РЕГИСТРА ДАННЫХ В СЕГМЕНТ ДАННЫХ

mov ax,@data ; идентификатор доступа к регистру данных ds

mov ds,ax

;1 ВВЕСТИ X Y

;1.1 ВЫВОД СТРОКИ ЗАПРОСА

mov ah,09h ; поместить в регистр ah номер функции прерывания 21h

lea dx,str1 ; в регистр dx помещается указатель на строку

int 21h ; вызов прерывания 21h

;1.2 ВВОД X

mov ah,01h ; помещение в регистр ah номера функции прерывания 21h

int 21h

sub al,48d

mov X,al ; перенос введенного значения в переменную X

;1.3 ПЕРЕНОС С НОВОЙ СТРОКИ

mov ah,09h

lea dx,str0

int 21h

;АНАЛОГИЧНО ДЛЯ Y

;1.4 ВЫВОД СТРОКИ ЗАПРОСА

mov ah,09h ; поместить в регистр ah номер функции прерывания 21h

lea dx,str2 ; в регистр dx помещается указатель на строку

int 21h ; вызов прерывания 21h

;1.5 ВВОД Y ВНУТРЬ

mov ah,01h ; помещение в регистр ah номера функции прерывания 21h

int 21h

```

sub al,48d
mov Y,al          ; перенос введенного значения в переменную Y
;1.6 ПЕРЕНОС С НОВОЙ СТРОКИ
mov ah,09h
lea dx,str0
int 21h
;2.1 ПРОВЕРКА ЛИНИИ (X = 3)
cmp X, 3
je Hit_the_line   ; если X=3 перейти на Hit_the_line
ja Hit_the_right  ; если X>3
jb J_Hit_the_left ; если X<3
jmp exit          ; после возвращения перейти к выходу
_Hit_the_left:    ; вспомогательный прыжок
jmp Hit_the_left
jmp exit
Hit_the_line:
; нужно проверить, попал ли в круг
; для этого нужно вычислить выражение  $(X-4)^2 + (Y-5)^2$ 
; и сравнить результат с 9 ( $R^2$ )
; а) вычитание
    mov bx,0
    mov cx,0
    mov bl, X; bl=X
    mov cl, Y; cl=Y
    sub bl,4d; bl=bl-4=X-4
    sub cl,5d; cl=Y-5
    mov X,bl; X=bl=X-4
    mov Y,cl; Y=Y-5
    mov bl,0
    mov cl,0
; б) возведение в квадрат ч/з умножение
    mov al,X      ; так как умножаем al на что-то, то загружаем в него
mov ah,0          ; обнуление ah
    imul X        ; al=al*X=X*X
    mov X,al      ; достаем из al в X=X*X=(X-4)^2
    mov ah,0
    mov al,Y
    imul Y
    mov Y,al
; в) сложение
    mov ah,0

```

```

        mov al,X           ;достаем в al X
        add al,Y           ;складываем al=X+Y=(X-4)^2+(Y-4)^2
        mov dl,al
        add dl,48d
;d) сравнение
        cmp dl,'9'
        je Print_Line_and_Cycle
        jb Print_Line_and_Cycle
        ja Print_the_line
        jmp exit
Print_Line_and_Cycle:
        mov ah,09h
        lea dx,str11
        int 21h
        jmp exit
Print_the_line:
        mov ah,09h
        lea dx,str10
        int 21h
        jmp exit
Hit_the_right:
        ;проверим прямоугольник
        cmp X,5
        jb JJ_Hit_the_left
        ja Check_Rectangle
        je Check_Rectangle
        jmp exit
JJ_Hit_the_left:           ;опять вспомогательный прыжок
        jmp Hit_the_left
        jmp exit
Check_Rectangle:
        cmp X,8
        ja Print_Empty_Field
        jb Check_Rectangle_Y
        je Check_Rectangle_Y
        jmp exit
Check_Rectangle_Y:
        cmp Y,5
        ja Hit_the_left
        jb Hit_Cycle_and_Rectangle
        je Hit_Cycle_and_Rectangle

```

```

    jmp exit
Hit_Cycle_and_Rectangle:
    ;a) вычитание
    mov bx,0
    mov cx,0
    mov bl, X;bl=X
    mov cl, Y;cl=Y
    sub bl,4d;bl=bl-4=X-4
    sub cl,5d;cl=Y-5
    mov X,bl;X=bl=X-4
    mov Y,cl;Y=Y-5
mov bl,0
    mov cl,0
;b) возведение в квадрат ч/з умножение
    mov al,X          ; так как умножаем al на что-то
                      ; то загружаем в него
    mov ah,0          ; обнуление ah
    imul X ;al=al*X=X*X
    mov X,al          ; достаем из al в X=X*X=(X-4)^2
    mov ah,0
    mov al,Y
    imul Y
    mov Y,al
;c) сложение
    mov ah,0
    mov al,X          ; достаем в al X
    add al,Y          ;складываем al=X+Y=(X-4)^2+(Y-4)^2
    mov dl,al
    add dl,48d
;d) сравнение
    cmp dl,'9'
    je Print_Cycle_and_Rectangle
    jb Print_Cycle_and_Rectangle
    ja Print_the_Rectangle
    jmp exit
Print_Cycle_and_Rectangle:
    mov ah,09h
    lea dx,str14
    int 21h
    jmp exit
Print_Empty_Field:

```

```

mov ah,09h
lea dx,str00
int 21h
jmp exit
Print_the_Rectangle:
mov ah,09h
lea dx,str13
int 21h
jmp exit
Hit_the_left:
;проверить попадает ли в круг
;a) вычитание
mov bx,0
    mov cx,0
    mov bl,X;bl=X
    mov cl,Y;cl=Y
    sub bl,4d;bl=bl-4=X-4
    sub cl,5d;cl=Y-5
    mov X,bl;X=bl=X-4
    mov Y,cl;Y=Y-5
    mov bl,0
    mov cl,0
;b) возведение в квадрат ч/з умножение
    mov al,X          ; так как умножаем al на что-то
                      ; то загружаем в него
    mov ah,0          ; обнуление ah
    imul X ;al=al*X=X*X
    mov X,al          ; достаем из al в X=X*X=(X-3)^2
    mov ah,0
    mov al,Y
    imul Y
    mov Y,al
;c) сложение
mov ah,0
    mov al,X          ; достаем в al X
    add al,Y          ; складываем al=X+Y=(X-3)^2+(Y-3)^2
    mov dl,al
    add dl,48d
;d) сравнение
    cmp dl,'9'
    je Print_the_Cycle

```



```

    jb Print_the_Cycle
    ja Print_Empty_Field
jmp exit
Print_the_Cycle:
    mov ah,09h
    lea dx,str12
    int 21h
    jmp exit
;завершение программы
exit:
    mov ah,4ch
    int 21h
end start

```

```

cmp
Enter X: 2
Enter Y: 2
Your point got in an empty field!
C:\TYR\BC\BIN>cmp
Enter X: 2
Enter Y: 5
Your point got in the cycle!
C:\TYR\BC\BIN>cmp
Enter X: 3
Enter Y: 5
Your point got in the line and cycle!
C:\TYR\BC\BIN>cmp
Enter X: 6
Enter Y: 1
Your point got in the rectangle!
C:\TYR\BC\BIN>cmp
Enter X: 6
Enter Y: 4
Your point got in the rectangle and cycle!
C:\TYR\BC\BIN>cmp
Enter X: 3
Enter Y: 1
Your point got in the line!
C:\TYR\BC\BIN>_

```

Рис. 24. Результат работы программы по описанию точки в области

5. ЦИКЛЫ В АССАМБЛЕРЕ

Все команды передачи управления можно разделить на группы:

- 1) безусловная передача управления:
 - а) безусловный переход,
 - б) вызов и возврат из процедуры,
 - в) вызов и возврат из программных прерываний;
- 2) условная передача управления:
 - а) переход по результатам сравнения,
 - б) переход по состоянию флага,
 - в) переход по состоянию регистра есх/сх;
- 3) команды управления циклами:
 - а) цикл со счетчиком есх/сх,
 - б) цикл со счетчиком есх/сх с возможностью выхода из цикла по дополнительному условию.

При организации переходов используются метки. Метка в Ассемблере имеет следующие атрибуты:

имя сегмента; смещение; тип метки.

Тип метки может принимать следующие значения:

- 1) near – адрес метки определен только в пределах сегмента;
- 2) far – адрес метки определен полным адресом.

Метка может быть задана двумя способами:

- 1) оператором «:» (только для меток типа near);
- 2) ключевым словом label.

Команда безусловного перехода переходит к оператору программы, помеченному указанной меткой.

Синтаксис команды безусловного перехода:

jmp [модификатор] адрес перехода.

Модификатор позволяет **преобразовать адрес перехода** к определенному типу. Использование режима short ptr позволяет реализовать переходы не более чем на 127 байт вниз и не более чем на 128 байт вверх (диапазон числа типа short –128...+127). Этот режим соответствует двухбайтному варианту команды jmp.

Прямой вариант команды jmp позволяет организовать переходы в пределах 64 Кбайт вверх и вниз по программе.

Косвенный переход. При косвенном переходе в команде jmp задается адрес переменной, в которой содержится адрес перехода.

Процессор поддерживает 18 команд организации перехода в зависимости от условий. Сравнение операндов осуществляется с помощью команды cmp. Формат команды:

cmp операнд1, операнд2

Результаты сравнения записываются в регистр флагов. **Команда перехода** осуществляет переход на метку, в зависимости от значения регистра флагов, установленных командой `str`. Кроме этого процессор поддерживает группу команд перехода в зависимости от значения регистров флагов.

Еще одной возможностью организации перехода является контроль значения регистра `ecx/cx`.

Синтаксис команды:

*jcxz метка перехода,
jesxz метка перехода.*

Команда `jcxz` осуществляет переход, если `cx = 0`,

Команда `jesxz` осуществляет переход, если `ecx = 0`.

Приведенные ранее команды условных переходов позволяют организовывать циклы. Однако в языке Ассемблера для организации циклов предусмотрены специальные команды, представленные в табл. 13. Регистр `ecx/cx` используется при организации циклов в качестве служебного.

Таблица 13

Команды передачи управления. Организация циклов

Команда	Описанные действия
Loop	<ul style="list-style-type: none"> • декремент регистра <code>ecx/cx</code>; • проверка регистра <code>ecx/cx</code>, если <code>ecx/cx > 0</code>, то осуществляется переход на указанную метку; • если <code>ecx/cx = 0</code>, то управление передается на следующую после <code>Loop</code> команду
Loope Loopz	<ul style="list-style-type: none"> • декремент регистра <code>ecx/cx</code>; • проверка регистра <code>ecx/cx</code> и флага <code>zf</code>, если в случае <code>ecx/cx > 0</code> и <code>zf = 1</code>, то управление передается на указанную метку; • если <code>ecx/cx = 0</code> и <code>zf = 0</code>, то управление передается на следующую после <code>Loop(Loopz)</code> команду
Loopne Loopnz	<ul style="list-style-type: none"> • декремент регистра <code>ecx/cx</code>; • проверка регистра <code>ecx/cx</code> и флага <code>zf</code>, если в случае <code>ecx/cx > 0</code> и <code>zf = 0</code>, то управление передается на указанную метку; • если <code>ecx/cx = 0</code> и <code>zf = 1</code>, то управление передается на следующую после <code>Loop(Loopz)</code> команду

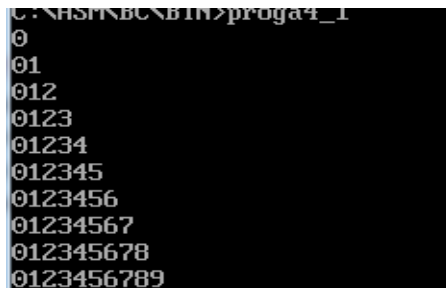
Для организации цикла предназначена команда **Loop**. У этой команды один операнд – имя метки, на которую осуществляется переход (рис. 25). В качестве счетчика цикла используется регистр `CX`. Команда `Loop` выполняет декремент `CX`, а затем проверяет его значение. Если содержимое `CX` не равно нулю, то осуществляется переход на метку, иначе управление переходит к следующей после `Loop` команде. Содержимое `CX` интерпретируется командой как число без знака. В `CX` нужно помещать число, равное требуемому количеству повторений цикла.

*model small
.stack 100h*

```

.data
str0 db 13,10,'$'           ;строка переноса
str1 db 'numbers: $'
i dw 1d
.code
start:
    mov ax,@data            ; идентификатор доступа к регистру данных ds
    mov ds,ax
    ; вывод строки "numbers:"
    ;mov ah,09h
    ;lea dx,str1
    ;int 21h
    mov dl,48d              ;числа начнутся с нуля
    mov cx,10d              ;количество повторений внешнего цикла
    mov ah,02h
metka1:
    mov bx,cx               ;сохраняем счетчик внешнего цикла в bx
    mov cx,i                ;количество повторений внутреннего цикла
    mov ah,02h
    mov dl,48d
metka2:
    int 21h                 ; вывод числа из dl
    inc dl                  ; dl++ переход от символа к следующему числу
    Loop metka2             ; пока cx не равен 0
    ;перевод в новую строку
    mov ah,09h
    lea dx,str0
    int 21h
    inc i                   ;счетчик внутреннего цикла ++
    mov cx,bx               ;возвращаем в cx счетчик внешнего цикла
    Loop metka1             ;пока cx не равен 0
    ;завершение программы
    mov ah,4ch
    int 21h
    end start

```



```

C:\HSM\BC\BIN>prg04_1
0
01
012
0123
01234
012345
0123456
01234567
012345678
0123456789

```

Рис. 25. Организация цикла на Ассемблере
в качестве примера

6. РЕВЕРС-ИНЖИНИРИНГ

Обратная разработка (обратный инжиниринг, реверс-инжиниринг или, кратко, реверсинг; от англ. reverse engineering) – это «исследование некоторого устройства или программы, а также документации на них с целью понять принцип его работы и, чаще всего, воспроизвести устройство, программу или иной объект с аналогичными функциями, но без копирования как такового». Реверс ПО применяется для анализа и взлома, а также для исследования работы вредоносных программ, с целью их дальнейшего обезвреживания. Анализ вредоносного кода – это целая индустрия в области обеспечения информационной безопасности. Им занимаются и антивирусные лаборатории, выпускающие свои продукты для защиты, и узкоспециализированные группы экспертов, и даже сами, конкурирующие между собой, «вирусописатели». Анализ кода требует нестандартного и креативного подхода, не существует универсальной методики для успешного взлома. Однако общие методики анализа, которых следует придерживаться, остаются уже долгое время неизменными. Иными словами, реверс – исследование и воссоздание алгоритмов работы программы, не имея на руках исходных кодов. По сравнению с анализом вредоносных программ тут возникает несколько нюансов. Во-первых, реверсинг ПО в абсолютном большинстве случаев запрещается лицензионным соглашением. Вот, например, как выглядит лицензионное соглашение для Kaspersky Rescue Disk 10: *«Запрещается декомпилировать, дизассемблировать, модифицировать или выполнять производные работы, основанные на ПО, целиком или частично за исключением случаев, предусмотренных применимым законодательством»*. Анализ же вирусов таких ограничений не содержит, более того, это «дело благородное». Во-вторых, реверсинг, как правило, направлен в сторону коммерческого ПО, делающего из trial или незарегистрированной версии ПО вполне рабочую. Иными словами, это распространение пиратских копий ПО. Эти действия нарушают множество статей авторского и интеллектуального права, патентного законодательства, международных соглашений и тому подобного.

Чаще всего при анализе вредоносного программного обеспечения в распоряжении исследователя есть только исполняемый файл или библиотека, скомпилированная в двоичном виде. Для того чтобы понять, как бинарный код работает, необходимо использовать специальные подходы и приемы. Существует два основных подхода к анализу программ: **статический** и **динамический**. При статическом анализе программы изучают, не запуская их на исполнение. Динамический же анализ включает в себя запуск программ и манипуляции с ними в оперативной памяти. Оба метода условно можно разделить на **базовый** и **продвинутый** анализ. **Базовый статический анализ** состоит из изучения исполняемого файла без просмотра машинных инструкций. **Базовый динамический анализ** связан с запуском

программы и наблюдением за ее поведением в системе. **Продвинутый статический анализ** подразумевает под собой загрузку исполняемого файла в дизассемблер без запуска кода в оперативной памяти и просмотр ассемблерных инструкций на предмет того, что делает код программы в системе. **Продвинутый динамический анализ** использует отладчик для изучения внутреннего состояния выполняемого кода в оперативной памяти.

6.1. Некоторые инструменты статического анализа

PEview. Позволяет просматривать информацию, хранящуюся в таблице PE-заголовков файлов и в различных сегментах файла (рис. 26).

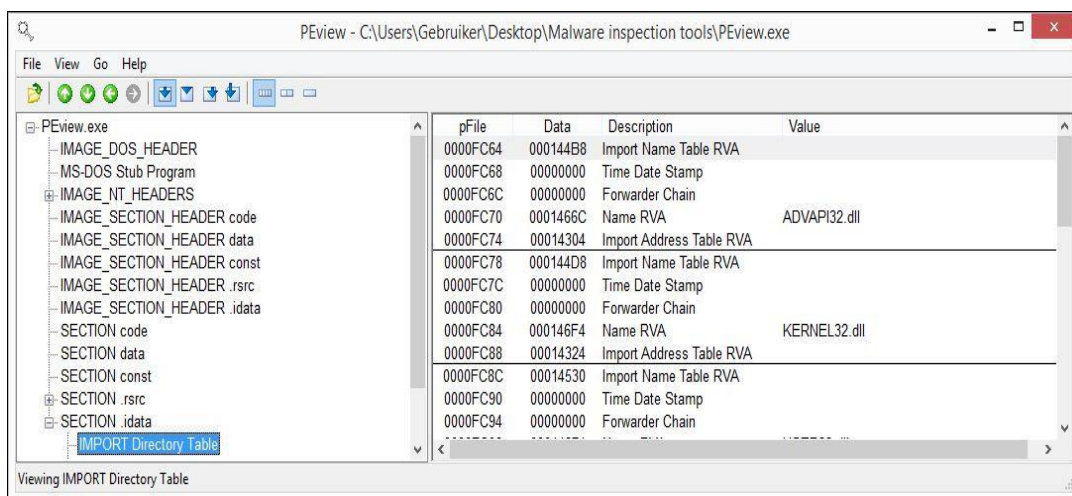


Рис. 26. PEview

FileAlyzer. Используется для чтения информации, хранящейся в PE-заголовках файлов (рис. 27), но предлагает немного больше функций и возможностей, чем PEview.

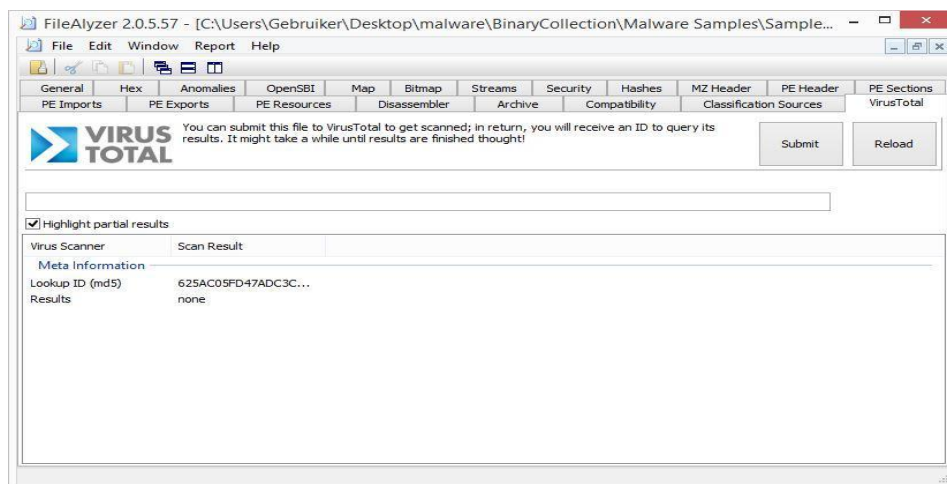


Рис. 27. FileAlyzer

PEiD. Используется для анализа бинарного файла и обнаружения стандартных упаковщиков, криптооров и компиляторов. PEiD ищет в исполняемом файле сигнатуры, характерные для исполняемых/бинарных файлов, полученных в результате упаковки или компиляции (рис. 28).

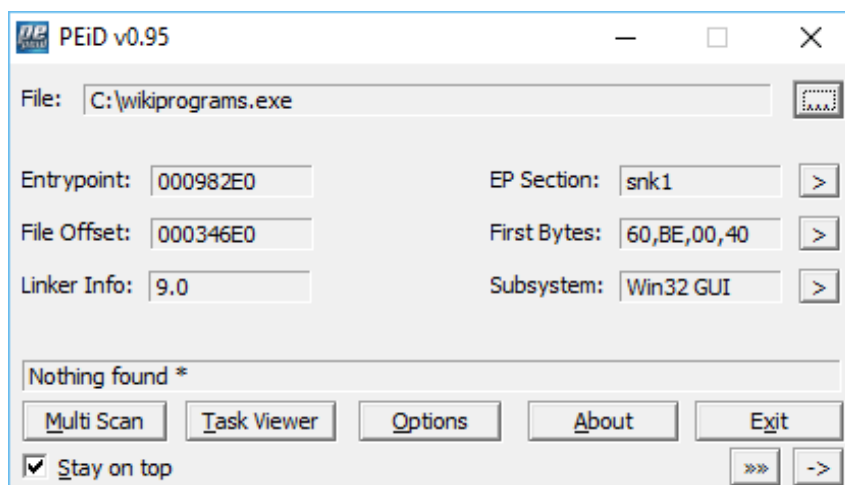


Рис. 28. PEiD

Dependency Walker. Используется для сканирования 32- и 64-битных модулей Windows (.exe, библиотеки DLL, .osx и т. д.), а также для получения списка таблиц импорта и экспорта (рис. 29).

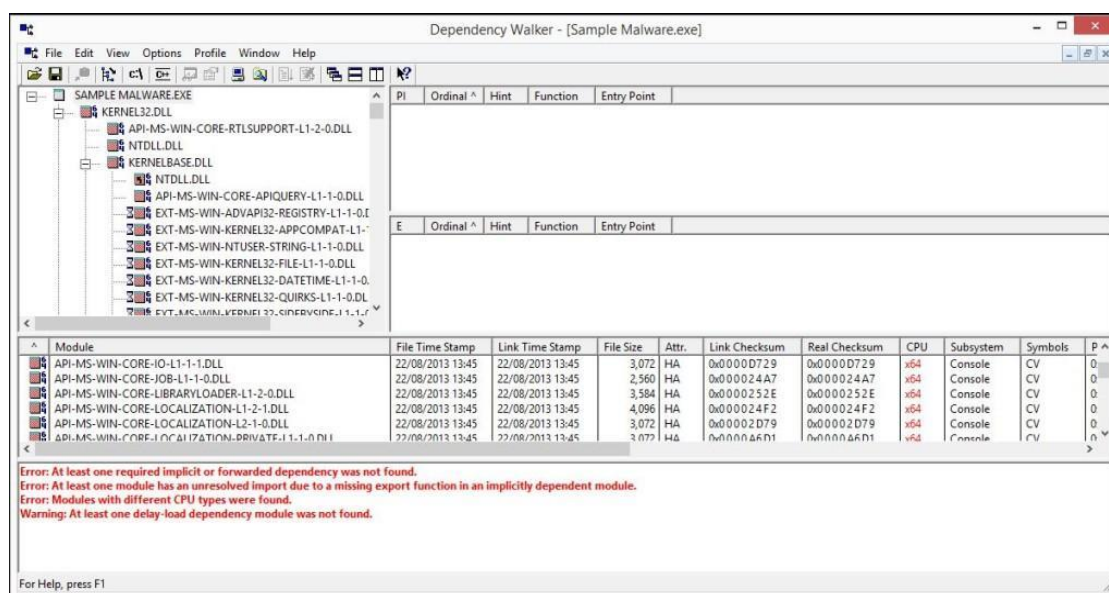


Рис. 29. Walker

Resource Hacker. Используется для извлечения ресурсов из бинарников Windows-программ (рис. 30). Позволяет добавлять, извлекать и изменять строки, изображения, меню, диалоги.

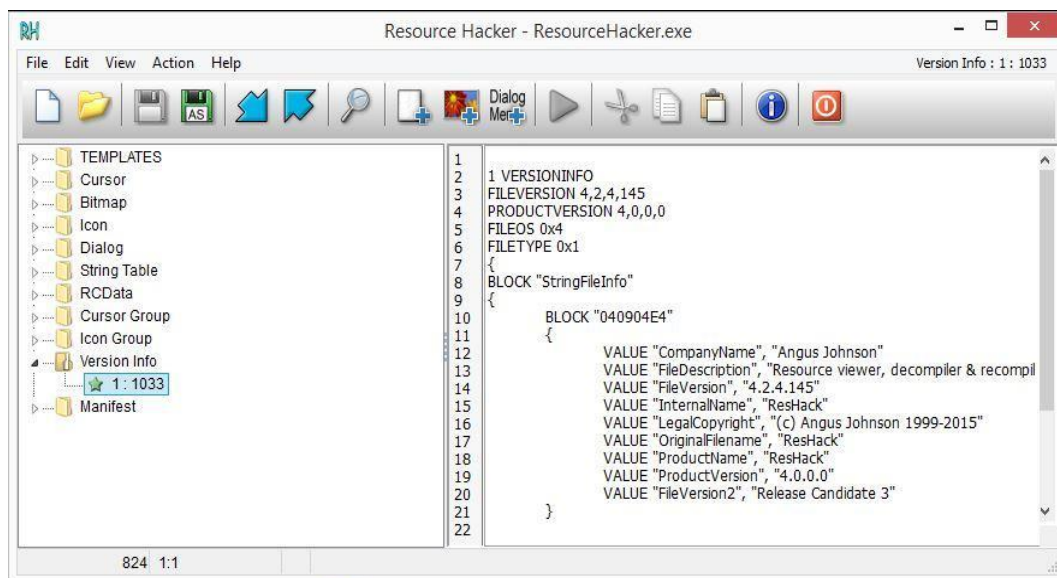


Рис. 30. Resource Hacker

6.2. Некоторые инструменты динамического анализа

Procmon. Используется для мониторинга файловой системы ОС Windows, реестра и процессов в реальном времени (рис. 31).

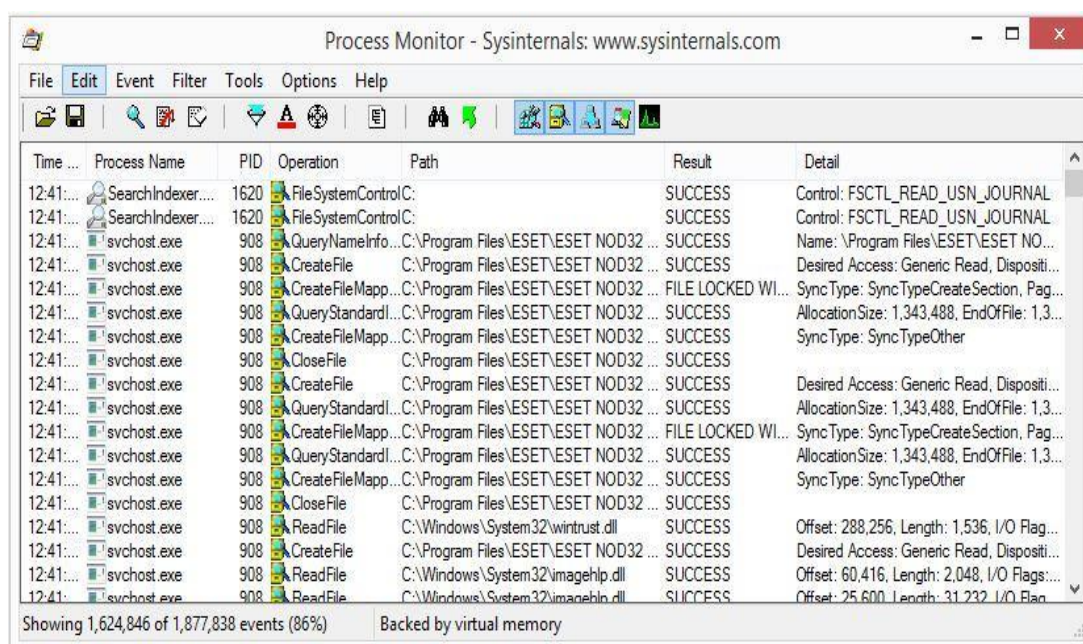


Рис. 31. Procmon

Process Explorer. Используется при выполнении динамического анализа программ. Программа показывает, какие приложения и DLL-файлы выполняются и загружаются для каждого процесса (рис. 32).

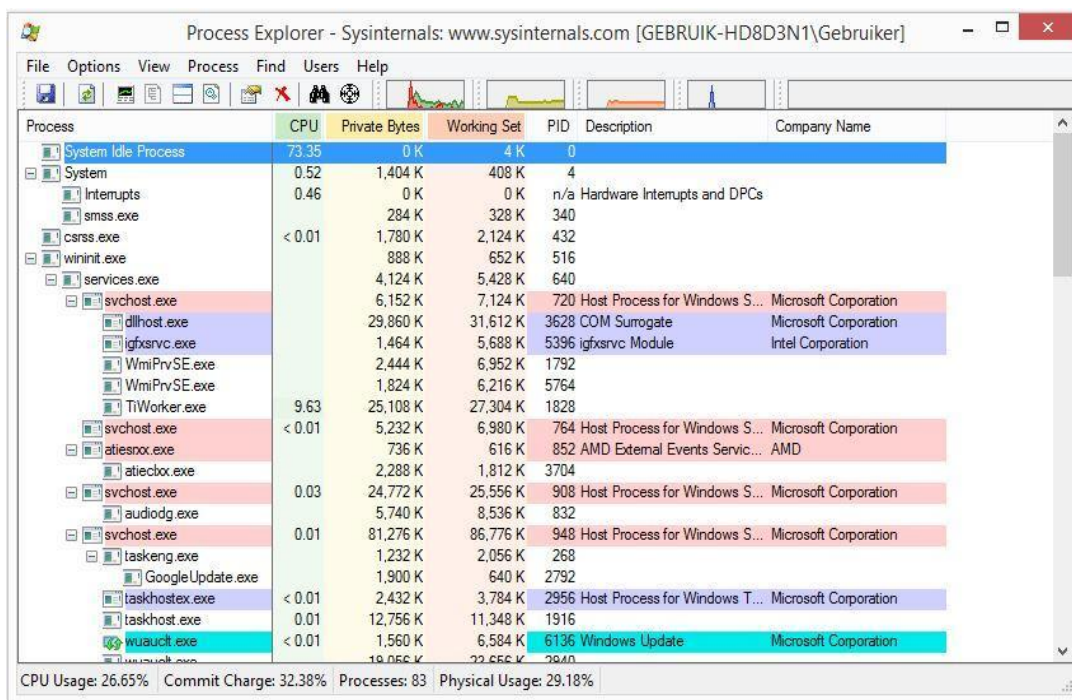


Рис. 32. Process Explorer

Regshot. Используется для мониторинга изменений реестра с возможностью моментального снимка, который можно сравнить с эталонным состоянием реестра. Это позволяет видеть изменения, внесенные после того, как программа была запущена в системе (рис. 33).

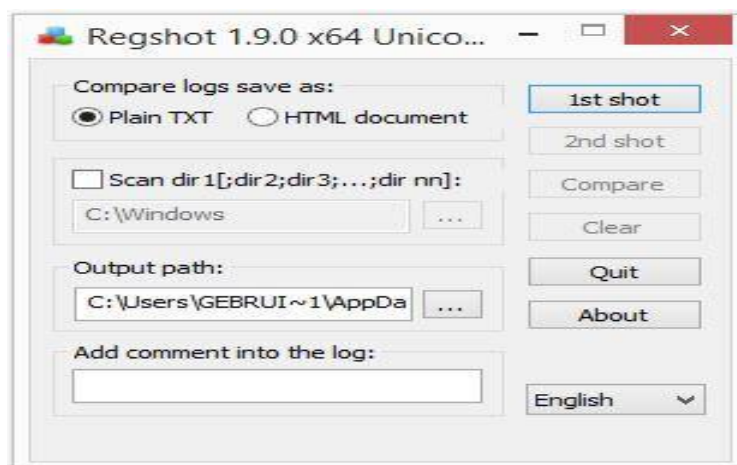


Рис. 33. Regshot

6.3. Специализированные инструменты для продвинутого анализа

OllyDbg. Это отладчик со встроенным 32-битным ассемблером и интуитивным интерфейсом. Поддерживает все инструкции вплоть до SSE (рис. 34). Работает исключительно с x32 исполняемыми файлами.

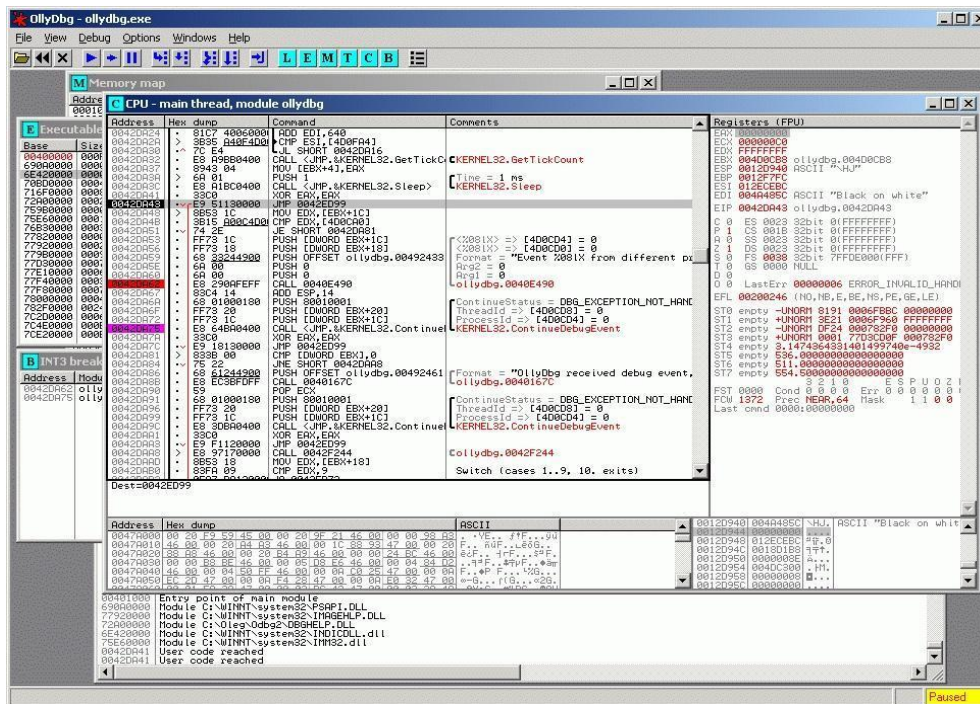


Рис. 34. OllyDbg

WinDbg, отладчик уровня ядра под Windows. Многоцелевой отладчик для ОС Windows. Может быть использован для отладки приложений в режиме пользователя, драйверов устройств и самой операционной системы в режиме ядра (рис. 35).

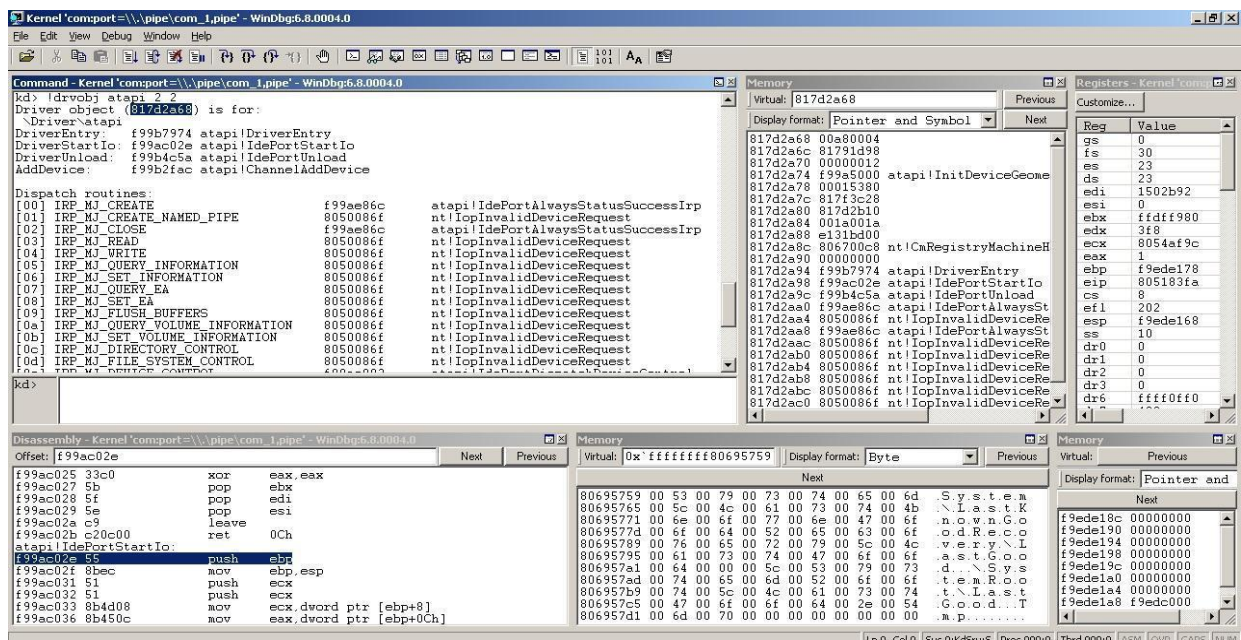


Рис. 35. WinDbg

Дизассемблер IDA Pro. Интерактивный дизассемблер, который широко используется для реверс-инжиниринга. Отличается исключительной

гибкостью, наличием встроенного командного языка, поддерживает множество форматов исполняемых файлов для большого числа процессоров и операционных систем (рис. 36). Позволяет строить блок-схемы, изменять названия меток, просматривать локальные процедуры в стеке и многое другое.

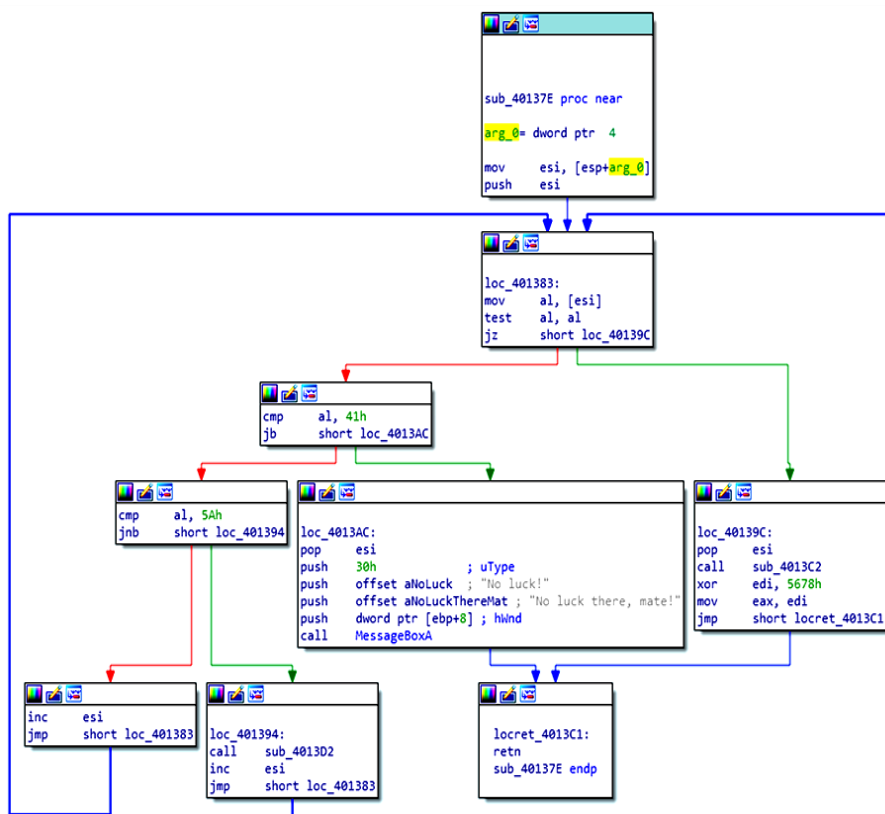


Рис. 36. Дизассемблер IDA Pro

7. СТРУКТУРА PE-ФОРМАТА

PE формат – это формат исполняемых файлов всех 32- и 64-разрядных Windows систем [47]. На данный момент существует два формата PE-файлов: PE32 для x86 систем, а PE32+ – для x64. На рис. 37 представлена структура данного формата.

Первая структура формата – **IMAGE_DOS_HEADER** [47]. Она имеет размер 64 байта и 19 полей, наиболее важные из них – `e_magic` (сигнатура, находящаяся по нулевому смещению от начала файла и равная «MZ») и `e_lfnew`. Поле `e_lfnew` (находится по смещению 0x3C от начала файла) содержит смещение PE заголовка относительно начала файла.

После 64 первых байт файла расположен **DOS-STUB** [47], необходимый только для обратной совместимости и нынешним системам он ни к чему. Поэтому в этой области памяти в большинстве своем расположены нули.

Следующая структура – **IMAGE_NT_HEADERS32** (PE заголовок) [47]. Она содержит 4-байтовую сигнатуру, которая характеризует формат файла, и указатели еще на два заголовка – **IMAGE_FILE_HEADER** и **IMAGE_OPTIONAL_HEADER32**. Для PE-формата сигнатура равна – PE\х0\х0. PE заголовок может располагаться в любом месте файла.

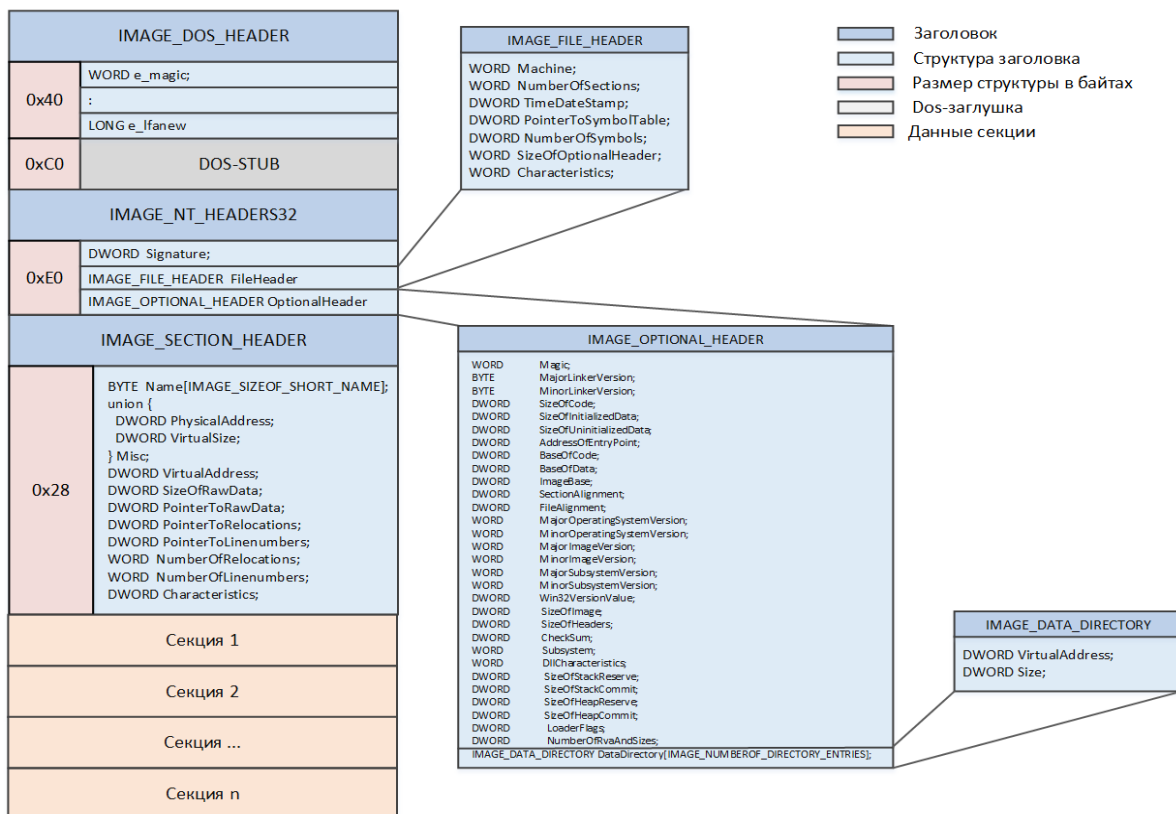


Рис. 37. PE32-формат

Заголовок (структура) **IMAGE_FILE_HEADER** описывает базовые характеристики файла [47] (табл. 14).

Таблица 14

Заголовок (структура) IMAGE_FILE_HEADER

№	Имя поля	Размер, байт	Описание
1	Machine	2	Архитектура процессора, на которой данное приложение может выполняться
2	NumberOfSections	2	Количество секций в файле. Таблица секций следует сразу после заголовка (PE-Header). Количество секций ограничено числом 96
3	TimeDateStamp	4	Дата и время создания файла
4	PointerToSymbolTable	4	Смещение (RAW) до таблицы символов. Таблица служит для хранения отладочной информации, но чаще всего это поле содержит ноль
5	SizeOfOptionalHeader	4	Размер таблицы символов
6	SizeOfOptionHeader	2	Размер IMAGE_OPTIONAL_HEADER (что следует сразу за текущим). Для объектного файла он устанавливается в 0
7	Characteristics	2	Характеристики файла

В табл. 15 указаны флаги поля Characteristics, относящиеся к исполняемым файлам.

Таблица 15

Флаги поля Characteristics

Название	Значение	Описание
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Файл является исполняемым
IMAGE_FILE_AGGRESSIVE_WS_TRIM	0x0010	Рекомендация операционной системе агрессивно выталкивать виртуальную память данного процесса в файл подкачки. Этот флаг устанавливается для сервисов, которые «просыпаются» очень редко
IMAGE_FILE_LARGE_ADDRESS_AWARE	0x0020	Программа может работать с адресами, больше 2 Гб
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Эти флаги устанавливаются, если порядок байтов (endianess) в файле отличен от ожидаемого ОС. Первый флаг говорит, что порядок байтов в памяти little-endian, второй – что big-endian. По-видимому, для исполняемых файлов эти флаги не применяются, так как загрузчик предполагает, что порядок байтов в файле правильный
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	

Название	Значение	Описание
IMAGE_FILE_32BIT_MACHINE	0x0100	Ожидается 32-разрядный процессор. Всегда установлен
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Отладочная информация вынесена в отдельный файл .DBG. Применимость этого флага к исполняемым файлам мне неизвестна
IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	0x0400	Указание операционной системе, что при загрузке с дискеты или компакт-диска этот файл нужно предварительно скопировать в файл подкачки
IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	Указание операционной системе, что при загрузке из сети этот файл нужно предварительно скопировать в файл подкачки
IMAGE_FILE_SYSTEM	0x1000	Файл является системным
IMAGE_FILE_DLL	0x2000	Файл является динамической библиотекой
IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	Файл может исполняться только на однопроцессорной машине

Заголовок **IMAGE_OPTIONAL_HEADER**. Этот заголовок является обязательным и имеет 2 формата PE32 и PE32+ (**IMAGE_OPTIONAL_HEADER32** и **IMAGE_OPTIONAL_HEADER64** соответственно). Формат хранится в поле Magic. Заголовок содержит необходимую информацию для загрузки файла. Поля структуры [47] приведены в табл. 16.

Таблица 16

Заголовок IMAGE_OPTIONAL_HEADER

№	Имя поля	Размер (PE32/PE32+)	Описание
1	Magic	2	Сигнатура заголовка
2	MajorLinkerVersion	1	Старшая цифра номера версии сборщика
3	MinorLinkerVersion	1	Младшая цифра номера версии сборщика
4	SizeOfCode	4	Размер всех секций, содержащих программный код
5	SizeOfInitializedData	4	Размер всех секций, содержащих инициализированные данные
6	SizeOfUninitializedData	4	Размер всех секций, содержащих неинициализированные данные
7	AddressOfEntryPoint	4	RVA точки запуска программы
8	BaseOfCode	4	RVA начала кода программы

№	Имя поля	Размер (PE32/ PE32+)	Описание
9	BaseOfData	4/0	RVA начала данных программы. В PE32+ отсутствует
10	ImageBase	4/8	Предпочтительный базовый адрес программы в памяти, кратный 64 Кб. Загрузка программы с этого адреса позволяет обойтись без настройки адресов
11	SectionAlignment	4	Выравнивание в байтах для секций при загрузке в память, большее или равное FileAlignment. По умолчанию равно размеру страницы виртуальной памяти для данного процессора
12	FileAlignment	4	Выравнивание в байтах для секций внутри файла. Должно быть степенью 2 от 512 до 64 Кб включительно. По умолчанию равно 512. Если SectionAlignment меньше размера страницы виртуальной памяти, то FileAlignment должно с ним совпадать
13	MajorOperatingSystemVersion	2	Старшая цифра номера версии операционной системы
14	MinorOperatingSystemVersion	2	Младшая цифра номера версии операционной системы
15	MajorImageVersion	2	Старшая цифра номера версии данного файла
16	MinorImageVersion	2	Младшая цифра номера версии данного файла
17	MajorSubsystemVersion	2	Старшая цифра номера версии подсистемы
18	MinorSubsystemVersion	2	Младшая цифра номера версии подсистемы
19	Win32VersionValue	0	Зарезервировано, всегда равно 0
20	SizeOfImage	4	Размер файла в памяти, включая все заголовки. Должен быть кратен SectionAlignment
21	SizeOfHeaders	4	Размер всех заголовков выравненный на FileAlignment. Задает смещение от начала файла до данных первой секции
22	Checksum	4	Контрольная сумма файла
23	Subsystem	2	Исполняющая подсистема Windows для данного файла
24	DllCharacteristics	2	Дополнительные атрибуты файла
25	SizeOfStackReserve	4/8	Размер стека стартового потока программы в байтах виртуальной памяти. При загрузке в физическую память отображается только SizeOfStackCommit байт, в дальнейшем отображается по одной странице виртуальной памяти. По умолчанию равен 1 Мб
26	SizeOfStackCommit	4/8	Начальный размер стека программы в байтах. По умолчанию равен 4 Кб

№	Имя поля	Размер (PE32/ PE32+)	Описание
27	SizeOfHeapReserve	4/8	Размер кучи программы в байтах. При загрузке в физическую память отображается только SizeOfHeapCommit байт, в дальнейшем отображается по одной странице виртуальной памяти. По умолчанию равен 1 Мб. Во всех 32-разрядных версиях Windows куча ограничена только размером виртуальной памяти и это поле, по-видимому, игнорируется
28	SizeOfHeapCommit	4/8	Начальный размер кучи программы в байтах. По умолчанию равен 4 Кб
29	LoaderFlags	4	Не используется
30	NumberOfRvaAndSizes	4	Количество описателей каталогов данных. На текущий момент всегда равно 16
31	DataDirectory	128	Описатели каталогов данных

В табл. 17 приведен список каталогов данных. В конце необязательного заголовка располагается 32-битовое число, в котором хранится количество описателей каталогов данных. За ним следует массив самих описателей, каждый из которых имеет такой вид:

```
struct IMAGE_DATA_DIRECTORY
{
    DWORD VirtualAddress;
    DWORD Size;
};
```

Таблица 17

Список каталогов данных

№	Название	Описание
0	IMAGE_DIRECTORY_ENTRY_EXPORT	Таблица экспорта
1	IMAGE_DIRECTORY_ENTRY_IMPORT	Таблица импорта
2	IMAGE_DIRECTORY_ENTRY_RESOURCE	Таблица ресурсов
3	IMAGE_DIRECTORY_ENTRY_EXCEPTION	Таблица обработки исключений
4	IMAGE_DIRECTORY_ENTRY_SECURITY	Таблица сертификатов безопасности
5	IMAGE_DIRECTORY_ENTRY_BASERELOC	Таблица настроек адресов
6	IMAGE_DIRECTORY_ENTRY_DEBUG	Отладочная информация
7	IMAGE_DIRECTORY_ENTRY_ARCHITECTURE	Данные, специфичные для процессора

№	Название	Описание
8	IMAGE_DIRECTORY_ENTRY_GLOBALPTR	RVA глобального регистра процессора
9	IMAGE_DIRECTORY_ENTRY_TLS	Таблица локальной памяти потоков (TLS)
10	IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	Таблица конфигурации загрузки
11	IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	Таблица связывания импорта
12	IMAGE_DIRECTORY_ENTRY_IAT	Таблица адресов импорта (IAT)
13	IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	Таблица отложенного импорта
14	IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	Дескриптор .NET
15	—	Зарезервировано

Таблица экспорта всегда присутствует в DLL-файлах, поскольку основным назначением динамических библиотек является экспорт символов, доступных другим исполнимым файлам. Данная таблица связывает имена и/или номера экспортируемых функций с их RVA, т. е. с положением в виртуальной памяти процесса.

Таблица импорта присутствует практически во всех исполнимых файлах и является массивом описателей IMAGE_IMPORT_DESCRIPTOR, в котором последовательно хранятся имена функций какой-либо DLL. На каждую импортируемую DLL приходится по одному описателю. Массив заканчивается описателем, который полностью заполнен нулями. Каждый из описателей имеет структуру, представленную в табл. 18.

Таблица 18

IMAGE_IMPORT_DESCRIPTOR

Название	Размер	Описание
OriginalFirstThunk	4	RVA таблицы имен импорта (INT)
TimeDateStamp	4	Дата и время.
ForwarderChain	4	Индекс первого перенаправленного символа
Name	4	RVA ASCIIZ-строки, содержащей имя DLL
FirstThunk	4	RVA таблицы адресов импорта (IAT)

Таблица имен (INT) и таблица адресов IAT представляют собой массивы структур IMAGE_THUNK_DATA. Если старший бит данного слова установлен, то остальные 31 или 63 бита содержат номер импортируемого символа (импорт по номеру). Если же этот бит сброшен, то остальные биты задают RVA описателя импортируемого символа (импорт по имени). Импорт может быть осуществлен следующими тремя способами.

Стандартный. С помощью самой таблицы импорта. Таблица импорта хранит имена функций/ординалов и указывать, в какое место загрузчик

должен записать эффективный адрес этой функций. Этот механизм не очень эффективен, так как сводится к перебору всей таблицы экспорта для каждой необходимой функции.

Связывающий (bound import). При данной схеме работы в поля (в первом элементе стандартной таблицы импорта) TimeDateStamp и ForwardChain заносится -1 и информация о связывании хранится в IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT. Для bound импорта также существуют свои структуры. Алгоритм работы заключается в следующем – в виртуальную память приложения выгружается необходимая библиотека и все необходимые адреса привязываются на этапе компиляции. Но при перекомпиляции dll, необходимо перекомпилировать само приложение, так как адреса функций будут изменены.

Отложенный (delay import). При данном методе подразумевается что .dll файл прикреплен к исполняемому файлу, но в память выгружается не сразу (как в предыдущих двух методах), а только при первом обращении приложения к символу. Программа выполняется и, как только процессу становится необходим вызов функции из динамической библиотеки, то вызывается специальный обработчик, который подгружает dll и разносит эффективные адреса ее функций. За отложенным импортом загрузчик обращается к каталогу IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT.

Сразу за массивом DataDirectory расположены секции [47]. Таблица секций (табл. 19) имеет NumberOfSections записей. Каждая запись имеет размер в 0x28 байт.

Таблица 19

Таблица секций DataDirectory

Название	Размер	Описание
Name	8	Название секции
Misc.VirtualSize	4	Размер секции в памяти
VirtualAddress	4	RVA секции в памяти
SizeOfRawData	4	Размер секции в файле. Всегда кратен FileAlignment
PointerToRawData	4	Смещение в файле до начала данных секций. Всегда кратно FileAlignment
PointerToRelocations	4	В исполняемых файлах это поле всегда равно нулю
PointerToLinenumbers	4	В исполняемых файлах это поле всегда равно нулю
NumberOfRelocations	2	В исполняемых файлах это поле всегда равно нулю
NumberOfLinenumbers	2	В исполняемых файлах это поле всегда равно нулю
Characteristics	4	Атрибуты доступа к секции и правила для ее загрузки в виртуальную память. Например, атрибут для определения содержимого секции (инициализированные или неинициализированные данные, код). Или атрибуты доступа – чтение, запись, исполнение

Секция с ресурсами всегда должна иметь имя «.rsrc». В противном случае ресурсы не будут подгружены. Что касается остальных секций – то имя может быть любым. Обычно встречаются осмысленные имена, например .data, .src и т. д. Секции – это такая область, которая выгружается в виртуальную память и вся работа происходит непосредственно с этими данными. Адрес в виртуальной памяти, без всяких смещений называется Virtual address (далее VA). Предпочитаемый адрес для загрузки приложения задается в поле ImageBase. Это как точка, с которой начинается область приложения в виртуальной памяти. И относительно этой точки отсчитываются смещения RVA (Relative virtual address). То есть $VA = ImageBase + RVA$; ImageBase нам всегда известно и узнав VA или RVA, мы можем выразить одно через другое.

Представление PE-файла в памяти называется его виртуальным образом или просто образом, а на диске – файлом или дисковым образом. Образ характеризуется адресом базовой загрузки (image base) и размером (image size). При наличии перемещаемой информации (relocation/fixup table), образ может быть загружен по адресу, отличному от image base и назначаемому непосредственно самой операционной системой. Образ естественным образом делится на страницы (pages), а файл – на сектора (sectors). Виртуальный размер страниц/секторов явным образом задается в заголовке файла и не обязательно должен совпадать с физическим. Для работы с PE-файлами используются три различных схемы адресации.

Физические адреса (называемые также сырыми указателями или смещениями raw pointers/raw offset или просто offset), отсчитываемые от начала файла.

Виртуальные адреса (virtual address или сокращенное VA), отсчитываемые от начала адресного пространства процесса.

Относительные виртуальные адреса (relative virtual address или сокращенно RVA), отсчитываемые от базового адреса загрузки.

Все типы адресов хранятся в 32-битных указателях и измеряются в байтах. Страничный имидж состоит из одной или нескольких секций. С каждой секцией связано четыре атрибута: физический адрес начала секции в файле/размер секции в файле, виртуальный адрес секции в памяти/размер секции в памяти, атрибут характеристик секции, описывающий права доступа, особенности ее обработки системным загрузчиком и т. д.

Начало каждой секции в памяти/на диске всегда совпадает с началом виртуальных страниц/секторов соответственно. Попытка создать секцию, начинающуюся с середины, пресекается системным загрузчиком, отказывающимся обрабатывать такой файл. С концом складывается более демократичная ситуация и загрузчик не требует, чтобы виртуальный (и частично физический) размер секций был кратен размеру страницы. Вместо этого

он самостоятельно выравнивает секции, забивая их хвост нулями, так что никакая страница (сектор) не может принадлежать двум и более секциям сразу. Все секции совершенно равноправны и тип каждой из них тесно связан с ее атрибутами, интерпретируемыми довольно неоднозначным и противоречивым образом. Служебные структуры данных (таблицы экспорта, импорта, перемещаемых элементов), могут быть расположены в любой секции с подходящими атрибутами доступа. Когда-то правила хорошего тона диктовали помещать каждую таблицу в свою персональную секцию, но теперь эта методика признана устаревшей, что существенно утяжеляет алгоритм внедрения в исполняемый файл.

8. МЕТОДЫ ЗАЩИТЫ ПРИЛОЖЕНИЙ ОТ АНАЛИЗА И ВЗЛОМА

Взлом программ осуществляется двумя основными способами.

Отладка (пошаговое исполнение) – позволяет произвести пошаговое исполнение любого исполнимого файла. Является единственным способом для разработчика узнать, почему его код работает неправильно.

Дизассемблирование – способ преобразования исполняемых модулей в язык программирования, понятный человеку – Ассемблер. В этом случае становится наглядно видно, что именно делает приложение.

Рассмотрим основные методы противодействия взлому.

Протекторы и упаковщики. Упаковка исполняемых файлов заключается в сжатии исполняемого файла и прикреплении к нему кода, необходимого для распаковки и выполнения содержимого файла. Упакованный файл занимает меньше места на носителе информации, что помогает ускорить его загрузку в память. Некоторые виды упаковки совмещены с шифрованием содержимого файла для предотвращения обратной разработки программы (протекторы). Упаковка с шифрованием может использоваться и злонамеренно при создании вирусов, чтобы зашифровать и видоизменить код вируса для затруднения его обнаружения системами, основанными на сигнатурах. Наиболее популярные сегодня упаковщики – *UPX*, *VMProtect*, *ASPack*, *PeShiel*. Общий алгоритм работы упаковщика принимает следующий вид.

1. Сохранение оригинальной точки входа EP.
2. Сжатие содержимого файла (обычно, это секция кода и данных), используя алгоритмы архивирования.
3. Запись своей сигнатуры после либо до упакованного кода программы.
4. Перенаправление ее не в основной код программы, а в код распаковщика.
5. Код распаковщика, получает управление первым и распаковывает упакованные секции в памяти. На диске исходный файл остается без изменений.

6. После того как код и данные программы распакованы, код распаковщика восстанавливает таблицу импорта и передает управление основному коду программы, на оригинальную точку входа.

Протекторы, в отличие от упаковщиков, призваны защитить исходный файл от обратной разработки, соответственно, при этом они используют более изощренные методы: *шифрование*, *обфускацию*, *встраивание антиотладочных приемов*.

Шифрование. Самый простой и эффективный способ противодействия. Подразумевает, что определенная часть кода никогда не появляется

в свободном виде. Код дешифруется только перед передачей ему управления. То есть вся программа или ее часть находится в зашифрованном виде, а расшифровывается только перед тем как исполниться. Соответственно, чтобы проанализировать ее код, надо воспользоваться отладчиком, а его работу можно очень и очень усложнить, например, с помощью обфускации кода.

Шифрование и дешифрование (динамическое изменение кода). Более продвинутый способ шифрования, который не просто дешифрует часть кода при исполнении, но и шифрует его обратно, как только он был исполнен.

Использование виртуальных машин. Еще одна модернизация шифрования. Способ заключается в том, чтобы не просто шифровать и дешифровать целые фрагменты кода целиком, а делать это покомандно, подобно тому, как действует отладчик или виртуальная машина: взять код, преобразовать в машинный и передать на исполнение, и так пока весь модуль не будет исполнен. Этот способ гораздо эффективнее предыдущих, так как функции приложения вообще никогда не бывают открытыми для хакера. Естественно, что его трудно реализовать, но реализовав, можно оградить себя от посягательств любых хакеров. В этом способе кроется также и недостаток – снижение производительности, ведь на подобное транслирование требуется много времени, и, соответственно, способ хорош для защиты только критических участков кода.

Обфускация кода программы. Обфускацией (от английского obfuscation – буквально «запутывание») называется совокупность методик и средств, направленных на затруднение анализа программного кода. Выделяют следующие виды обфускации:

- 1) лексическая обфускация;
- 2) преобразование данных;
- 3) преобразование управления;
- 4) профилактическая обфускация.

Лексическая обфускация заключается в изменении названий функций и переменных. Например, переменная, с интуитивно понятным именем – array_name[number], может быть изменена на prrq[feh].

Преобразование данных включает в себя изменение и создание новых типов данных и применение к ним комбинаторики. Например, число 5 можно представить, как 1000001 (количество нулей), 35 (вторая цифра) и еще бесконечным количеством способов.

Профилактическая обфускация защищает код от деобфускации. Деобфускация основывается на обнаружении неиспользуемых кусков кода, нахождении наиболее сложных структур и анализе статистических и динамических данных. Именно борьба с этими операциями – наиболее сложный и эффективный процесс обфускации.

Преобразование управления заключается в нарушении естественного хода программы. Результат исполняемых действий трудно предугадать в ходе заданной процедуры. Возможно создание дополнительных блоков кода: в одном выполняются вычисления, в другом происходит присваивание. В более сложных ситуациях создается таблица преобразований и замещений, полностью изменяющих общую структуру кода. В простейшем случае исполнимый модуль наполняется мусорными командами (например: `por, xchg reg,reg, or reg,reg`) и никогда не выполняющимися переходами.

Например:

<i>or bl, bl</i>	;незначащая команда, регистр bl не ;изменяется, изменяется регистр ;флагов, но это изменение отменяется ;следующим хог
<i>xor eax eax</i>	;потенциально значащая команда
<i>seto cl</i>	;незначащая команда, ;устанавливающая cl в 1, если ;установлен флаг переполнения, но ;после хог его всегда нет
<i>jnz short loc_4567A</i>	;незначащая команда, передающая ;управление, если не установлен ;флаг нуля, но после хог он всегда ;установлен
<i>repne jnp short loc_4569F</i>	;незначащая команда, передающая ;управление, если не установлен ;флаг четности, но после хог он ;всегда установлен, кроме того ;присутствует бессмысленный префикс ;repne
<i>xchg ecx,ecx</i>	;незначащая команда, обмен ;регистра ecx местами

Более сложные методики обфускации используют «перемешивание кода» (рис. 38), закручивая поток управления в запутанную спираль условных/безусловных переходов.

Рис. 38. Запутывание кода

Такие обфускаторы используют технику «перекрытия» команд: некоторые байты привязываются к двум–пяти машинным инструкциям. В этом случае дизассемблер генерирует неполный и неправильный листинг.

Например:

```
.adata:0043400E loc_43400E:                ;CODE XREF .adata:00434023
.adata:0043400E                            ;.adata:loc_43401A
.adata:0043400E    mov eax, 0ffh
.adata:00434013
.adata:00434013 loc_434013:                ;CODE XREF .adata:0043401D
.adata:00434013    seto bl                    ;прыжок в середину инструкции
.adata:00434016    or bh,ch
.adata:00434018    jmp short loc_434025
.adata:00434018
.adata:0043401A loc_43401A:
.adata:0043401A    xor ecx,ecx
.adata:0043401D
.adata:0043401D loc_43401D:                ;CODE XREF .adata: loc_43400C
.adata:0043401D    jmp short near ptr loc_434013+2
```

Команда "jmp short loc_434013+2", осуществляет переход по адресу 434013h+2h == 434015h, т. е. в середину инструкции seto bl. С точки зрения дизассемблера команда является неделимой структурной единицей. Но в реальности машинная инструкция является последовательностью байт и может быть выполнена с любого места! Если начать выполнение инструкции не с первого байта мы получим совсем другую команду. Чтобы выполнить переход "jmp short loc_434013+2" мы должны подвести курсор к метке loc_434013 и нажать «U», чтобы «раскрошить» дизассемблерный код на байты, а после перейти по адресу 434015h и нажать «C», чтобы превратить байты в дизассемблерный код. В результате чего получится следующее:

```
.adata:0043400E unk_43400E:db 0b8h            ;CODE XREF .adata:00434023
.adata:0043400F    db 0ebh                    ;.adata:loc_43401A
.adata:00434010    db 7
.adata:00434011    db 0b9h
.adata:00434012 loc_434012:                ;CODE XREF .adata:0043401A
.adata:00434012    jmp short loc_434025
.adata:00434014
.adata:00434014    nop
.adata:00434015
.adata:00434015 loc_434015:                ;CODE XREF
.adata:00434015                            .adata:loc_43401D
.adata:00434015    jmp short loc_43401F        ;переход сюда
```

```

.adata:00434017
.adata:00434017      std

.adata:00434018      jmp short loc_434025
.adata:0043401A
.adata:0043401A loc_43401A:                                ;CODE XREF
                                                            .adata:loc_434009
.adata:0043401A      repne jmp short loc_434012

```

Антиотладочные приемы. Как понятно из названия, применение данных приемов используют для обеспечения невозможности (или точнее сказать – затруднения) отладки приложения.

Функция *BOOL WINAPI IsDebuggerPresent()* – *kernel32.dll*. Данная функция выясняет, находится ли программа под отладчиком. Функция возвращает *EAX = 1* – если процесс запущен под отладчиком, *EAX = 0* – в противном случае. Пример кода:

```

CALL IsDebuggerPresent
TEST al, al                ; 1 – процесс отлаживается
JNE being_debugged

```

Проверка байта *PEB.BeingDebugged*. *PEB* – структура процесса в Windows, которая заполняется загрузчиком на этапе создания процесса, и которая содержит информацию об окружении, загруженных модулях, базовой информации по текущему модулю и другие критичные данные необходимые для функционирования процесса. *PEB.BeingDebugged* – 2-й байт в структуре *Process Environment Block* процесса. Он устанавливается в 1 самой системой, когда процесс запущен под отладчиком. Это чисто информационный флажок, поэтому смело можно сбрасывать в 0 без ущерба для программы. Пример кода:

```

mov  eax,dword ptr fs:[30h]
movzx eax,byte ptr [eax+2]
ret

```

Заполнение мусором массива *Data Directories*. Если каталог не используется, то в качестве меры по антиотладке, можно указать в полях (*RVA* адрес и размер) элемента любой мусор.

***DebugService*.** Выполнение этого прерывания, если программа не отлаживается, генерирует исключение точки останова (*int 3*). Если программа запущена под отладчиком, и выполняется со сброшенным флагом трассировки, то никакого исключения не произойдет, и выполнение будет продолжаться в обычном режиме. Если программа отлаживается, и команда трассируется, то следующий байт будет пропущен, и выполнение продолжится через байт.

Тайминговые атаки. Очень эффективны против отладчиков. Когда отладчик присутствует, и выполняет пошаговую трассировку, есть существенная задержка между выполнением отдельных команд по сравнению с обычным выполнением. Команда RDTSC (Read Time-Stamp Counter) для платформы x86, читает счетчик TSC (Time Stamp Counter) и возвращает в регистрах EDX:EAX 64-битное количество тактов с момента последнего сброса процессора. Для того, чтобы использовать инструкцию RDTSC в антиотладочных целях, необходимо выполнить ее дважды: до и после выполнения кода, для которого будет производиться замер. Пример кода:

```
rdtsc
xchg    ecx, eax
rdtsc
sub     eax, ecx
cmp     eax, 500h
jbe     __нет отладчика__ ; количество тактов меньше 500h – отлад-
чика нет
__      отладчик есть__
```

Изменение размера образа. Дамп – содержимое рабочей памяти процесса, ядра или всей операционной системы в определенный момент времени. Дамп помогает против зашифрованных программ, ибо «сфотографировать» память можно в любой момент, в то время, как перед тем как инструкция выполнится она должна быть расшифрована. SizeOfImage – размер образа, показывает (в байтах), какое количество памяти должно быть выделено в адресном пространстве для загрузки образа исполняемого файла. Это сумма длин всех секций бинарного (исполняемого байта). Определяется по формуле: SizeOfImage = VirtualOffset последней секции + VirtualSize последней секции. Информация о размере модуля храниться в PEВ структуре (а также поле SizeOfImage можно найти в PE-заголовке файла). Такой прием препятствует доступу к процессу, мешает отладчику присоединиться к процессу. Однако легко обойти эту уловку можно даже из пользовательского режима. Просто игнорируем значение SizeOfImage, и вместо этого вызываем функцию VirtualQuery(). Она возвращает число последовательных страниц виртуального адресного пространства с одинаковыми атрибутами. В памяти не может быть промежутка между секциями, и страницы памяти могут быть подсчитаны, начиная с первой страницы после конца предыдущего диапазона. Подсчет начинается со страницы ImageBase (базовый адрес образа) и продолжается, пока возвращается тип MEM_IMAGE. Страница, которая не имеет тип MEM_IMAGE, не относится к файлу.

9. ПРАКТИЧЕСКИЕ РАБОТЫ ПО РЕВЕРСУ

Практическая работа 1. Динамический анализ кода исполняемого файла на примере CrackMe

Цель работы

1. Ознакомиться с основными принципами динамического анализа кода.
2. Освоить методику использования дизАссемблера IDA PRO.

Задачи

1. Произвести исследование предлагаемого исполняемого файла.
2. Получить правильный серийный номер.

Требуемое ПО: дизассемблер IDA PRO.

Алгоритм работы

1. Запускаем исследуемое приложение. Вбиваем случайные данные в окно регистрации (рис. 39, 40).

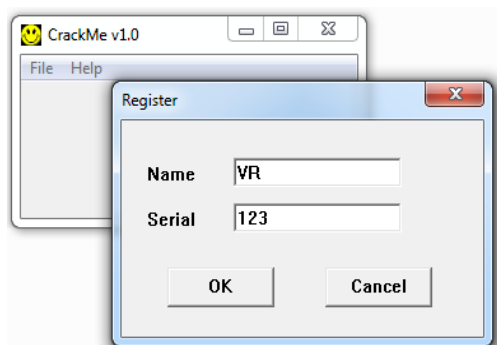


Рис. 39. Первичный запуск исследуемого приложения

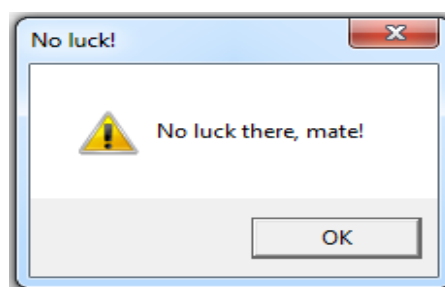


Рис. 40. Сообщение о неудачном вводе

2. Загружаем приложение в IDA. Смотрим строки, которые здесь используются (рис. 41) (**View->Open subviews->Strings**).

Address	Length	Type	String
DATA:00402...	00000011	C	Try to crack me!
DATA:00402...	0000000D	C	CrackMe v1.0
DATA:00402...	0000001C	C	No need to disasm the code!
DATA:00402...	00000005	C	MENU
DATA:00402...	0000000A	C	DLG_REGIS
DATA:00402...	0000000A	C	DLG_ABOUT
DATA:00402...	0000000B	C	Good work!
DATA:00402...	0000002C	C	Great work, mate! \r\nNow try the next CrackMe!
DATA:00402...	00000009	C	No luck!
DATA:00402...	00000015	C	No luck there, mate!

Рис. 41. Строки, используемые в модуле

Нас интересуют две строки:

– Great work, mate!\rNow try the next CrackMe!

– No luck there, mate!

3. Смотрим, в какой функции используется любая из этих строк (рис. 42).

```
-----  
sub_401340 proc near ; CODE XREF: WndProc:loc_40124C↑p  
push 30h ; uType  
push offset Caption ; "Good work!"  
push offset Text ; "Great work, mate!\rNow try the next Cra"...  
push dword ptr [ebp+8] ; hWnd  
call MessageBoxA  
retn  
sub_401340 endp  
  
; ===== S U B R O U T I N E =====  
  
sub_401362 proc near ; CODE XREF: WndProc+11D↑p  
push 0 ; uType  
call MessageBeep  
push 30h ; uType  
push offset aNoLuck ; "No luck!"  
push offset aNoLuckThereMat ; "No luck there, mate!"  
push dword ptr [ebp+8] ; hWnd  
call MessageBoxA  
retn  
sub_401362 endp
```

Рис. 42. Листинг кода,
в котором используются найденные строки

4. Находим место принятия решения о том, правильно ли был введен серийный номер. В данном случае, если регистр eax равен регистру ebx, то осуществляется переход на функцию, вызывающую MessageBox со строкой об успешном вводе. В обратном случае вызывается MessageBox со строкой, сигнализирующей о неправильном вводе (рис. 43).

```
-----  
CODE:00401209 loc_401209: ; CODE XREF: WndProc+BA↑j  
CODE:00401209 push 0 ; dwInitParam  
CODE:00401209 push offset sub_401253 ; lpDialogFunc  
CODE:00401210 push [ebp+hWnd] ; hWndParent  
CODE:00401213 push offset aDlgRegis ; "DLG_REGIS"  
CODE:00401218 push ds:hInstance ; hInstance  
CODE:0040121E call DialogBoxParamA  
CODE:00401223 cmp eax, 0  
CODE:00401226 jz short loc_4011E6  
CODE:00401228 push offset String  
CODE:0040122D call sub_40137E  
CODE:00401232 push eax  
CODE:00401233 push offset byte_40217E  
CODE:00401238 call sub_4013D8  
CODE:0040123D add esp, 4  
CODE:00401240 pop eax  
CODE:00401241 cmp eax, ebx  
CODE:00401243 jz short loc_40124C  
CODE:00401245 call sub_401362  
CODE:0040124A jmp short loc_4011E6  
CODE:0040124C ;  
CODE:0040124C loc_40124C: ; CODE XREF: WndProc+11B↑j  
CODE:0040124C call sub_401340  
CODE:00401251 jmp short loc_4011E6  
CODE:00401251 WndProc endp  
CODE:00401253 ; ===== S U B R O U T I N E =====  
CONF:00401253
```

Рис. 43. Место принятия решения
о правильности серийного номера

5. Определяем места загрузки значений в регистры `eax` и `ebx`. Не трудно заметить, что выше по коду вызываются две функции:

- `sub_40137E`,
- `sub_4013D8`.

После вызова функции `sub_40137E` (рис. 44) на стеке сохраняется значение регистра `eax`, а перед сравнением с регистром `ebx`, значение `eax` со стека забирается. Предполагаем, что в функции `sub_4013D8` загружается регистр `ebx`.

6. Рассмотрим функцию `sub_40137E`:

1) алгоритм посимвольно обрабатывает строку, пока не встретит NULL-byte;

2) символ должен иметь аски-код не меньше чем `0x41` = 'A';

3) если он больше или равен `0x5A` = 'Z', то из него вычитается `0x20`;

4) после проверки «валидности», коды всех символов складываются в `DWORD`;

5) и его (0 или 1) «XOR'ят» (производят дизъюнкцию) с `0x5678`, это есть своеобразный «хеш» от `UserName`.

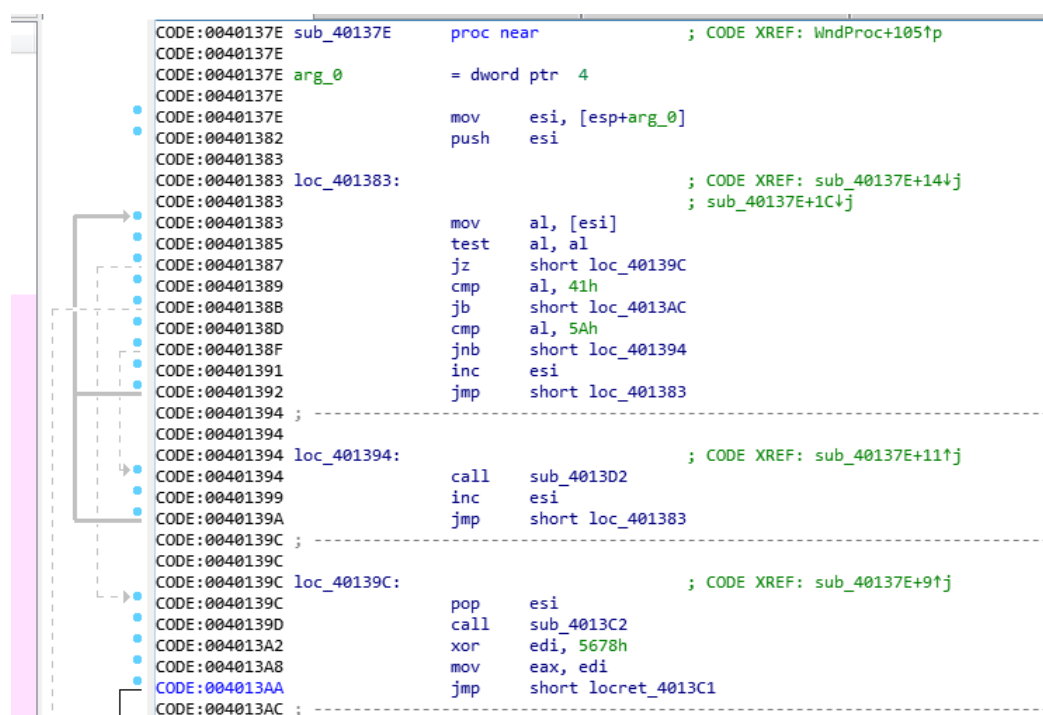


Рис. 44. Листинг функции `sub_40137E`

7. Рассмотрим функцию `sub_4013D8` (рис. 45):

1) строка обрабатывается ровно до NULL-byte;

2) из каждого символа вычитается `0x30` = '0';

3) значение `EDI` умножается на `0xA` = 10;

4) к нему прибавляется аски-код текущего символа;

5) после конца строки `EDI` «XOR'ят» на `0x1234`, это есть своеобразный «хеш» от `PassWord`.

```

CODE:004013D8 sub_4013D8 proc near ; CODE XREF: WndProc+110↑p
CODE:004013D8
CODE:004013D8 arg_0 = dword ptr 4
CODE:004013D8
CODE:004013D8 xor eax, eax
CODE:004013DA xor edi, edi
CODE:004013DC xor ebx, ebx
CODE:004013DE mov esi, [esp+arg_0]
CODE:004013E2
CODE:004013E2 loc_4013E2: mov al, 0Ah ; CODE XREF: sub_4013D8+1B↑j
CODE:004013E2
CODE:004013E4 mov bl, [esi]
CODE:004013E6 test bl, bl
CODE:004013E8 jz short loc_4013F5
CODE:004013EA sub bl, 30h
CODE:004013ED imul edi, eax
CODE:004013F0 add edi, ebx
CODE:004013F2 inc esi
CODE:004013F3 jmp short loc_4013E2
CODE:004013F5 ;
CODE:004013F5
CODE:004013F5 loc_4013F5: xor edi, 1234h ; CODE XREF: sub_4013D8+10↑j
CODE:004013F5
CODE:004013FB mov ebx, edi
CODE:004013FD retn
CODE:004013FD sub_4013D8 endp
CODE:004013FD

```

Рис. 45. Листинг функции sub_4013D8

8. Для успешного прохождения CrackMe значения обоих «хешей» должны быть равны. Введем имя **‘VR’** и получим в регистре EAX = 0x56D0. Сразу избавимся от лишнего «XOR’а»: 0x56D0 XOR 0x1234 = **0x44E4**, это значение и требуется подобрать. Одним из возможных вариантов может быть **-17636**. Проверяем и убеждаемся в этом (рис. 46).

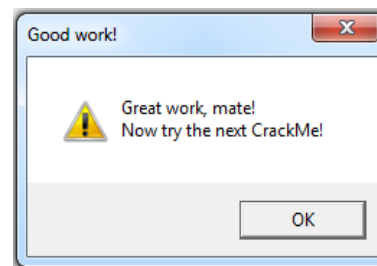


Рис. 46. Сообщение об успешном вводе серийного номера

Практическая работа 2. Ручная распаковка вирусного приложения

Цель работы: на примере предложенного вируса разобрать базовый алгоритм первичного анализа файла, поиска OEP, оригинального кода.

Требуемое ПО:

- 1) PEiD;
- 2) OllyDbg;
- 3) IDA Pro.

Алгоритм работы

1. Создаем виртуальную машину, на которой будем исследовать предложенный вирус. Из виртуальных сред наиболее доступны VirtualBox, MS Hyper-V и QEMU.
2. Загружаем вирус в анализатор PEiD (рис. 47).

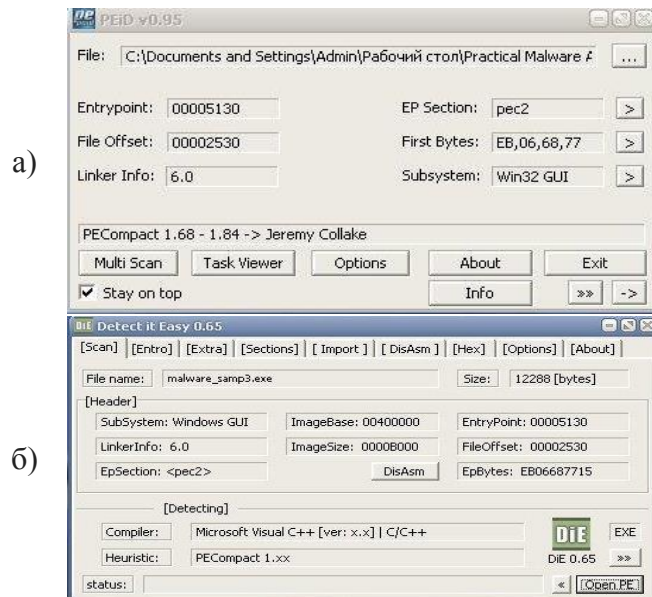


Рис. 47. Результат анализа PEiD:

а) с загрузкой вируса; б) с отображением характеристик вируса

Результат анализа:

- 1) файл представляет собой Win32-приложение;
- 2) бинарный файл был скомпилирован Visual C++;
- 3) файл упакован PECompact.

3. Загружаем вирус в OllyDbg (рис. 48).

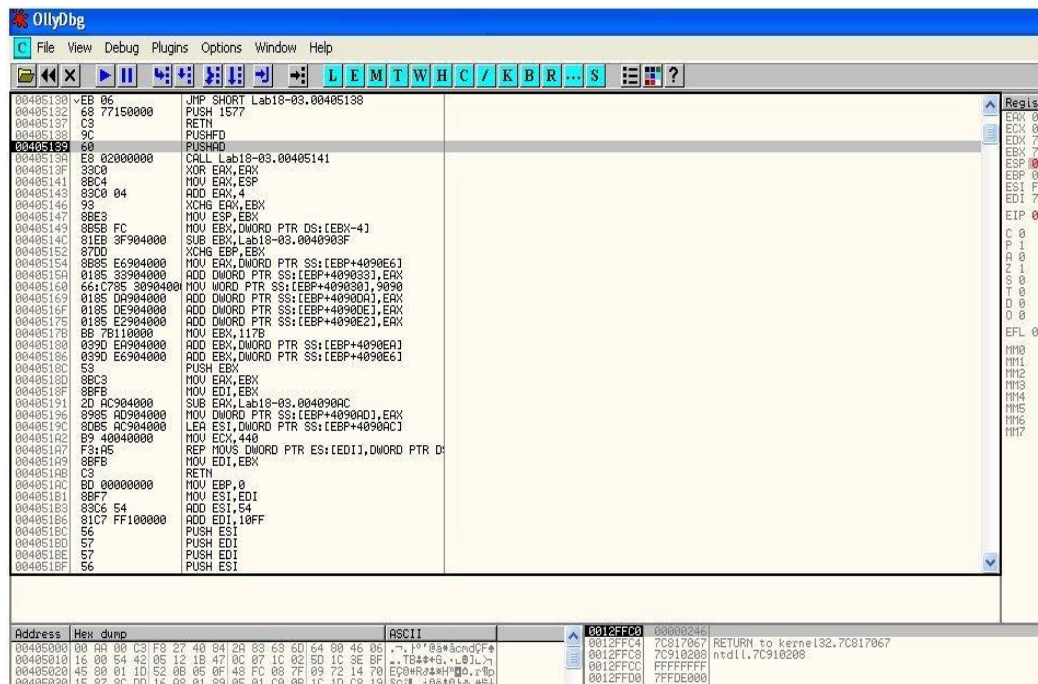


Рис. 48. Окно OllyDbg после загрузки вируса

Курсор встает на адресе 00405139 PUSHAD. Справа в окне можем посмотреть текущее значение регистров. Дальше ставим точку останова hardware on access на регистре ESP. Жмем несколько раз F9 (рис. 49), чтобы запустить программу, после чего она наткнется на нашу точку останова. Курсор остановился на адресе 0045013A CALL malware01.00405141, соответственно, это главный CALL.

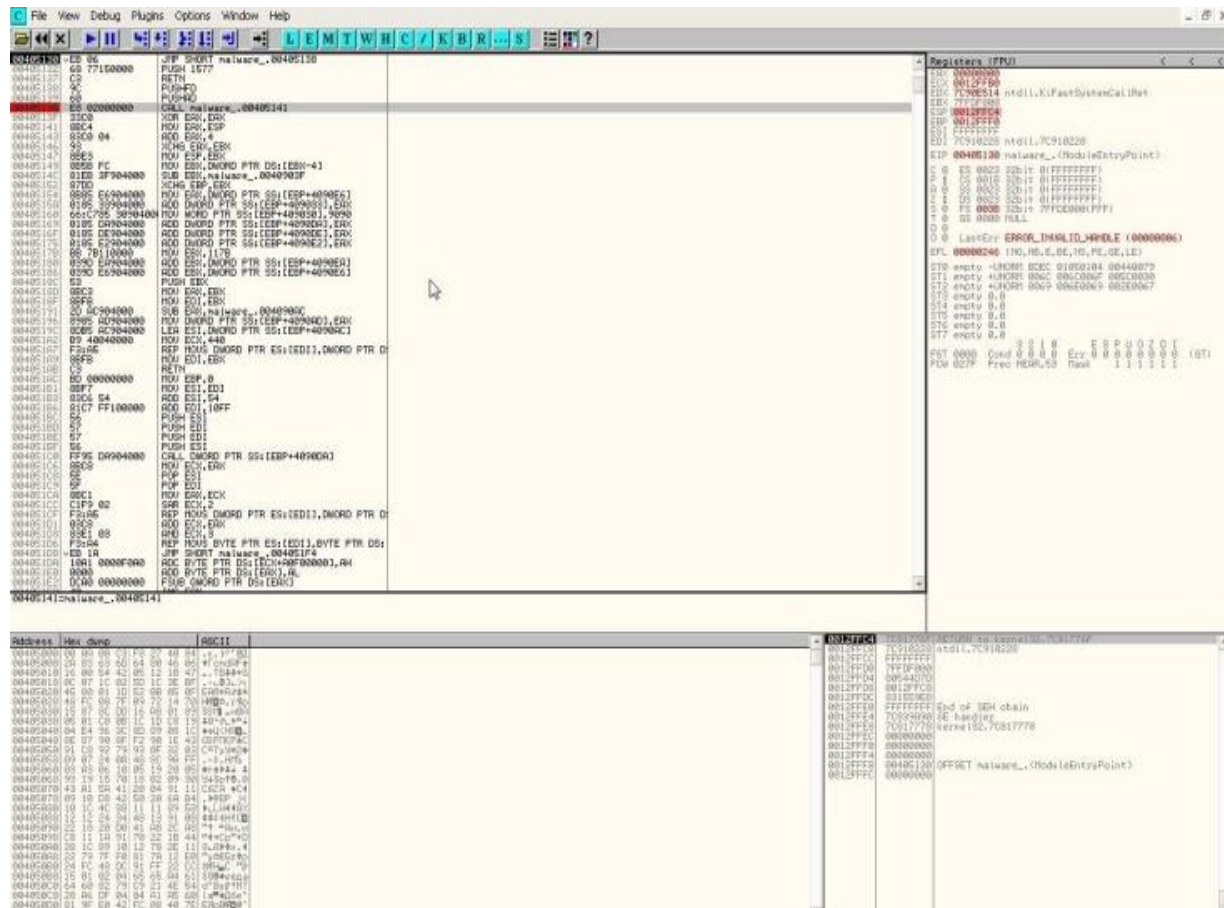


Рис. 49. OllyDbg после установки точки останова

В правом окне регистров на значении ESP 0012FfA0 щелкаем правой клавишей и выбираем Follow Dump. Далее переключаемся в нижнее окно, где содержится Hex dump, и, выделив несколько элементов, также щелкаем правой клавишей на Breakpoint → memory on access. Запускаем выполнение программы F9. Ставим еще одну точку останова: Breakpoint → Hardware on access → Dword. Далее выполнение кода останавливается на точке POPAD (рис. 50).

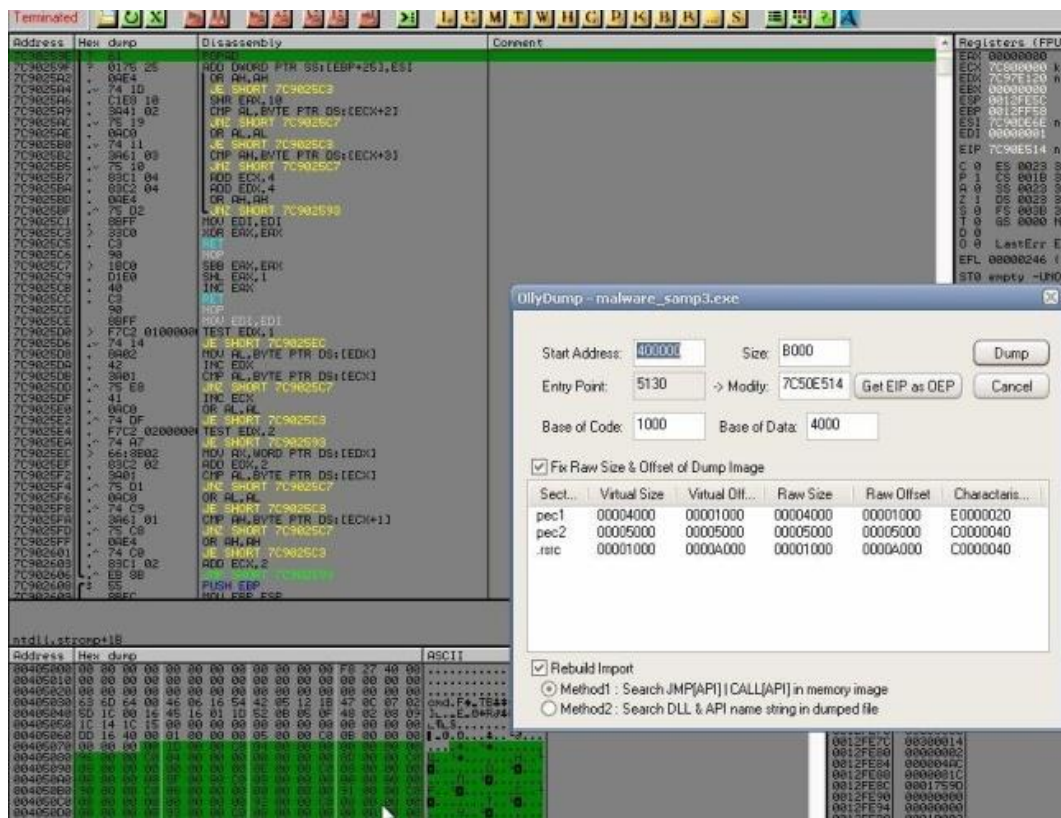


Рис. 50. Окно OllyDbg в точке останова POPAD

Если вернуться на шаг назад, то мы увидим распакованный оригинальный код, однако он будет в нечитаемом для нас виде (рис. 51).

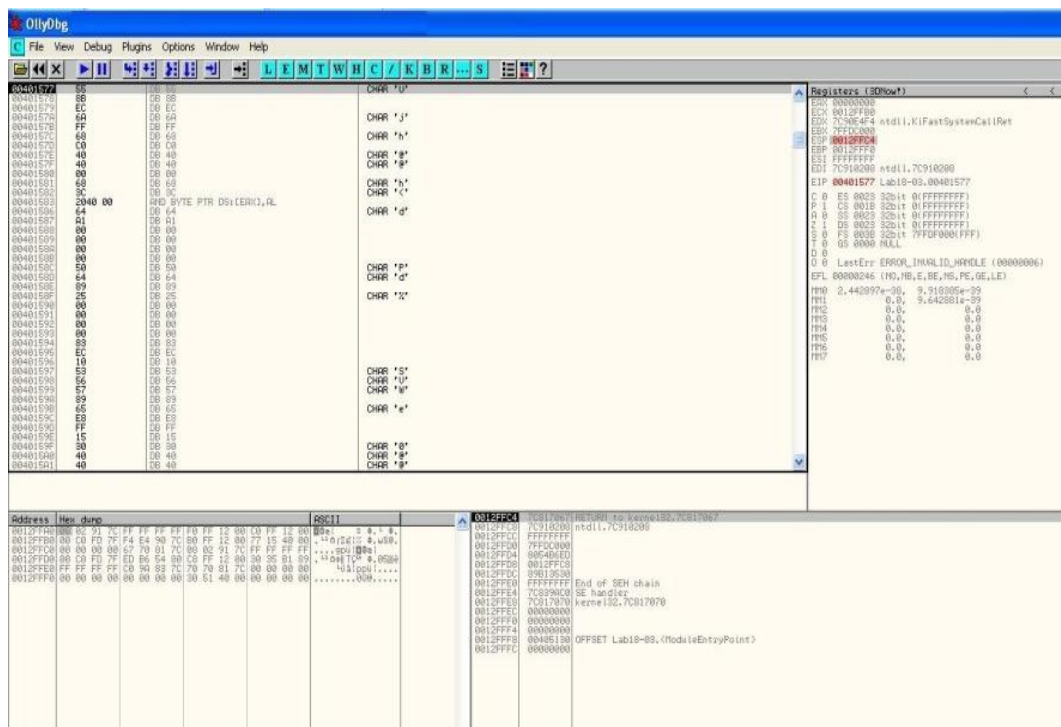


Рис. 51. Окно OllyDbg с упакованным кодом

Чтобы это исправить, жмем Ctrl + A и видим, как строки преобразуются (рис. 52) в понятный набор инструкций.

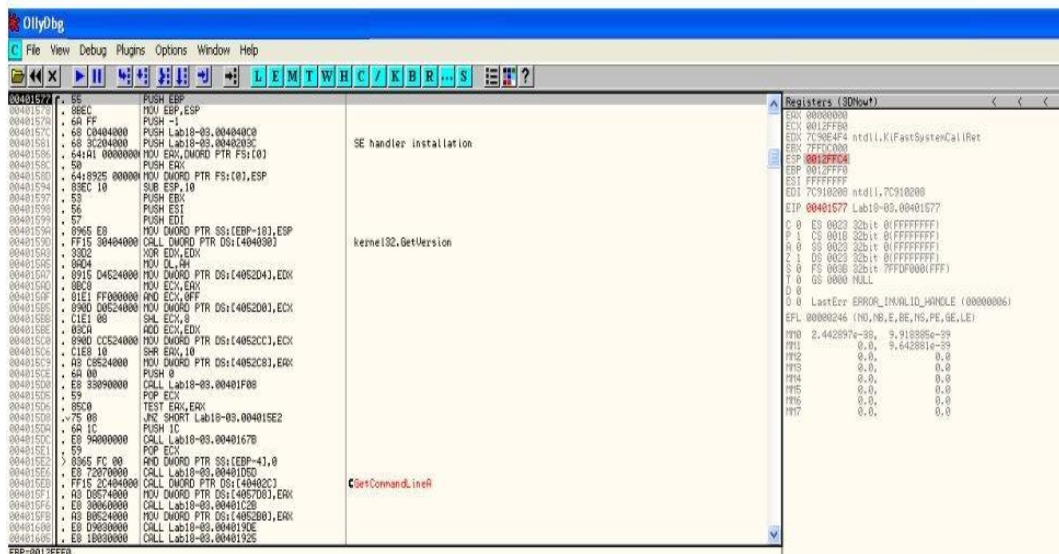


Рис. 52. Преобразованный код

После этого мы дамвим процесс, открываем Plugins → OllyDmp → Dump → Debugged process, в открывшемся окне обязательно щелкаем Get EIP as OEP и потом кнопку Dump (рис. 53).

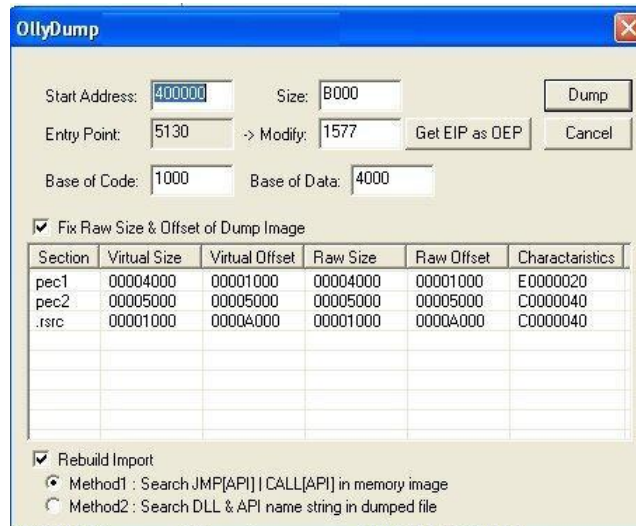


Рис. 53. Окно OllyDbg с опциями дампа

Получаем распакованный файл, который можно исследовать в дизАссемблере.

10. ПРОГРАММИРОВАНИЕ В СРЕДЕ GNU ASSEMBLER

Для работы в Mac OS X необходимо установить пакет XCode, имеющийся на установочном диске.

Для работы в Linux необходимо установить пакеты GNU Binutils и Java Runtime Environment (JRE).

Другие операционные системы не поддерживаются в силу того, что не соответствуют Single UNIX Specification, либо мало распространены.

В операционных системах Mac OS X и Linux для ассемблирования программ используется GNU Assembler. Его можно запустить, выполнив в терминале команду **as** с указанием файлов с исходным кодом и параметров ассемблирования. В результате ассемблирования создается объектный файл, который нужно скомпоновать в исполняемый файл командой **ld**. Пример ассемблирования и запуска программы приведен на рис. 54.

```
static-ip-192-168-1-11:examples sasha$ as -arch i386 -o ex.o ex1_mac32.s
static-ip-192-168-1-11:examples sasha$ ld -o example ex.o
static-ip-192-168-1-11:examples sasha$ ./example
Text string
```

Рис. 54. Пример ассемблирования программы

Директивы Ассемблера:

- **.intel_syntax noprefix** – переключает ассемблер на использование синтаксиса Intel;
- **.data** – начало секции данных;
- **.text** – начало секции кода;
- **.globl <идентификатор>** – указывает на то, что идентификатор должен быть глобальным;
- **.ascii "string"** – определяет C-строку string;
- **.space n, m** – заполняет n байтов значением m.

Комментарии

Комментарием считается любой текст, заключенный в скобки **/* */**. При ассемблировании комментарии игнорируются.

Константы

Константы могут быть записаны в следующих формах:

- 23 (целое число в десятичной системе счисления);
- 077 (целое число в восьмеричной системе счисления);
- 0xf80a (целое число в шестнадцатеричной системе счисления);
- 0b01101 (целое число в двоичной системе счисления);
- 't (целочисленный код символа t);
- '\n (целочисленный код символа перевода строки);

- “строка” (последовательность байтов символов);
- 0f+3.14 (вещественное число);
- 0f-1.6e-19 (вещественное число).

Список некоторых системных вызовов Unix

1. *exit*

SYS_EXIT = 1

Параметры:

1. *status*.

Завершает выполнение программы с указанным кодом завершения *status*.

2. *fork*

SYS_FORK = 2

Параметров нет.

Создает новый процесс, являющийся точной копией текущего процесса, фактически разделяет программу на два независимых параллельных потока выполнения.

Возвращаемые значения:

1. Идентификатор созданного процесса.
2. Содержит 0, если управление возвращено в родительский процесс, или 1, если управление возвращено в дочерний процесс.

Пример на псевдокоде:

$(pid, who) \leftarrow fork()$

if who = 1

Находимся в дочернем процессе с идентификатором *pid* *else*.

Находимся в родительском процессе.

3. *read*

SYS_READ = 3

Параметры:

1. *fd*.
2. **buf*.
3. *nbyte*.

Пытается прочитать *nbyte* байтов из файла, на который ссылается дескриптор *fd*, в буфер, на который указывает *buf*. В качестве дескрипторов могут быть использованы стандартные дескрипторы.

1. *STDIN* = 0 – стандартный ввод. Возвращаемые значения:

Положительное число, равное количеству прочитанных байтов в интервале 1..*nbyte*, 0 –

если достигнут конец файла, -1 в случае ошибки.

Пример на псевдокоде:

do

$br \leftarrow read(fd, buf, nbyte)$

if br = -1 Ошибка чтения *break*


```
nbyte ← nbyte − br buf ← buf + br  
while br > 0
```

4. *write*

SYS_WRITE = 4

Параметры:

1. *fildes*.
2. **buf*.
3. *nbyte*.

Пытается записать *nbyte* байтов из файла, на который ссылается дескриптор *fildes*, из буфера, на который указывает *buf*. В качестве дескрипторов могут быть использованы следующие стандартные дескрипторы.

1. *STDOUT* = 1 – стандартный вывод.
2. *STDERR* = 2 – стандартный вывод ошибок. Возвращаемые значения:
3. Положительное число, равное количеству записанных байтов в интервале 1..*nbyte*, -1 в случае ошибки.

Пример на псевдокоде:

```
do  
bw ← write(fd, buf, nbyte)  
if bw = -1 Ошибка записи break  
nbyte ← nbyte − bw buf ← buf + bw  
while nbyte > 0  
4. open  
SYS_OPEN = 5
```

Параметры:

1. **path*.
2. *oflag*.

Открывает файл, путь которого задан указателем на символьную строку *path*. Режим открытия файла задается следующими флагами *oflag*.

1. *O_RDONLY* = 0 – открытие файла только для чтения.
2. *O_WRONLY* = 1 – открытие файла только для записи.
3. *O_RDWR* = 2 – открытие файла для чтения и записи.
4. *O_APPEND* = 8 – открывает файл и перемещает указатель на конец файла, позволяя дописывать в него данные.

5. *O_CREAT* = 0x200 – создает новый файл, если файл с указанным путем не существует.

6. *O_TRUNC* = 0x400 – устанавливает размер открываемого файла в 0 и перемещает указатель на начало файла.

7. *O_EXCL* = 0x800 – возвращает ошибку, если установлен флаг *O_CREAT* и открываемый файл уже существует.

После завершения всех операций с файлом его необходимо закрыть с помощью системного вызова *close*.

Возвращаемые значения:

1. Неотрицательное число, являющееся дескриптором открытого файла или -1 в случае ошибки.

6. *close*

SYS_CLOSE = 6

Параметры:

1. *fildes*.

Закрывает файловый дескриптор *fildes*.

Возвращаемые значения:

1. Возвращает 0 в случае успешного закрытия, -1 в случае ошибки.

Системные вызовы в 32-битном окружении Mac OS X

В 32-битном окружении Mac OS X используются системные вызовы в стиле BSD. Для вызова ядра используется прерывание 0x80. Номер системного вызова передается через регистр *eax*.

Параметры системного вызова передаются через стек. После возврата из системного вызова вызывающая программа должна самостоятельно очищать стек. Первое возвращаемое значение записывается в регистр *eax*. Если есть второе возвращаемое значение, оно записывается в регистр *edx*.

Листинг 1.

.intel_syntax noprefix

/ Объявление констант */ SYS_EXIT = 1*

SYS_WRITE = 4

STDOUT = 1

/ Секция данных */*

.data

str:

.ascii "Text string\n"

/ Секция кода */*

.text

.g lobl start start:

/ Вывод строки */ push 12*

push offset str push STDOUT

mov eax, SYS_WRITE push eax

int 0x80 add esp, 16

/ Завершение программы */ push 0*

mov eax, SYS_EXIT push eax

int 0x80 add esp, 8

Исходный код помещается в файл с расширением *.s*, допустим, в файл *source.s*. Программа ассемблируется командами

```
as --arch i386 -o source.o source.s ld -o program source.o
```

После выполнения этих команд создается исполняемый файл program, который можно запустить командой

```
./program
```

Системные вызовы в 64-битном окружении Mac OS X

В 64-битном окружении Mac OS X системные вызовы осуществляются с помощью инструкции syscall. Номер системного вызова передается через регистр rax. Параметры системного вызова передаются через регистры в следующем порядке:

1. rdi.
2. rsi.
3. rdx.
4. rcx.
5. r8.
6. r9.

Первое возвращаемое значение записывается в регистр rax. Если есть второе возвращаемое значение, оно записывается в регистр rdx.

Листинг 2.

```
.intel_syntax noprefix
/* Объявление констант */ SYS_EXIT = 0x2000001 SYS_WRITE =
0x2000004
STDOUT = 1
/* Секция данных */
.data
str:
.ascii "Text string\n"
/* Секция кода */
.text
.g      lobl start start:
/* Вывод строки */ mov rax, SYS_WRITE mov rdi, STDOUT
lea rsi, [rip + str] mov rdx, 12
syscall
/* Завершение программы */ mov rax, SYS_EXIT
mov rdi, 0 syscall
```

Исходный код помещается в файл с расширением .s, допустим, в файл source.s. Программа ассемблируется командами

```
as --arch x86_64 -o source.o source.s ld -o program source.o
```

После выполнение этих команд создается исполняемый файл program, который можно запустить командой

```
./program
```


Системные вызовы в Linux

В Linux системные вызовы осуществляются с помощью прерывания 0x80, номер системного вызова передается через регистр *eax*, параметры системного вызова передаются через регистры в следующем порядке:

1. *ebx*.
2. *ecx*.
3. *edx*.
4. *esi*.
5. *edi*.

Первое возвращаемое значение записывается в регистр *eax*. Если есть второе возвращаемое значение, оно записывается в регистр *edx*.

Листинг 3.

```
.intel_syntax noprefix
/* Объявление констант */ SYS_EXIT = 1
SYS_WRITE = 4
STDOUT = 1
/* Секция данных */
.data
str:
.ascii "Text string\n"
/* Секция кода */
.text
.globl _start
_start:
/* Вывод строки */ mov edx, 12
mov ecx, offset str mov ebx, STDOUT
mov eax, SYS_WRITE int 0x80
/* Завершение программы */ mov ebx, 0
mov eax, SYS_EXIT int 0x80
```

Исходный код помещается в файл с расширением *.s*, допустим, в файл *source.s*.

В 32-битном окружении программа ассемблируется командами

```
as --32 -o source.o source.s
ld -m elf_i386 -o program source.o
```

В 64-битном окружении программа ассемблируется командами

```
as --64 -o source.o source.s
ld -m elf_x86_64 -o program source.o
```

После выполнения этих команд создается исполняемый файл *program*, который можно запустить командой

```
./program
```

11. КОМПОНОВКА АССЕМБЛЕРНЫХ ФУНКЦИЙ С ПРОГРАММАМИ, НАПИСАННЫМИ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Параметры функции – вещественные и мнимые части первого и второго комплексного числа. Функция должна выполнить некоторые действия с этими комплексными числами и вернуть квадрат модуля результата. Для упрощения задания все операции являются целочисленными.

Обратите внимание, что ко всем идентификаторам в коде на языке Си на этапе компиляции добавляется префикс в виде символа подчеркивания. Поэтому, чтобы использовать эти идентификаторы в ассемблерном коде, нужно добавлять к ним префикс, а в Си-коде использовать те же идентификаторы без префикса.

Для проверки корректности выполнения задания можно использовать следующую программу на языке Си:

```
#include <stdio.h>
extern int process( int, int, int, int ); int main() {
int re1, im1, re2, im2;
printf( "Enter four integer numbers and have fun\n" ); scanf( "%d", &re1 );
scanf( "%d", &im1 );
scanf( "%d", &re2 );
scanf( "%d", &im2 );
int result = process( re1, im1, re2, im2 );
printf( "The answer is %d. Have you expected such result? " "Think about
it and make it work.\n", result );
}
```

Допустим, эта программа содержится в файле main.c, а функция на Ассемблере – в файле func.s. Тогда, чтобы скомпоновать их в одну программу program и запустить ее, нужно выполнить следующие команды (Mac OS X 32):

```
gcc -c -arch i386 -o main.o main.c as -arch i386 -o func.o func.s
gcc -arch i386 -o program main.o func.o
./program
```

Для хранения параметров функций и локальных переменных в языке Си используется стек-фрейм. Это область памяти, выделенная на стеке. Вершина стека содержится в регистре esp. Перед вызовом функции в стек помещаются ее параметры, при этом вершина стека сдвигается вниз. Вызываемая функция использует стек для хранения локальных переменных и передачи параметров другим функциям. Поэтому, для удобства значение регистра esp в момент входа в функцию сохраняется в регистр ebp, после

чего регистр `esp` можно свободно изменять, а регистр `ebp` служит для доступа к параметрам функции (вверх от `ebp`) и локальным переменным (вниз от `ebp`). Таким образом образуется стек-фрейм. Перед завершением функция должна восстановить стек-фрейм вызывающей функции.

Программы на языке Си используют следующее соглашение о порядке вызова функций.

1. Вызывающий код помещает параметры вызываемой функции в стек (инструкция `push`) и вызывает функцию (инструкция `call`).

2. Вызываемая функция сохраняет текущее значение регистра `ebp` (база стек-фрейма вызывающей функции) в стек, а в регистр `ebp` помещает текущую вершину стека, содержащуюся в регистре `esp`.

3. Если функция должна возвращать какое-то значение, это значение сохраняется в регистр `eax`.

4. Перед завершением функция должна восстановить значения регистров `esp` и `ebp`.

5. Для возврата в вызывающий код нужно выполнить инструкцию `ret`.

Рассмотрим реализацию функций на языке Ассемблера на примере функции, которая возвращает сумму двух своих параметров. На языке Си такая функции выглядела бы следующим образом:

```
int sum( int a, int b ) { return a + b;
}
```

Эта же функция на языке Ассемблера:

```
.intel_syntax noprefix
.text
/* Начало функции sum */
.globl _sum
_sum:
/* Сохранение стек-фрейма вызывающей функции */ push  ebp
mov  ebp, esp
/* Доступ к параметрам функции осуществляется через регистр ebp
Первый параметр: [ebp + 8]
Второй параметр: [ebp + 12] Третий параметр: [ebp + 16] и так да-
лее */
mov  eax, [ebp + 8]
/* Результат сохраняется в регистре eax */ add  eax, [ebp + 12]
/* Восстановление стек-фрейма вызывающей функции */ mov  esp, ebp
pop  ebp
/* Выход из функции */ ret
```

12. ВЗАИМОДЕЙСТВИЕ С МИКРОКОНТРОЛЛЕРА ЧЕРЕЗ ATMEL STUDIO НА ЯЗЫКЕ АССЕМБЛЕРА

В данной работе мы будем рассматривать цифровые выходы контроллера PB0 – PB5. Данные разъемы являются цифровыми и могут работать в качестве ввода/вывода. Кроме того, если разъемы работают в режиме вывода, необходимо задать уровень напряжения для них. Всего уровней два. Максимальный уровень задается одним байтом.

За разъемы PB0-PB5 отвечают регистры **DDRB** и **PORTB**.

За определение режима работы разъема на вход/выход отвечает регистр – **DDRB**.

За указание разъемов, которые будут задействованы в работе, отвечает регистр **PORTB**.

Как вы могли заметить, у микроконтроллера имеются также разъемы с индексами C и D. Их мы в работе использовать не будем, однако они также имеют свои регистры ввода/вывода.

Прошивка микроконтроллера

Для записи готовой программы в микроконтроллер необходимо сформировать hex-файл. Он представляет собой построчный набор команд. Файл состоит из текстовых ASCII строк. Каждая строка представляет собой одну запись. Каждая запись начинается с двоеточия (:), после которого идет набор шестнадцатеричных цифр кратных байту:

```
:10010000214601360121470136007EFE09D2190140  
:100110002146017EB7C20001FF5F16002148011988  
:10012000194E79234623965778239EDA3F01B2CAA7  
:100130003F0156702B5E712B722B732146013421C7  
:00000001FF
```

Что соответствует:

- Начало записи
- Количество байт данных в этой записи (строке)
- Адрес, по которому размещаются данные этой записи
- Тип записи
- Данные
- Контрольная сумма записи

В данной лабораторной работе hex-файл будет формироваться после компиляции написанной вами программы.

Работа в Atmel Studio

Главный интерфейс программы Atmel Studio представлен на рис. 55.

Выбираем слева шаблон Assembler и указываем опционально параметры для проекта.

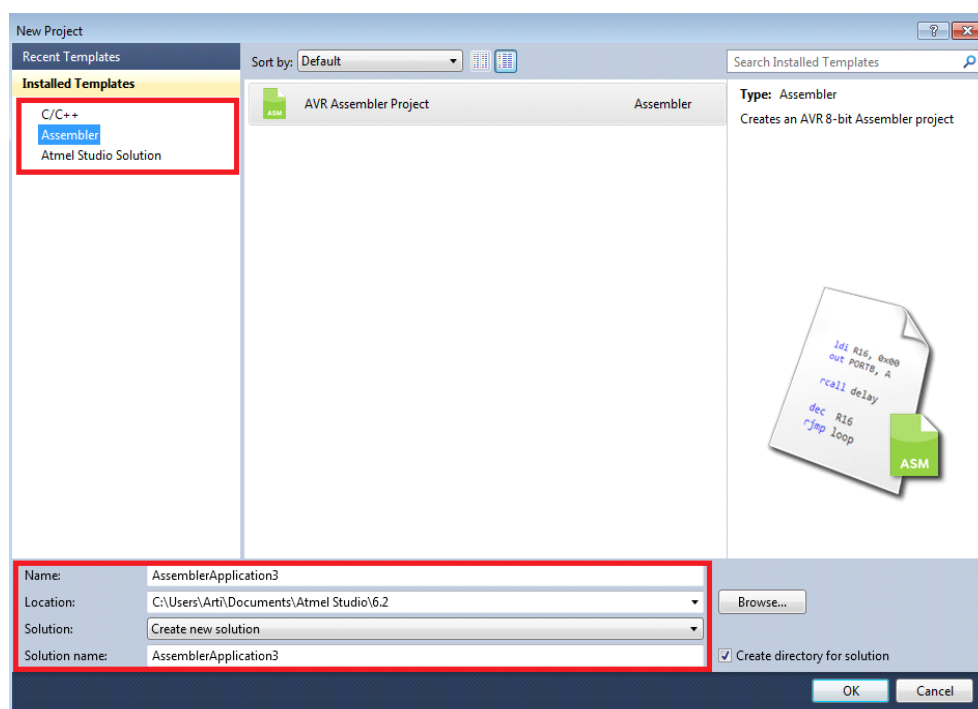


Рис. 55. Создаем проект в среде разработки AtmelStudio

Следующий шаг: указываем тип микроконтроллера – Atmega328p (рис. 56).

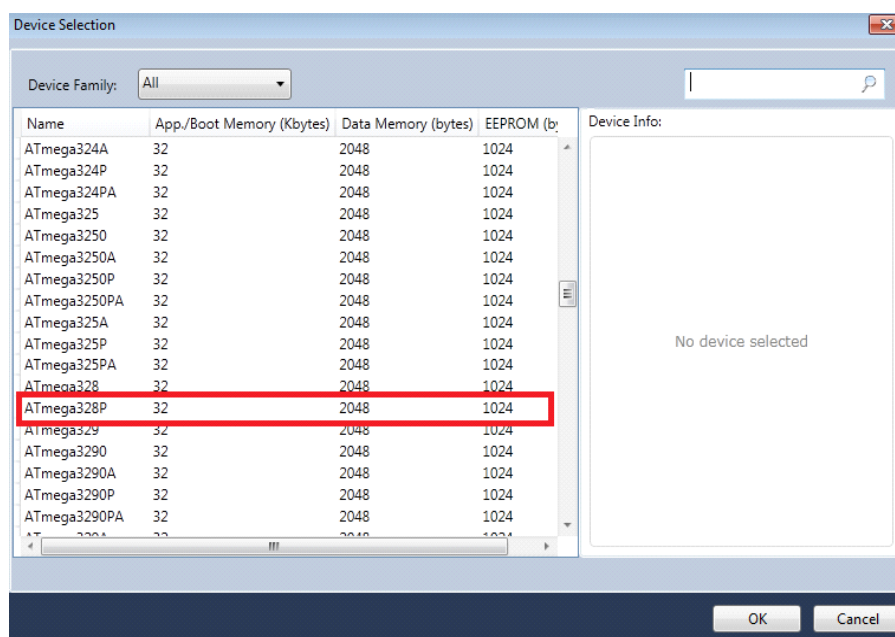


Рис. 56. Указываем тип микроконтроллера – Atmega328p

Создается пустой файл с расширением .asm. Напишем простую программу индикации светодиода (рис. 57).

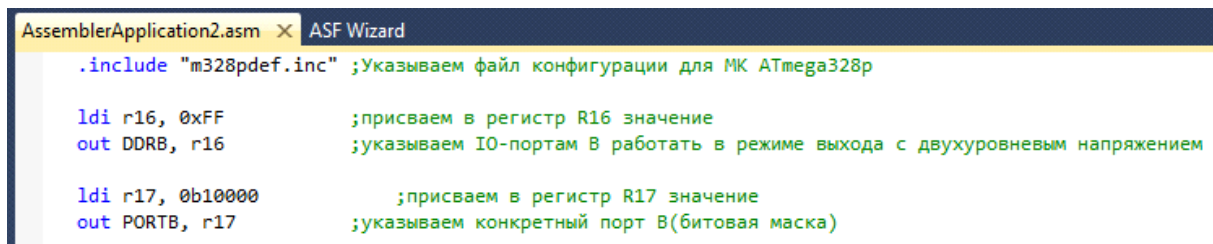


Рис. 57. Исходный файл

Собираем проект для получения файла прошивки .hex (рис. 58).

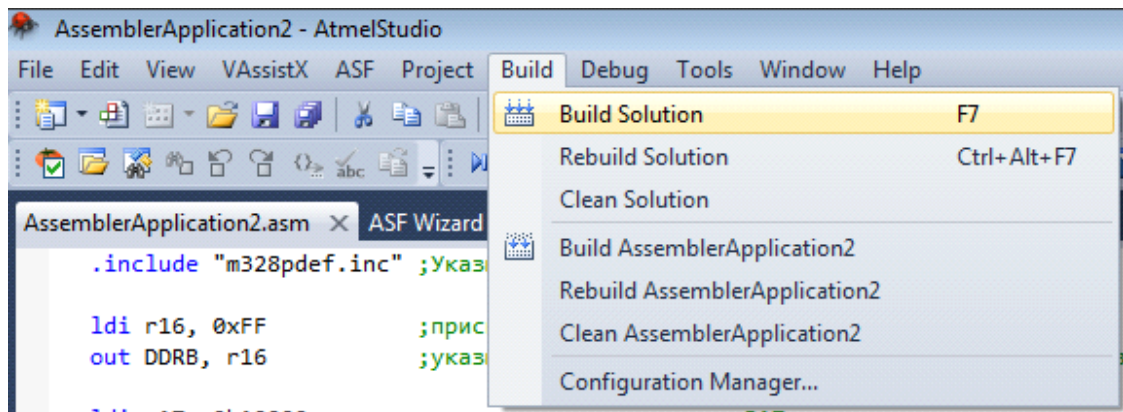


Рис. 58. Исходный файл 2

Итоговая компиляция представлена на рис. 59.

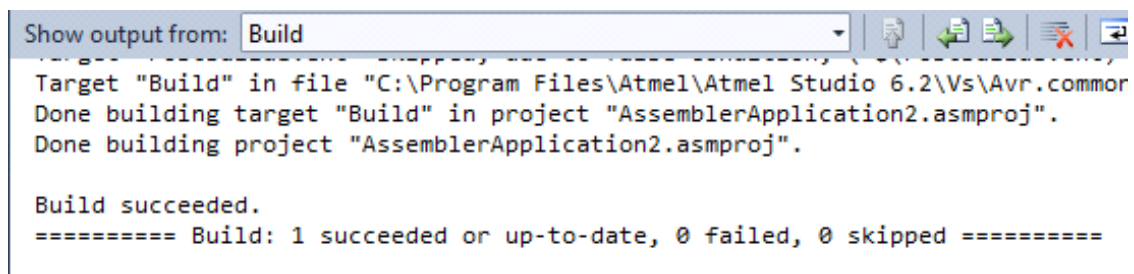


Рис. 59. Компиляция проекта

Получили файл с расширением .hex. Далее выполним пример программы:

```

.def delay1    = r17
.def delay2    = r18
.def delayv    = r19
.equ led       = 0b10 ; PORTD bit number to blink LED on
rjmp main
delay:
    clr    delay1
    clr    delay2
    ldi    delayv, 10

```

```

delay_Loop:
    dec    delay2
    brne   delay_Loop
    dec    delay1
    brne   delay_Loop
    dec    delayv
    brne   delay_Loop
    ret          ; go back to where we came from
main:
    sbi    DDRD, led ; connect PORTD pin 4 to LED
Loop:
    cbi    PORTD, led ; turn PD4 high
    rcall  delay      ; delay for an short bit
    sbi    PORTD, led ; turn PD4 low
    rcall  delay      ; delay again for a short bit
    rjmp   Loop       ; recurse back to the head of Loop

```

Поскольку ток протекает от положительного к отрицательному, анод светодиода должен быть подключен к цифровому сигналу 5 В, а катод должен быть подключен к земле. Мы будем подключать светодиод к цифровому контакту D10 Arduino последовательно с резистором. Светодиоды должны быть всегда соединены последовательно с резистором, который выступает в качестве ограничителя тока. Чем больше значение резистора, тем больше он ограничивает ток. В этом эксперименте мы используем резистор номиналом 220 Ом. Схема подключения приведена на рис. 60.

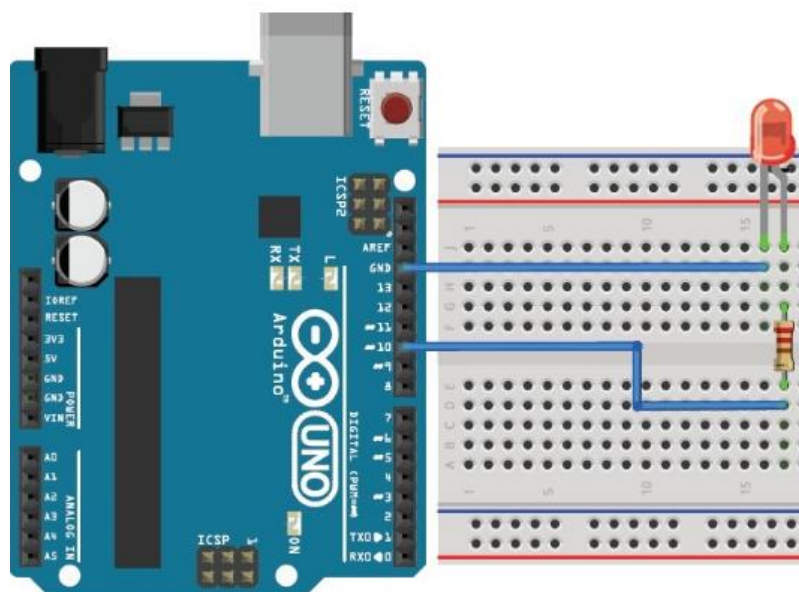


Рис. 60. Расположение светодиода

Для записи готовой программы в микроконтроллер необходимо сформировать hex-файл. Он представляет собой построчный набор команд. Файл состоит из текстовых ASCII строк. Каждая строка представляет собой одну запись. Каждая запись начинается с двоеточия (:), после которого идет набор шестнадцатеричных цифр, кратных байту (рис. 61).

```
:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

- Начало записи
- Количество байт данных в этой записи (строке)
- Адрес, по которому размещаются данные этой записи
- Тип записи
- Данные
- Контрольная сумма записи

Рис. 61. Представление записи в hex-файл

В данной работе hex-файл будет формироваться после компиляции написанной вами, программы. Далее следует прошивка микроконтроллера при помощи .hex (рис. 62).

```
C:\Windows\system32\cmd.exe
C:\Users\Arti\Documents\Atmel Studio\6.2\AssemblerApplication2\AssemblerApplication2\Debug>echo Firmwared controller
Firmwared controller
C:\Users\Arti\Documents\Atmel Studio\6.2\AssemblerApplication2\AssemblerApplication2\Debug>avrdude.exe -p stk500v1 -P COM3 -U flash:w:C:\Users\Arti\Documents\Atmel Studio\6.2\AssemblerApplication2\AssemblerApplication2\Debug\AssemblerApplication2.hex:i -U hfuse:w:<0x91>:m -U lfuse:w:<0xFF>:m
avrdude.exe: AVR device initialized and ready to accept instructions
Reading ! ##### ! 100% 0.00s
avrdude.exe: Device signature = 0x1e950f
avrdude.exe: erasing chip
avrdude.exe: reading input file "C:\Users\Arti\Documents\Atmel Studio\6.2\AssemblerApplication2\AssemblerApplication2\Debug\AssemblerApplication2.hex"
avrdude.exe: writing flash <8 bytes>:
Writing ! ##### ! 100% 0.01s
avrdude.exe: 8 bytes of flash written
avrdude.exe: verifying flash memory against C:\Users\Arti\Documents\Atmel Studio\6.2\AssemblerApplication2\AssemblerApplication2\Debug\AssemblerApplication2.hex
avrdude.exe: load data flash data from input file C:\Users\Arti\Documents\Atmel Studio\6.2\AssemblerApplication2\AssemblerApplication2\Debug\AssemblerApplication2.hex:
avrdude.exe: input file C:\Users\Arti\Documents\Atmel Studio\6.2\AssemblerApplication2\AssemblerApplication2\Debug\AssemblerApplication2.hex contains 8 bytes
avrdude.exe: reading on-chip flash data:
Reading ! ##### ! 100% 0.01s
avrdude.exe: verifying ...
avrdude.exe: 8 bytes of flash verified
avrdude.exe: reading input file "<0x91>"
avrdude.exe: invalid byte value <<0x91>> specified for immediate mode
avrdude.exe: write to file '<0x91>' failed
avrdude.exe: safemode: Fuses OK
avrdude.exe done. Thank you.
Для продолжения нажмите любую клавишу . . .
```

Рис. 62. Прошивка микроконтроллера

СПИСОК ЛИТЕРАТУРЫ

1. Штеренберг, С. И. Основы программирования на языке Ассемблер. Часть 1 : лабораторный практикум / С. И. Штеренберг, А. В. Красов, В. П. Просихин ; СПбГУТ. – СПб., 2015. – 25 с.
2. Штеренберг, С. И. Основы программирования на языке Ассемблер. Часть 2 : лабораторный практикум / С. И. Штеренберг, А. В. Красов, В. П. Просихин ; СПбГУТ. – СПб., 2015. – 35 с.
3. Красов, А. В. Программирование на языке Си++. Ч. 1. : методические указания к лабораторным работам / А. В. Красов ; СПбГУТ. – СПб., 2005. – 48 с.
4. Красов, А. В. Программирование на языке Си++. Ч. 2. : методические указания к лабораторным работам / А. В. Красов ; СПбГУТ. – СПб., 2005. – 28 с.
5. Красов, А. В. Основы информационных технологий : учеб. пособие / А. В. Красов ; СПбГУТ. – СПб., 2007. – 56 с.
6. Андрианов, В. И. Компьютерные курсы : методические указания к лабораторным работам / В. И. Андрианов, А. В. Красов, А. Ю. Цветков, И. А. Федянин ; СПбГУТ. – СПб., 2011. – 23 с.
7. Березин, Б. И. Начальный курс С и С++ / Б. И. Березин, С. Б. Березин. – М. : ДИАЛОГ-МИФИ, 2003. – 288 с.
8. Холзнер, С. Visual C++ 6. Учебный курс / С. Холзнер. – СПб. : Питер, 2006. – 570 с.
9. Абель, П. Ассемблер. Язык и программирование для IBM PC : пер. с англ. / П. Абель. – К. : Век+, М. : ЭНТРОП, М. : Корона-ВЕК, 2007. – 736 с.
10. Фролов, А. MS-DOS для программиста. Том 18. Часть 1 / А. Фролов, Г. Фролов. – М. : ДИАЛОГ-МИФИ, 1995. – 254 с.
11. Гульев, И. Компьютерные вирусы, взгляд изнутри / И. Гульев. – М. : ДМК, 1998. – 304 с.
12. Дойникова, Е. В. Оценка защищенности и выбор защитных мер в компьютерных сетях на основе графов атак и зависимостей сервисов : 05.13.19 : диссертация на соискание ученой степени кандидата технических наук / Е. В. Дойникова ; Санкт-Петербургский институт информатики и автоматизации РАН. – Санкт-Петербург, 2017. – 206 с.
13. Ахо, А. Компиляторы. Принципы, технологии, инструменты / А. Ахо, Р. Сети, Д. Ульман [и др.]. – М. : Вильямс, 2001. – 1184 с.
14. Федотов, И. Е. Параллельное программирование. Модели и приемы / И. Е. Федотов. – М. : СОЛОН-Пресс, 2017. – 390 с.
15. Андрианов, В. И. Способ защиты информационно-вычислительных сетей от компьютерных атак / В. И. Андрианов [и др.] // Патент РФ № 2011147613/08, 23.11.2011.
16. Буйневич, М. В. Метод алгоритмизации машинного кода телекоммуникационных устройств / М. В. Буйневич, К. Е. Израйлов // Телекоммуникации. – 2012. – № 12. – С. 2–6.
17. Buinevich, M. V. Method and utility for recovering code algorithms of telecommunication devices for vulnerability search / M. V. Buinevich, K. E. Izrailov // 16th International Conference on Advanced Communication Technology (ICACT). – 2014. – С. 172–176.
18. Буйневич, М. В. Автоматизированное средство алгоритмизации машинного кода телекоммуникационных устройств / М. В. Буйневич, К. Е. Израйлов // Телекоммуникации. – 2013. – № 6. – С. 2–9.
19. Лазарев, В. Г. Синтез управляющих автоматов. – 3-е изд., перераб. и доп. / В. Г. Лазарев, Е. И. Пийль. – М. : Издательство Энергоатомиздат, 1989.
20. Касперски, К. Искусство дизассемблирования / К. Касперски, Е. Рокко. – СПб. : БЧВ-Петербург, 2008. – 896 с.

**Штеренберг Станислав Игоревич
Красов Андрей Владимирович
Радынская Виктория Евгеньевна**

**АССЕМБЛЕР В ЗАДАЧАХ
ЗАЩИТЫ ИНФОРМАЦИИ**

Учебное пособие

Редактор *Л. К. Паршина*
Компьютерная верстка Н. А. Ефремовой

План издания 2019 г., п. 73

Подписано к печати 10.02.2020
Объем 5,25 печ. л. Тираж 30 экз. Заказ 1004
Редакционно-издательский отдел СПбГУТ
193232 СПб., пр. Большевиков, 22
Отпечатано в СПбГУТ