Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Оренбургский государственный университет»

В.В. Извозчикова

ПРОГРАММИРОВАНИЕ МИКРОПРОЦЕССОРНЫХ СИСТЕМ

Учебное пособие

Рекомендовано ученым советом федерального государственного бюджетного образовательного учреждения высшего образования «Оренбургский государственный университет» для обучающихся по образовательной программе высшего образования по направлению подготовки 09.03.02 Информационные системы и технологии

Оренбург
2021

УДК 004.3/.4’23(075.8)
ББК 32.973-018.я73
И34

Рецензент – доцент, кандидат технических наук, Е.В. Бурькова

Извозчикова, В.В.
И34 Программирование микропроцессорных систем: учебное пособие / В.В. Извозчикова. – Оренбургский гос. ун.-т. – Оренбург: ОГУ, 2021. –153 с.
ISBN 978-5-7410-2564-2

Учебное пособие включает теоретический материал и задания для выполнения 8 лабораторных работ и расчетно-графического задания, что позволит изучить базовые понятия и технологии, необходимые для разработки, компиляции и сборки программ на языке Ассемблера. Пособие также содержит описание возможностей и принципа работы программы Proteus и материал для выполнения лабораторных работ, связанных с программиро-ванием современных микроконтроллеров.

Каждая работа включает изложение теоретического материала, при-меры фрагментов программ на языке Ассемблера, а также примеры мо-делирования схем в программе Proteus, контрольные вопросы и задания.

Учебное пособие предназначено для изучения теоретического мате-риала и выполнения лабораторного практикума и расчетно-графического задания по дисциплине "Программирование микропроцессорных систем" для бакалавров по направлению подготовки 09.03.02 Информационные си-стемы и технологии.

УДК 004.3/.4’23(075.3)
ББК 32.973-018.я73

ISBN 978-5-7410-2564-2

© Извозчикова В.В., 2021
© ОГУ, 2021

Содержание

Введение

Добавление интеллектуальных функций в современную электронную аппаратуру базируется на использовании микропроцессоров (МП), программируемых логических интегральных схем (ПЛИС) или устройств типа «система на одном кристалле». Подобная интеграция позволяет автоматизировать процессы измерения, управления, контроля, регулирования и обработки информации, а также обеспечить такие свойства вычислительных комплексов, как многофункциональность, модифицируемость, адаптивность, обучаемость и ряд других. В отличие от персональных компьютеров, вычислительным ядром которых служат универсальные высокопроизводительные микропроцессоры, для интеграции в различные устройства применяют микроконтроллеры. Микроконтро́ллер (МК) – микросхема, которая сочетает на одном кристалле функции процессора и периферийных устройств, содержит ОЗУ или ПЗУ, то есть представляет собой однокристальный компьютер, способный выполнять простые задачи.

С появлением однокристальных микроЭВМ связывают начало эры массового применения компьютерной автоматизации в области управления. В связи со спадом отечественного производства и возросшим импортом техники, в том числе вычислительной, термин «микроконтроллер» (МК) вытеснил из употребления ранее использовавшийся термин «однокристальная микроЭВМ».

В пособии предложены базовые понятия и технологии, необходимые для разработки, компиляции и сборки программ на языке Ассемблера для микропроцессоров семейства Intel x86. Даны основные сведения для работы в режиме командной строки, запуска командных файлов, некоторые необходимые сведения по системам программирования на языке Ассемблера применительно к объему курса, изучаемого по данной специальности.

Кроме того, в учебном пособии рассматривается семейство микроконтроллеров AVR на примере ATmega. Дается описание ассемблера AVR, описание работы с программой AVR Studio 4. В практической части приводятся лабораторные задания для освоения работы с микроконтроллерами семейства AVR.

1 Система команд языка Ассемблера

1.1 Команды пересылки данных

1.1.1 Рассмотрим наиболее распространенные команды общего назначения.

Команда ***MOV*** приемник, источник.

Пересылка байта или слова между регистром и ячейкой или между двумя регистрами. Можно пересылать непосредственно адресуемое значение.

Примеры.

```
MOV AX, TABLE        ; из памяти в регистр
MOV TABLE, AX        ; из регистра в память
MOV ES:[BX], AX      ; с заменой используемого регистра сегмента
MOV CL, -30          ; константа в регистр
MOV TABLE, 5H        ; константа в память
```

Исключения:

1) нельзя пересылать из памяти в память (надо через регистр);

2) нельзя непосредственно адресуемый операнд пересылать в регистр сегмента;

3) нельзя пересылать регистр сегмента в регистр сегмента (надо через регистр общего назначения);

4) нельзя использовать регистр CS в качестве приемника.

Команда ***PUSH*** - источник; слово пересылается на вершину стека.

Команда ***POP*** - приемник ; вершина стека пересылается в слово.

Можно использовать любой 16-разрядный регистр (включая CS).

Команда ***XCHG*** приемник, источник.

Меняет между собой:

– значения двух регистров;

– значение регистра и ячейки памяти;

Нельзя обменять значения регистров сегмента.

Команда ***XLAT*** таблица-источник.

Выбирает значение элемента таблицы байтов и загружает его в регистр AL. Таблица может иметь до 256 элементов. Перед исполнением команды начальный адрес таблицы надо загрузить в регистр BX, а номер извлекаемого байта в регистр AL. Результат в регистре AL.

Пример. Извлечь 10 байт из таблицы TAB.

```
MOV AL, 10          ; номер байта – в AL
MOV BX, OFFSET      ; смещение TAB – в BX
XLAT TAB            ; 10-ый байт пересылается в AL
```

1.1.2. Команды ввода-вывода, работающие с изолированным адресным пространством ввода-вывода состоят из двух команд:

1) ***IN***– ввод;

2) ***OUT*** – вывод.

Команда ***IN*** аккумулятор, порт.

Команда ***OUT*** порт, аккумулятор.

Аккумулятор – регистр AL при обмене байтами и регистр AX при обмене словами.

Номер порта – десятичное значение от 0 до 255. В качестве операнда “порт” можно использовать регистр DX (если требуется указать порт больше 255).

Примеры.

```
IN AL, 200          ; ввести байт из порта 200
IN AL, PORT_V AL    ; из порта, указанного константой.
OUT 30H, AX         ; вывести слово в порт 30H
OUT DX, AX          ; или в порт, указанный DX.
```

(порт 96 – от клавиатуры, 97 – динамик, 64-67 – таймер, 994, . . . – монохромный дисплей, 976, . . . – цветной графический дисплей, 1001, . . . – контролер гибкого диска, 1013 – данные на гибком диске).

1.1.3. Команды пересылки адресов.

Команда ***LEA*** регистр 16, память 16.

Пересылает смещение ячейки памяти в любой 16-битовый регистр общего назначения, регистр указателя или индексный регистр.

В отличие от команды MOV с операцией OFFSET, операнд “память 16” в команде LEA может быть индексирован.

Пример.

LEA BX, TAB [DI]

*Команда **LDS*** (load pointer using DS).

LDS регистр 16, память 32.

Считывает из памяти 32-битовое двойное слово и загружает первые 16 битов в заданный регистр, а следующие 16 битов – в регистр сегмента данных DS.

Пример.

```
HERE_A DD HERE      ;    экви-   ; MOV BX, OFFSET HERE
. . .               ;   валент-  ; MOV AX, SEG HERE
LDS BX, HERE_A      ;      но    ; MOV DS, AX
```

В результате смещение и номер блока адреса метки HERE загружается в регистры BX и DS.

*Команда **LES*** (load pointer using ES).

Идентична LDS по загрузке номера блока в регистр ES, а не в регистр DS.

1.1.4. Пересылка флагов включает четыре команды.

*Команда **LAHF*** (load AH from Flags).

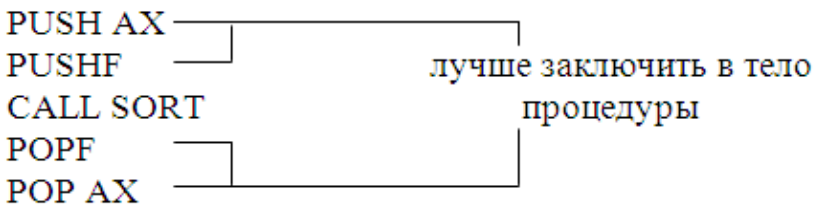
Команда LAHF без параметров. Загружает в регистр AH флаги CF, PF, AF, ZF, SF в соответствующие разряды (0, 2, 4, 6, 7).

*Команда **SAHF*** (store AH into Flags)

Загружает пять упомянутых выше разрядов регистра AH в регистр флагов.

*Команды **PUSHF**, **POPF*** – пересылка регистра флагов в стек и обратно.

Пример. Обратиться к процедуре с сохранением флагов и регистра AX.



1.2 Арифметические команды

В арифметических командах применяются следующие типы данных:

– десятичные без знака в упакованной форме (код старшей цифры занимает четыре старших бита байта, следующей цифры - четыре младших бита байта;

– десятичные без знака в неупакованной форме;

– двоичные со знаком (8 или 16 битов) в обратном коде, старший бит указывает знак;

– двоичные без знака (8 или 16 битов).

Над десятичными числами операции выполняются как над двоичными, но имеются специальные команды коррекции, которые обеспечивают получение правильного результата после выполнения операций над десятичными числами. 16 -битовые числа хранятся по принципу “младший байт-младший адрес, старший байт – старший адрес”.

1.2.1. Команды сложения ***ADD и ADC.***

Команда ***ADD*** приемник, источник.

Выполняется два действия:

а) приемник = приемник + источник;

б) формируется флаг переноса при выходе результата за разрядную сетку.

Команда ***ADC*** приемник, источник.

Выполняется сложение с учетом флага переноса:

приемник = приемник + источник + флаг переноса

Пример. Сложить 32-разрядные числа, которые предварительно загружены в регистры AX,BX– первое слагаемое и в регистры CX,DX – второе слагаемое.

```
(CX, DX) ← (AX, BX)
ADD AX, CX          ; младшие шестнадцать битов
ADC BX, DX          ; старшие 16 битов
```

1.2.2 Коррекция результата сложения (в регистрах AX, BX)/

Команда ***AAA*** – результат сложения корректируется для представления в кодах ACSII (после сложения неупакованных).

Команда ***DAA*** – результат сложения корректируется для представления в 10-й форме (после сложения упакованных).

1.2.3 Приращение приемника на единицу. При сложении операнда с единицей используется вместо команды ***ADD*** команда ***INC.***

Команда ***INC*** приемник.

1.2.4 Команды вычитания.

Команда ***SUB*** приемник, источник.

Вычитание с формированием знака переноса:

приемник = приемник - источник.

Команда ***SBB*** приемник, источник.

Вычитание с использованием знака переноса (вычитание с заемом):

приемник = приемник - источник - перенос.

1.2.5 Коррекция результата вычитания

Команда ***AAS*** – в кодах ASCII.

Команда ***DAS*** – в упакованной форме.

1.2.6 Уменьшение приемника на единицу. При вычитании из операнда единицы команду ***SUB*** заменяют специальной более короткой командой ***DEC.***

Команда ***DEC*** приемник.

Команда уменьшает приемник, но не воздействует на флаг переноса CF.

1.2.7 Обращение знака применяется при необходимости изменения знака у операнда.

Команда ***NEG*** приемник.

Выполнение: приемник = 0 - приемник.

1.2.8 Сравнение значений.

Команда ***CMP*** (compare).

CMP приемник, источник.

Команда производит вычитание, но не сохраняет результат. Используется только для формирования флагов.

1.2.9 Команды умножения.

Команды **MUL/IMUL** используются для выполнения операций умножения целых чисел без учета знака и с учетом знака соответственно. Особенностью ко-

манды умножения является то, что полученный результат может иметь двойную размерность. Формат команд одинаковый:

Команда ***MUL*** множитель (reg/mem); беззнаковое умножение

Команда ***IMUL*** множитель (reg/mem); знаковое умножение

Если при умножении оба операнда имеют размерность байт, то множимое должно быть предварительно загружено в регистр AL, а множитель может находиться в байте памяти или в однобайтовом регистре. После умножения результат будет находиться (согласно принятым соглашениям) в регистре AX.

При умножении слова на слово, множимое должно находиться в регистре AX, а множитель – либо в слове памяти или в двухбайтовом регистре. После умножения произведение будет находиться в двух регистрах (размер двойное слово): старшее слово произведения будет находиться в регистре DX, а младшее слово в регистре AX.

Примеры умножения.

```
dat_1 segment para public 'data‘ ;Описание сегмента данных
byte_1 db 36h
byte_2 db 58h
byte_3 db - 25h
word_1 dw 6020h
word_2 dw 3480h
word_3 dw - 1450h ; Данные
res_1 dw ?
res_2 dw ?
res_3 dd ?
res_4 dd ?
res_5 dd ?
res_6 dd ? ; Результаты
dat_1 ends
cod_1 segment para public 'code' ; Сегмент кода
. . . .
;Беззнаковое умножение.
```

```
MOV AL, byte_1                  ; байт на байт
MUL byte_2                       ; произведение в AX
MOV res_1, AX                   ; запись результата в память
MOV AX, word_1                  ; слово на слово
MUL word_2                       ; произведение в DX:AX
MOV word ptr  rez_3 , AX            ; запись младшей части в память
MOV word ptr  rez_3 +2 , DX         ; запись старшей части в память
MOV AL, byte_1                  ; байт на слово
SUB AH, AH                      ; обнуление AH
MUL word_1                      ; произведение в DX:AX
MOV word ptr  rez_4 , AX            ; запись младшей части в память
MOV word ptr  rez_4 +2 , DX         ; запись старшей части в память
;Знаковое умножение.
MOV AL, byte_1                  ; байт на байт
IMUL byte_3                      ; произведение в AX
MOV res_2, AX                   ; запись результата в память
MOV AX, word_1                  ; слово на слово
IMUL word_3                      ; произведение в DX:AX
MOV word ptr  rez_5 , AX            ; запись младшей части в память
MOV word ptr  rez_5 +2 , DX         ; запись старшей части в память
MOV AL, byte_1                  ; байт на слово
CBW                             ; расширение множимого в  AH
IMUL word_1                     ; произведение в DX:AX
MOV word ptr  rez_6 , AX            ; запись младшей части в память
MOV word ptr  rez_6 +2 , DX         ; запись старшей части в память
 . . . .
Cod_1 ends
End .
```

В случае умножения на число, которое является степенью числа 2 (2, 4, 8 и т.д.) более эффективным способом является сдвиг влево на число битов, соответствующее показателю степени. При сдвиге более чем на один разряд необходимо

загрузить величину сдвига в регистр CL, например: В следующих примерах предположим, что множимое находится в регистре AL или AX:

1) умножение на 2: SHL AL,1;

2) умножение на 8: MOV CL,3

SHL AL, CL.

1.2.10 Команды деления.

Команды ***DIV/IDIV*** используются для выполнения операций деления целых чисел без учета знака и с учетом знака соответственно. Особенностью команды деления является наличие двух результатов (частного и остатка). Это влияет на выполнение команд, в частности на использование регистров. Формат команд одинаковый:

Команда ***DIV*** делитель (reg/mem); беззнаковое деление

Команда ***IDIV*** делитель (reg/mem); знаковое деление, при этом знак устанавливается у частного

Деление «слова на байт». Делимое находится в регистре AX, а делитель – в байте памяти или а однобайтовом регистре. После деления остаток получается в регистре AH, а частное – в AL. Так как однобайтовое частное очень мало (максимально +255 (шест.FF) для беззнакового деления и +127 (шест.7F) для знакового), то данная операция имеет ограниченное использование.

Деление «двойного слова на слово». Делимое находится в регистровой паре DX:AX, а делитель – в слове памяти или а регистре. После деления остаток получается в регистре DX, а частное в регистре AX. Частное в одном слове допускает максимальное значение +32767 (шест.FFFF) для беззнакового деления и +16383 (шест.7FFF) для знакового.

1.2.11. Примеры деления.

```
data1 segment para public 'data‘ ;Описание сегмента данных
byte1 db 80h
byte2 db 16h
word1 dw 2000h
word2 dw 0010h
word3 dw 1000h ; Данные
```

```
data1 ends
code1 segment para public 'code'                ; Сегмент кода
. . . .
; Беззнаковое деление
MOV AX, word1                          ;слово / на байт
DIV byte1                          ;ост. : частное в AH:AL
MOV AL, byte1                          ;байт / на байт
SUB AH, AH                             ;обнуление AH
DIV byte2                          ; ост. : частное в AH:AL
MOV DX, word2                          ;двойное слово / на слово
MOV AX, word3                          ;делимое в DX:AX
DIV word1                          ;ост. : частное в AH:AL
MOV AX, word1                          ;слово / на слово
SUB DX, DX                             ;обнуление DX
DIV word3                           ; ост. : частное в DX:AX
; Знаковое деление
MOV AX, word1                        ;слово / на байт
IDIV byte1                         ;ост. : частное в AH:AL
MOV AL, byte1                        ;байт / на байт
CBW                                  ;расширяем делимое в  AH
IDIV byte2                          ; ост. : частное в AH:AL
MOV DX, word2                         ;двойное слово / на слово
MOV AX, word3                         ;делимое в DX:AX
IDIV word1                         ;ост. : частное в AH:AL
MOV AX, word1                         ;слово / на слово
CWD                                   ;расширяем делимое в  DX
IDIV word3                          ; ост. : частное в DX:AX
code ends
End
```

При делении на степень числа 2 (2, 4, и т.д.) более эффективным является сдвиг вправо на требуемое число битов. В следующих примерах предположим, что делимое находится в регистре AX:

Деление на 2: SHR AX,1

Деление на 8: MOV CL,3

SHR AX,CL

В операциях деления над знаковыми числами (IDIV), знак устанавливается у частного.

Нужно также учитывать, что в современных микропроцессорах (старше 386) появились и новые возможности команд деления и умножения. Изучите эти возможности самостоятельно.

1.3 Команды обработки битов

1.3.1 Логические команды.

*Команда **AND*** приемник, источник ; логическое “и”

*Команда **OR*** приемник, источник ; логическое “или”

*Команда **XOR*** приемник, источник ; логическое “исключающее или”

*Команды **NOT*** приемник ; логическое “не”

*Команды **TEST*** приемник, источник ; логическое “и” без записи в результат.

1.4 Команды передачи управления

Команды передачи управления МП К1810 подразделяются на команды безусловных переходов, условных переходов, вызовов, возвратов, управления циклами и команды прерываний.

1.4.1 Команды безусловных переходов.

При выполнении команд безусловных переходов происходит модификация PC или PC и CS, а их прежнее содержимое теряется.

*Команда **JMP*** или безусловный переход. Команда занимает от 2 до 5 байтов в зависимости от нахождения метки в том же, в котором используется команда JMP или в другом сегменте.

*Двухбайтная команда **JMP dispL*** содержит во втором байте смещение, которое интерпретируется как знаковое целое. Если смещение положительно, осуществляется переход вперед, если отрицательное – назад. Удобна для организации коротких программных циклов.

Пример.

(PC)=1240 JMP E8 (PC)=1228

*Трехбайтная команда **JMP disp*** производит такое же действие, как предыдущая команда, но содержит 16-битное смещение. При этом увеличивается область перехода до -32768 (+32767 относительно адреса команды, находящейся после команды JMP disp.

Пример.

(PC)=3E60 JMP 0002 (PC)=4060

*Команда **JMP mem/reg*** реализует косвенный безусловный переход в программе. Здесь адресом перехода, загружаемым в PC, служит содержимое 16-битного общего регистра или слова памяти, определяемое постбайтом режима адресации.

Пример.

(BX)=3000 JMP BX (PC)=3000

Пример.

(BX)=68A0 JMP [BX] (PC)=3560

(DS)=AA00

([B08A0])=3560

*Команда **JMP addr*** (прямого межсегментного перехода) содержит 4 байта прямого адреса перехода, которые определяют новое содержимое регистров PC и CS: значение **off** загружается в PC, а значение **seg** – в регистр CS.

Пример.

(PC)=18A6 JMP 0020 A040 (PC)=2000

(CS)=0200 (CS)=40A0

Команда ***JMP mem*** (косвенного межсегментного перехода) допускает адресацию только памяти. Слово из адресуемой ячейки памяти загружается в PC, а следующее слово – в регистр CS.

Пример.

(DS)=6000 JMP[DI+1H] (PC)=3750

(DI)=2680 (CS)=F000

([62780])=3750

([62782])=F000

1.4.2 Команды условных переходов.

В системе команд МП К1810 есть 19 двухбайтных команд условных переходов, называемыми также разветвлениями. Все они имеют единый формат: <код> <близкая метка>, представленный на рисунке 1.1.

Рисунок 1.1 – Формат команд условных переходов

При выполнении этих команд анализируется некоторое условие, закодированное текущими состояниями флажков (а в команде JCXZ – содержимым регистра CX), и в зависимости от удовлетворения условия переход осуществляется или нет. Данные команды позволяют проверить оба состояния всех флажков арифметических операций (кроме флажка AF), а также ряд комбинаций состояний нескольких флажков. Если условие истинно, управление передается по адресу перехода путем прибавления к содержимому PC однобайтового знакового смещения (с расширением знака до 16 бит), а если условие ложно, выполняется следующая по порядку команда. Таким образом, все условные переходы в МП К1810 являются короткими. Команды условных переходов представлены в таблице 1.1.

Знак * указывает, что команды относятся к действиям над числами со знаком (в обратном коде).

Используемые флаги:

– ***OF***-переполнение (overflow);

– ***SF***-знака (sign);

– ***ZF***-нуля (zero);

– ***PF***-четности (pariti);

– ***CF***-переноса (carry, равняется 1, если произошел перенос 1 при сложении или заем при вычитании);

– ***AF***-вспомогательный флаг переноса (аналогичен флагу CF для 3-го бита данных, полезен при операциях над упакованными десятичными числами).

Таблица 1.1– Команды условных переходов

Код	**Описание**	**Условие перехода**
JA JNBE	перейти, если выше если не ниже и не равно	CF=0 2F=0
JAE JNB	если выше или равно если не ниже	CF=0
JB JNAE JC	если ниже если не выше и не равно если перенос	CF=1
JBE JNA	если ниже или равно если не выше	CF=1 или 2F=1
JCXZ	если значение регистра CX равно 0	 CX=0
JE JZ	если равно если 0	ZF=1
JG* JNLE *	если больше если не меньше и не равно	ZF=0 & ZF=OF
JGE* JNL*	если больше или равно если не меньше	ZF=OF
JL* JNGE*	если меньше если не больше и не равно	ZF(CF
JLE* JNG*	если меньше или равно если не больше	ZF=1 или ZF(OF
JNC	если нет переноса	CF=0
JNE JNZ	если не равно если не нуль	ZF=0
JNO*	если не переполнение	OF=0
JNP JPO	если нет четности (нечетная сумма разрядов) если сумма битов нечетная	PF=0
JNC*	если знаковый разряд нулевой	CF=0
JO*	если переполнение	OF=1
JP JPE	если есть четность (четная сумма битов)	PF=1
JS	если знаковый бит равен 1	SF=1

Командам условной передачи управления могут предшествовать любые команды, не изменяющие состояния флагов, но обычно они используются совместно с командой сравнения CMP (таблица 1.2).

Таблица 1.2 – Совместное использование команд условного перехода с CMP

Условия перехода	**Следующая за CMP команда**	
	Для чисел без знака	**Для чисел со знаком**
Приемник больше источника	JA	JG
Приемник равен источнику	JE	JE
Приемник не равен источнику	JNE	JNE
Приемник меньше источника	JB	JL
Приемник меньше либо равен источнику	JBE	JLZ
Приемник больше либо равен источнику	JAE	JGS

*Команду **JCXZ*** удобно помещать в начале цикла, особенно в том случае, если возможна ситуация, при которой цикл (со счетчиком CX) не выполняется ни разу.

При программировании может возникнуть необходимость передачи управления за пределы действия команд условного перехода. Для осуществления длинных условных переходов на расстояние больше +127 или -128 байт от команд перехода надо привлекать команду длинного безусловного перехода.

Например. При далекой метке M, оператор IF AX=BX THEN GOTO M следует реализовать согласно приведенному ниже примеру.

```
IF AX<>BX THEN GOTO L;  ( короткий переход)
GOTO M;     (длинный переход)
L: . . . .
```

На ЯА это записывается следующим образом.

```
CMP  AX, BX
JNE L
JMP M
L: . . . .
```

Получается не очень хорошо, но иного варианта нет.

1.4.3 Управление циклами.

Формат у всех одинаковый: <код> <близкая метка>. Команды управления циклами приведены в таблице 1.3.

Таблица 1.3 – Команды управления циклами

Команда	Действие	Используемые данные
LOOP	Повторяет цикл до конца счетчика, т.е. уменьшает CX на 1 и передает управление на метку, если CX неравен 0	CX-счетчик
LOOPE LOOPZ	Уменьшает CX на 1 и осуществляет переход, если CX неравно 0 и флаг нуля ZF=1	CX-счетчик ZF-флаг нуля (пока ZF=1)
LOOPNE LOOPNZ	Уменьшает CX на 1 и осуществляет переход, если CX не равно 0, и флаг нуля ZF=0	CX-счетчик ZF-флаг нуля (пока ZF=0)

Пример. Найти первый ненулевой байт в заданном блоке памяти.

Дано: BX – смещение начального адреса; DI – смещение конечного адреса.

Результат: BX – смещение ненулевого байта или он равен DI, если такого нет.

```
SUB DI, BX                  ; Счетчик байтов = (DI) - (BX)+1
INC DI
MOV CX, DI                  ; Счетчик в CX
DEC BX                      ; BX := BX - 1
NEXT: INC BX                ; BX := BX + 1
CMP BYTE PTR [BX], 0        ; сравнение с 0
LOOPE NEXT
JNZ ZERO_FIND               ; если не 0, то переход.

  ……..                      ; участок программы, соответствующий
                            ; не найденному ненулевому элементу.
ZERO FIND:                  ;участок программы, соответствующий
                            ; найденному ненулевому элементу.
```

1.5 Команды прерывания

Прерывание заставляет микропроцессор сохранить в стеке информацию для последующего возврата, а затем перейти к группе команд, находящейся в некоторой ячейке памяти. Адрес получается из вектора прерывания-3 битовый. Прерывание сохраняет флаги.

Имеется три *команды* прерывания:

– две вызова ***INT, INTO;***

– одна возврата ***IRET.***

1.5.1 Команда прерывания INT.

Команда ***INT*** <тип прерывания> (тип прерывания – число от 0 до 255).

Выполнение.

1. Регистр флагов загружается в стек.
2. Обнуляет флаг трассировки TF и флаг включения/выключения прерываний IF для исключения пошагового режима пополнения команд и блокировки других маскируемых прерываний.
3. Помещается в стек значение регистра CS.
4. Вычисляется адрес вектора прерываний (умножением <типа прерывания> на 4).
5. Загружается второе слово вектора прерываний в регистр CS.
6. Помещается в стек значение указателя команд IP.
7. Загружается в указатель команд IP первое слово вектора прерываний.

Состояние стека показано на рисунке 1.2.

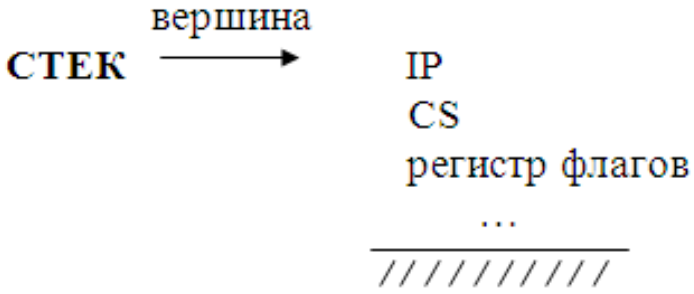


Рисунок 1.2 – Состояние стека после команды прерывания

1.5.2 Команда прерывания по переполнению INTO.

Условное прерывание. Инициирует прерывание только при флаге переполнения OF=1.

1.5.3 Возврат после прерывания IRET.

Извлекает из стека три 16-битовых значения и загружает их:

1) в IP;

2) в CS;

3) в регистр флагов.

Содержимое других регистров может быть уничтожено, если в программе обработки прерываний не предусмотрено их сохранение.

2 Структура программы на языке Ассемблера

Ассемблер – это машинно-ориентированный язык низкого уровня. Программа-ассемблер заменяет мнемонические обозначения команд и операндов соответственно на коды команд и адреса операндов. Этот процесс называют ассембли-рованием кода.

Ассемблерная программа состоит из операторов и директив. Операторы – это инструкции, исполняемые процессором (например, MOV, ADD и т. д.). Директивы, как правило, служат для указания режимов работы ассемблера (например, директива .MODEL, см. ниже), для разбиения потока операторов на сегменты и процедуры, определения данных (также см. ниже), указания размера операнда (BYTE PTR, WORD PTR) и выполнения некоторых других операций.

Некоторые команды (операторы) могут быть снабжены меткой, по которой будет сгенерирован адрес команды. Метка должна начинаться с латинской буквы либо символов $, _, @, и может содержать в себе латинские буквы и символы $, _, @, и цифры. В качестве метки нельзя использовать зарезервированные слова, названия команд и директив.

Общий вид оператора на языке ассемблера приведен ниже.

[Метка:] Код_операции [Операнды] [; Комментарий].

В квадратных скобках указаны поля, которые могут отсутствовать, то есть необязательные поля.

Приведем несколько примеров операторов:

```
Metk_1: MOV BX, [SI]        ; Загрузить  в регистр BX содержимое ячейки
                              ; памяти  со смещением в регистре SI.
        ADD BX, ES:[SI]     ; Сложить содержимое регистра BX с содержи-
                              ; мым ячейки памяти в дополнительном сег-
                            ; менте данных со смещением в регистре SI.
        PUSH BX               ; Запись содержимое регистра BX в стек.
        STD                   ; Установить флаг направления в единицу.
```

Программа обычно начинается с директивы «.MODEL SMALL», которая сообщает ассемблеру, что используются сегменты размером не более 64 килобайт.

Программа состоит из трех сегментов: кода, данных и стека. Сегменты определяются с помощью двух директив: начало (открытие) сегмента с помощью директивы SEGMENT, а конец (закрытие) сегмента - ENDS. Рассмотрим пример определения сегмента.

Имя SEGMENT Тип

Операторы

…

Имя ENDS,

где Имя – произвольное имя сегмента;

Тип – ‘code’ – сегмент кода; ‘data’ – сегмент данных; ‘stack’ – сегмент стека.

Написание сегментов будет рассмотрено ниже.

Заканчивается программа директивой END. Она имеет вид:

END метка,

где метка – название метки, определяющей адрес оператора программы, который должен быть выполнен первым при старте программы.

Рассмотрим создание сегмента данных.

Сегмент данных предназначен для определения и хранения данных и результатов программы. Для определения и резервирования данных существуют специальные директивы:

1) DB – размер константы один байт;

2) DW – размер константы два байта (слово);;

3) DD – размер константы четыре байта (двойное слово);.

Эти директивы имеют 3 формы:

1) D* константа – выделение памяти под переменную и занесение в нее указанной константы;

2) D* константа, константа, …, константа – выделение памяти под массив с одновременным занесением в него данных. Используется для небольших массивов;

3) D* число элементов DUP (константа) – используется для больших массивов и служит для выделения памяти под массив с нужным числом элементов и занесение в каждый элемент массива определенной константы.

Вместо конкретной константы в этих директивах можно указать символ ?, обозначающий, что под переменную или массив просто выделяется место с неопределенным значением.

Пример сегмента данных.

```
Dats_1 SEGMENT ‘data’
X_1   DB    20              ; Переменная с начальным значением 15
X_2   DW    ?               ; Переменная с неопределенным значением
Mass_5 DB 25 DUP(1)         ; Массив из 25 элементов с единичным значением
Mass_2 DW 1,17,4,-25,0,3    ; Массив из 6 элементов с заданными значениями
Dats_1 ENDS
```

Рассмотрим определение сегмента стека.

Сегмент стека обычно состоит из директивы DB 100 DUP (?), предписывающей ассемблеру выделить 100 байт под стек:

```
Stseg2 SEGMENT STACK     ‘stack’
DB 256 DUP (?)
Stseg2 ENDS.
```

Рассмотрим определение сегмента кода.

Сегмент кода содержит все операторы (команды) программы, Которые могут быть разбиты на процедуры и макросы. Описание этого сегмента всегда содержит директиву ASSUME, которая делает привязку сегментов к сегментным регистрам и имеет следующий вид:

ASSUME CS: имя сегмента кода, DS: имя сегмента данных, SS: имя сегмента стека.

Подпрограммы (процедуры) определяются с помощью директив начала PROC, конца ENDP и возврата RET следующим образом.

```
Имя   PROC Тип
Операторы
```

…

RET ; Возврат из подпрограммы в точку вызова

Имя ENDP.

Имя подпрограммы – это метка, по которой определяется смещение подпрограммы в сегменте кода. Тип может быть NEAR (близкий) и FAR (далекий). Подпрограммы типа NEAR вызываются в пределах текущего сегмента, а FAR – из текущего или другого сегмента. По умолчанию тип сегмента принимается NEAR.

Как правило, в программах, не использующих библиотеки подпрограмм и имеющих один сегмент кода, все подпрограммы, кроме одной, делают NEAR. Основная (главная) подпрограмма всегда объявляется как FAR, так как в конце программы необходимо передать управление операционной системе с помощью команды, находящейся в другом сегменте.

Любая подпрограмма может вызывать другие подпрограммы (аналогично тому, как это делается в языках высокого уровня). Вызов подпрограммы осуществляется с помощью команды CALL.

CALL имя_подпрограммы.

При вызове подпрограмм типа NEAR в стек автоматически записывается значение смещения следующей за CALL команды и производится переход к вызываемой подпрограмме. По команде RET это значение из стека записывается в регистр IP, и выполнение программы продолжается с команды, следующей за CALL. Для подпрограмм типа FAR в стек записывается не только значение смещения, но и значение базового адреса сегмента точки возврата. В этом случае по команде возврата RET в регистры CS и IP записывается адрес точки возврата CS:IP.

2.1 Форматы ассемблерных программ для последующего создания EXE и COM файлов

Существует два типа выполняемых программ: EXE-программы и COM-программы.

В EXE-программе обычно используется несколько сегментов памяти (сегменты кода, данных, стека и.т.д) и каждый из этих сегментов должен быть в программе описан.

COM-программа всегда использует только один сегмент памяти. То есть под код, данные и стек выделяется в памяти один и тот же сегмент.

При запуске EXE-программы и загрузке ее в оперативную память DOS автоматически настраивает сегментные регистры DS и ES на начало PSP, который резервируется операционной системой DOS непосредственно перед COM- или EXE-программой в памяти, а регистр CS – на начало сегмента кода.

Для COM-программы CS также как и DS и ES настраивается на начало PSP.

EXE-программа должна иметь примерно следующий формат.

```
stacks_1 segment para stack 'stack'          ;Описание сегмента стека
db 100 dup (?)                               ;Под стек выделяется 100 байт
stacks_1 ends
dats_1 segment para public  'data'           ;Описание сегмента данных
…                                            ; Данные
dats_1 ends
cods_1 segment para public 'code'            ; Сегмент кода
assume cs: cods_1, ds: dats_1, ss: stacks_1  ;Оператор осуществляет при-
                                             ; вязку сегментных регистров к
                                             : ;описанным сегментам. Необ-
                                             хо- ;дим для правильной транс-
                                             ляции
start: mov ax,dats_1                         ; В сегментный регистр
mov ds,ax                    ; ds загружается базовый адрес сегмента данных
    …                        ; программа
mov ah,4ch                   ; выход в операционную систему
int 21h
cods_1  ends
end start
```

COM-программа должна иметь примерно следующий формат.

```
code segment 'code'          ; описание сегмента кода
assume cs:code, ds:code      ; привязка сегментных регистров
org 100h                     ; выделяет 100h байт под PSP
begin: jmp start             ; обходим область данных
…                           ; область данных
start:
…                            ; программа
mov ah, 4ch                  ; выход в операционную систему
int 21h
code ends
end begin
```

Ассемблерная программа пишется практически в любом редакторе, даже в notepad, но имейте в виду, что ввод для Ассемблера должен осуществляться однобайтовыми символами и должен включать служебные символы. Поэтому ввод текста в MS WORD недопустим. Ассемблерная программа должна иметь расширение .asm.

После написания программа (имя.asm) должна быть оттранслирована с помощью транслятора tasm.exe (в командной строке набирается: tasm.exe имя asm или atasm.bat имя asm). В результате получается файл имя.obj.

В последнем необходимо установить связи с помощью редактора связей tlink.exe (в командной строке для EXE-файла набирается: tlink.exe имя.obj или atlink.bat имя.obj, для COM-файла- tlink.exe /t имя.obj).

В результате получится требуемый файл имя.exe (имя.com), который может быть запущен на выполнение.

Если программе требуется отладка, рекомендуется использовать отладчик Turbo Debuger.

Пример работы с отладчиком приведен в следующем разделе.

2.2 Ввод текста, компиляция, редактирование и отладка

В предыдущем разделе мы кратко рассмотрели технологические вопросы, связанные с подготовкой и созданием программ на Ассемблере.

В этом разделе мы более детально обратимся к основным понятиям и особенностям такой работы.

Процессы преобразования программ можно упрощенно представить, как показано на рисунке 2.1.

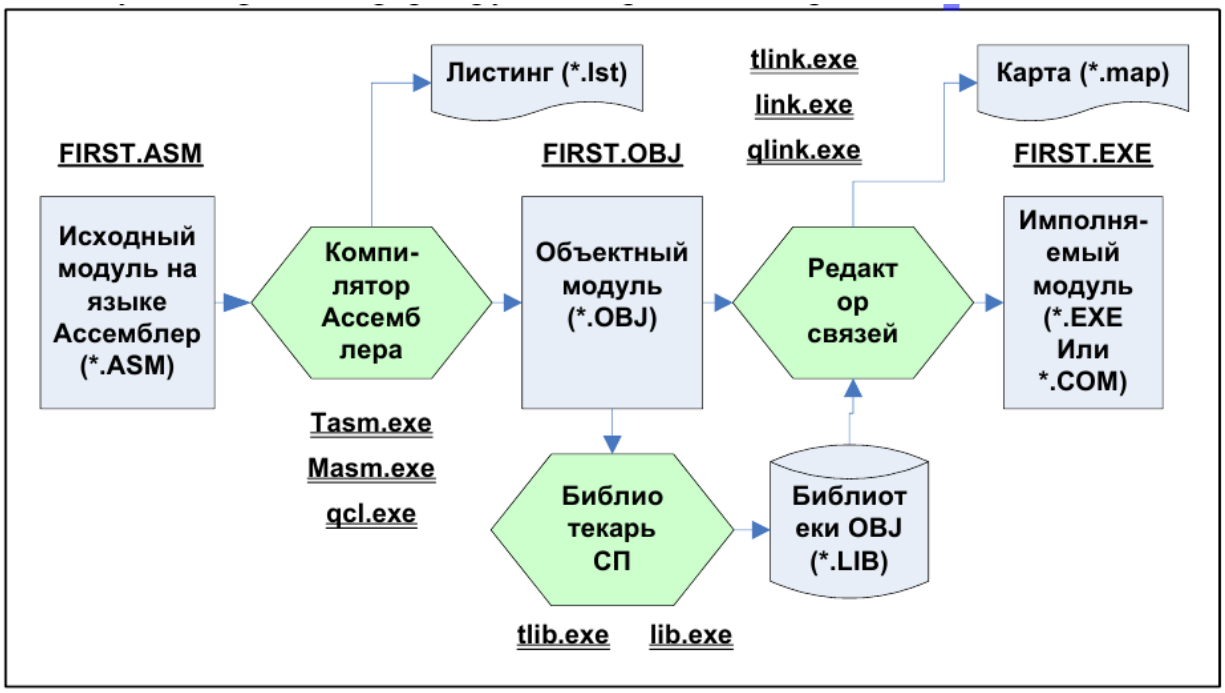


Рисунок 2.1– Процесс преобразования программ

На этом рисунке представлены основные компоненты систем программирования, которые участвуют в обработке программ. Кроме того, на рисунке выделены основные виды модулей и файлов, формируемых при такой обработке.

Данная технология формирования программ характерна практически для всех систем программирования, используемых в настоящее время. В тех случаях, когда мы работаем в интегрированной оболочке СП, мы можем не увидеть в явном виде промежуточных компонент, участвующих в такой обработке, однако они всегда присутствуют. Эти компоненты отмечены на рисунке двойным подчеркиванием.

Для лабораторных работ рекомендуются три разновидности систем программирования (они все есть на сайтах):

− Turbo Assembler (TASM) разных версий. Он включает: tasm.exe, tlink.exe, tlib.exe, td.exe и много других вспомогательных программ.

− Quick C and Quick Assembler (QC25). Он включает: qcl.exe, qlink.exe, lib.exe и много других вспомогательных программ. QC является интегрированной оболочкой, позволяющей выполнять все операции создания программ без переключения в командную строку. В QC встроен также текстовый редактор для подготовки исходных текстов программ;

− ASM Visual.

2.2.1 Ввод и редактирование текста программы Ассемблера.

Ввод и редактирования исходного текста программы, очевидно, самая простая операция процесса подготовки программ, однако она является, с другой стороны, достаточно трудоемкой и, если можно так сказать, “ошибкоемкой” (Исследования, проведенные очень давно, показывают, что на каждые 100 вводимых символов, даже самый опытный оператор, делает в среднем одну ошибку!). Поэтому выбор и освоение удобного текстового редактора серьезная задача. Текстовый редактор для Ассемблера должен обладать следующими свойствами:

− обеспечивать ввод в кодировках однобайтовыми символами (не UNICODE);

− не включать в текст специальные служебные символы (возможно скрытые) для форматирования текста (исключая символ табуляции и конца строки);

− обеспечивать ввод данных в кодировке ASCII (кодировка ДОС – для отладки программ) и кодировке ANSI (кодировка WINDOWS – для подготовки отчетов по ЛР и КР). Данный пункт относится к русским символам, так как они по разному кодируются в этих кодах;

− обеспечивать перевод из одной кодировки в другую (ASCII=>ANSI и ANSI =>ASCII);

− обеспечивать русификацию клавиатуры и шрифтов дисплея;

− просто развертываться и занимать немного места на диске;

− быть надежным в эксплуатации и легко осваиваться пользователями.

Учитывая сказанное, можно предложить следующие варианты текстовых редакторов для выполнения ЛР:

− любой текстовый редактор, удовлетворяющий приведенными выше требованиями;

− текстовый редактор ASM Editor for Windows;

− специальная программка перекодировки DOS<=>Windows (trans.exe);

− редактор NOTEPAD совместно с trans.exe;

− редактор в оболочке QC25 с trans.exe;

− редакторы в файловых менеджерах – DN и FAR совместно с trans.exe и русификатором.

Для редактирования текста в полноэкранном режиме и в режиме эмулятора ДОС необходим русификатор шрифта дисплея и клавиатуры. Его необходимо запустить до запуска текстового редактора. Рекомендуется русификатор RKM. Переклю-

чение раскладки клавиатуры по – умолчанию в нем выполняется клавишей – “правый Shift”.

В следующем разделе рассмотрим основные элементы подготовки программ на языке Ассемблер, в программе ASM Visual.

3 Инструкция по работе с программой ASM Visual

ASM Visual – IDE, представляет собой средства ведения проектов и редактор кода с подсветкой синтаксиса и автодополнением, включая ряд дополнительных модулей: взаимодействие с компиляторами, информирование об ошибках, работа с отладкой, инструменты рефакторинга, метрики кода, обратная связь с разработчиком. Комплекс представленных функциональных возможностей наряду с простыми пользовательскими интерфейсами, взаимодействующими с компилятором, позволяет отвлечься от выполнения вспомогательных задач, тем самым предоставляет возможность программисту сосредоточиться на решении собственно алгоритмической задачи и избежать потерь времени при выполнении типичных технических действий (например, вызове компилятора).

Поддерживаемые ОС: Windows XP, Vista, 2008, 7, 8, 10 и выше (для tasm16 только 32-bit)

Требования: .NET Framework 4.0 и выше.

3.1 Загрузка и установка

1. Загрузите дистрибутив со страницы https://gri-software.com/ru/download или получите дистрибутив у преподавателя.

2. Запустите дистрибутив **setupASMVisual.exe** и перед Вами появится окно Мастера установки (рисунок 3.1).

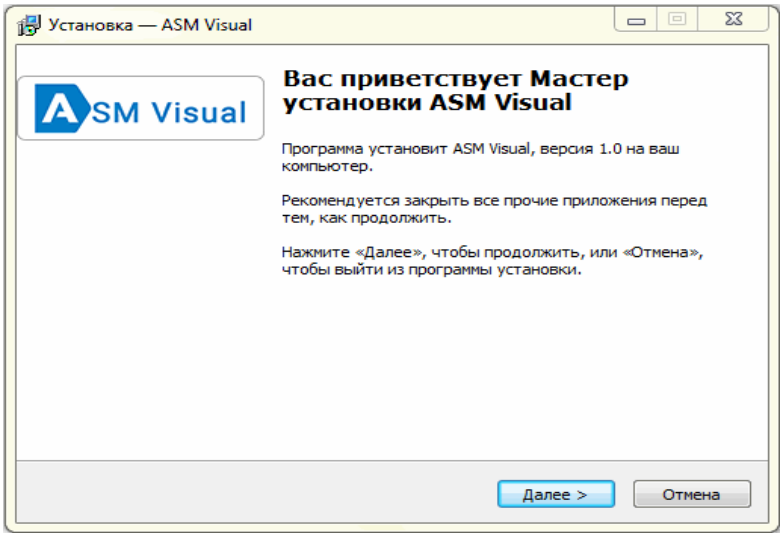


Рисунок 3.1 – Окно мастера установки

3. Выполните действия, рекомендуемые мастером установки.

По завершению установки, при первом запуске вы увидите следующее окно (рисунок 3.2).

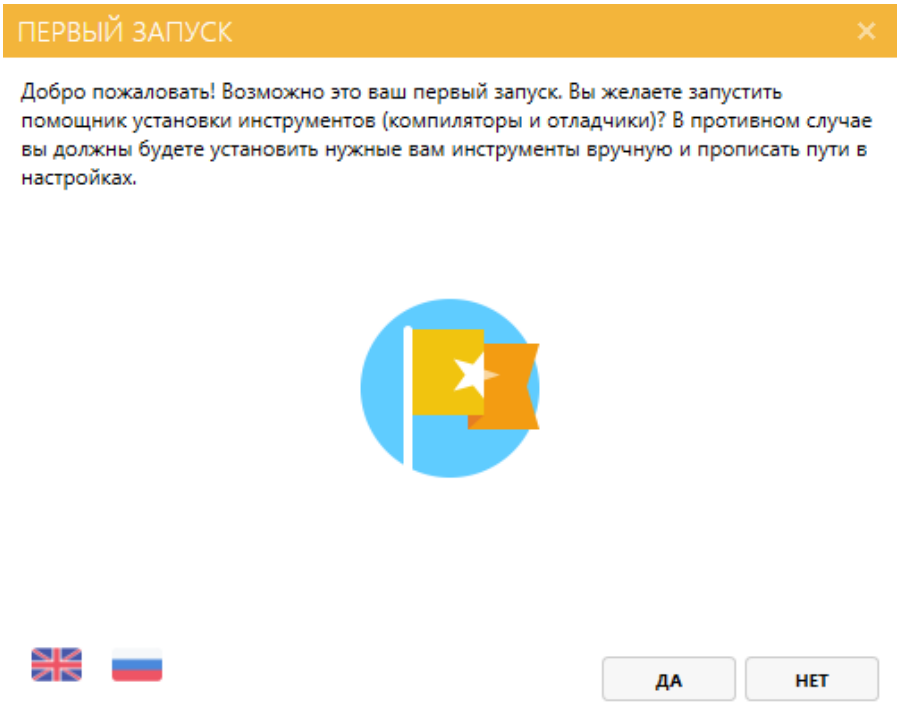


Рисунок 3.2 – Окно при первом запуске

Нажмите "Да", и запустится установщик инструментов (рисунок 3.3). Следуйте указаниям мастера установки:

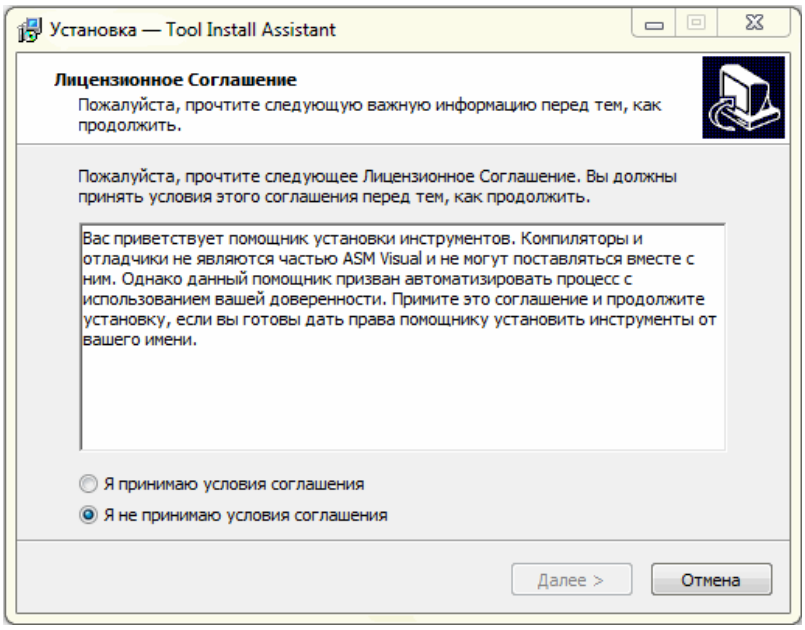


Рисунок 3.3 – Окно установщика инструментов

4. После окончания установки инструментов, либо при выборе "Нет", можно будет запустить непосредственно среду ASM Visual (рисунок 3.4) .

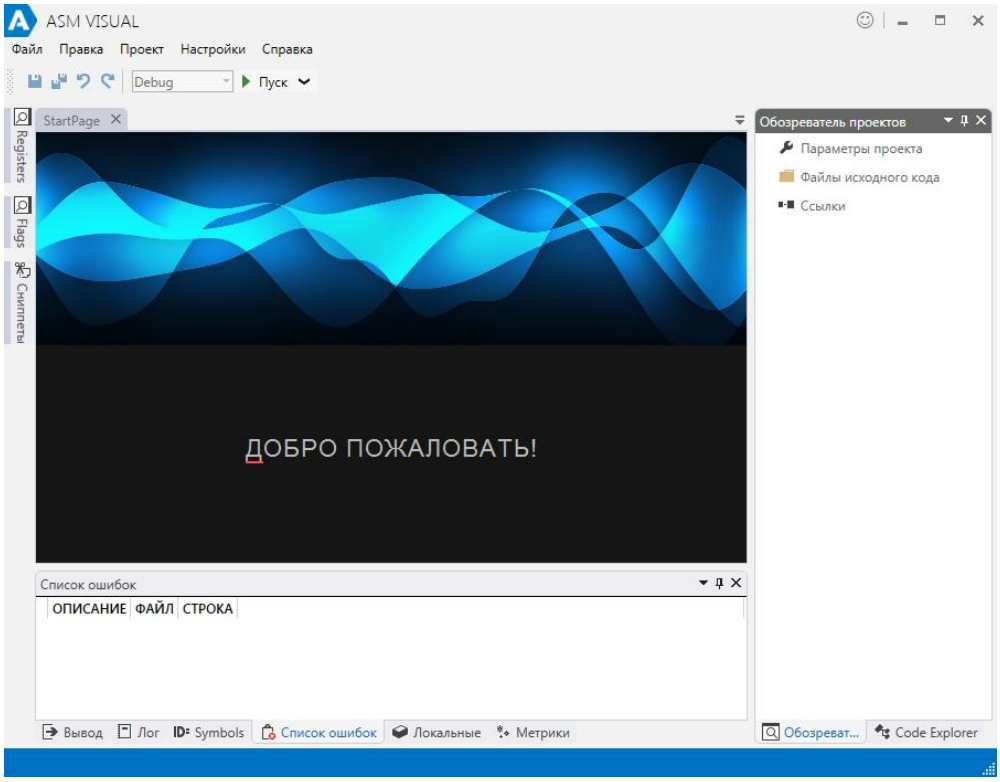


Рисунок 3.4 – Среда ASM Visual

Интерфейс программы состоит из главного меню, панелей работы с файлами проекта, основного редактора, панели отладки и информационной панели.

В окне редактирования поддерживается подсветка синтаксиса языка ассемблер.

Панель информации включает дополнительные окна для вывода информации, сообщений об ошибках при построении и выполнении приложения, окно вывода логов и окно вывода метрик кода разрабатываемой программы.

Так же на снимке экрана слева вы видите панель, предназначенную для вывода отладочной информации.

Справа расположена панель навигации.

Все положения и режим отображения панелей могут быть гибко настроены пользователем. Если, результат манипуляций с настройкой интерфейса вас не устраивает, вы всегда можете вернуться к первоначальному варианту, вызвав меню **Настройки > Внешний вид** и нажав кнопку **СБРОС МАКЕТА ОКОН**.

5. Для создания нового проекта воспользуйтесь меню "Файл"->"Создать новый проект" (рисунок 3.5).

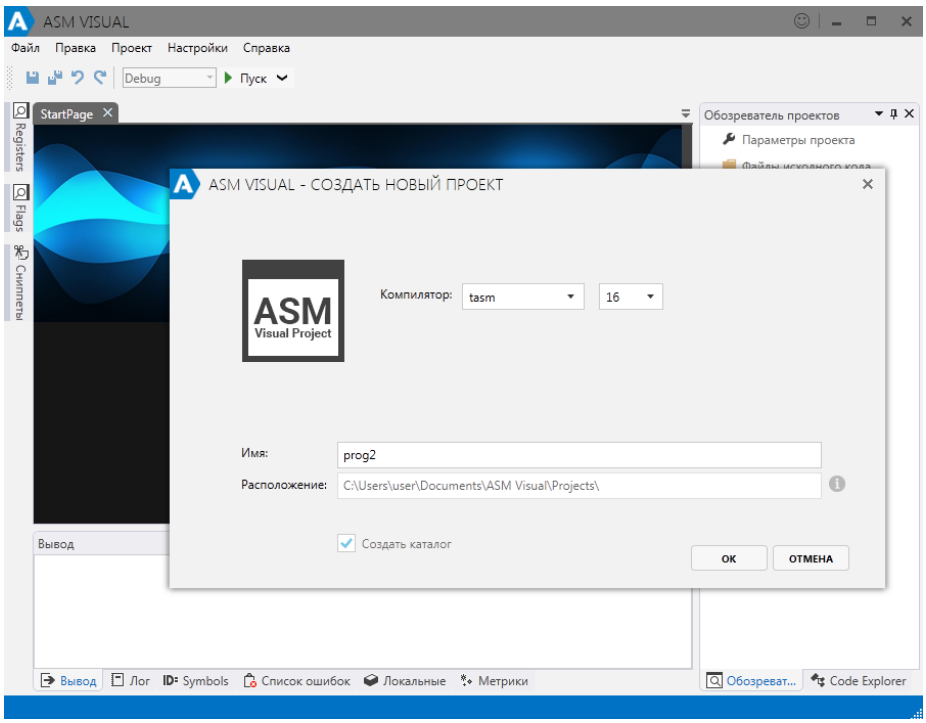


Рисунок 3.5 – Создание нового проекта в среде ASM Visual

В появившемся окне можно:

– выбрать используемый ассемблер;

– задать разрядность создаваемой программы;

– задать имя программы;

– указать месторасположение нового проекта.

В меню "Файл" можно открыть уже существующий проект или выбрать один из списка недавно открывавшихся проектов.

После открытия проекта можно изменить его свойства:

– выбрав в меню "Проект" подпункт "Свойства проекта";

– выбрав в "Обозревателе проектов "Параметры проекта" .

Окно "Свойства проекта" содержит три вкладки: «Проект», «Сборка», «Ещё». На вкладке «Проект» можно задать имя запускаемого файла вашего проекта (рисунок 3.6).

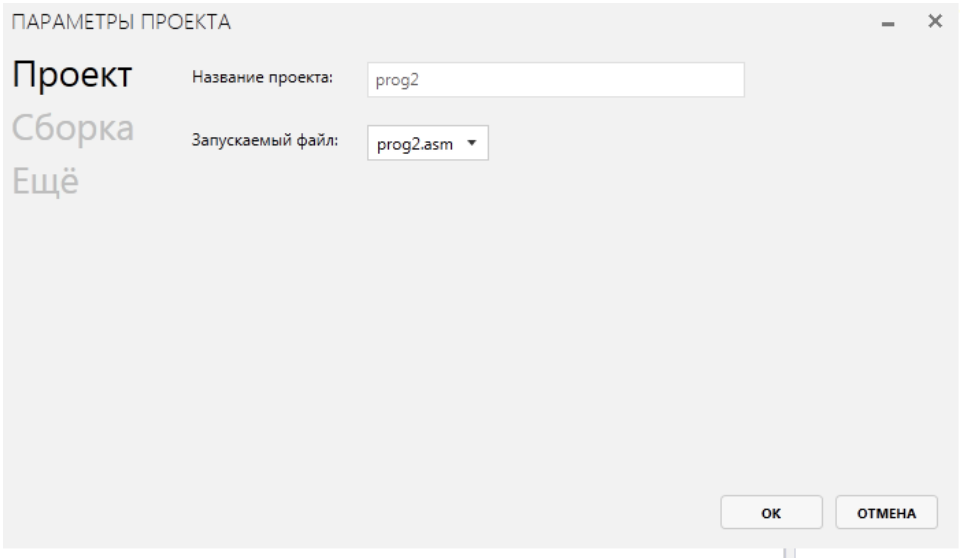


Рисунок 3.6 – Вкладка «Проект»

На вкладке «Сборка» задать параметры для используемых компилятора и сборщика программы (рисунок 3.7).

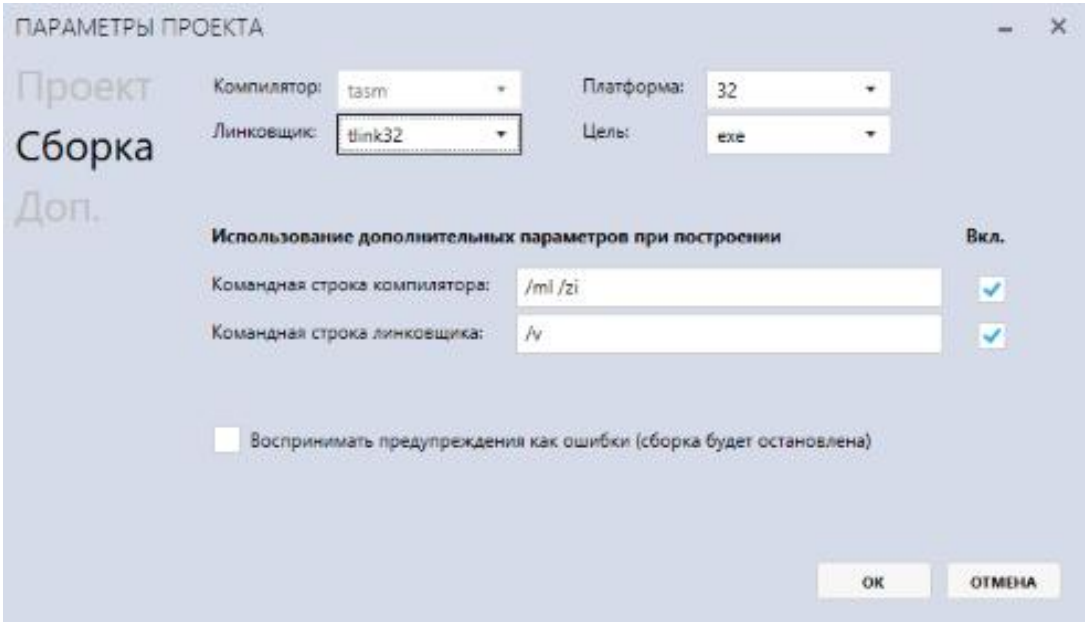


Рисунок 3.7 – Вкладка «Сборка»

На вкладке «Ещё» указать способ запуска программы и компилятора (рисунок 3.8).

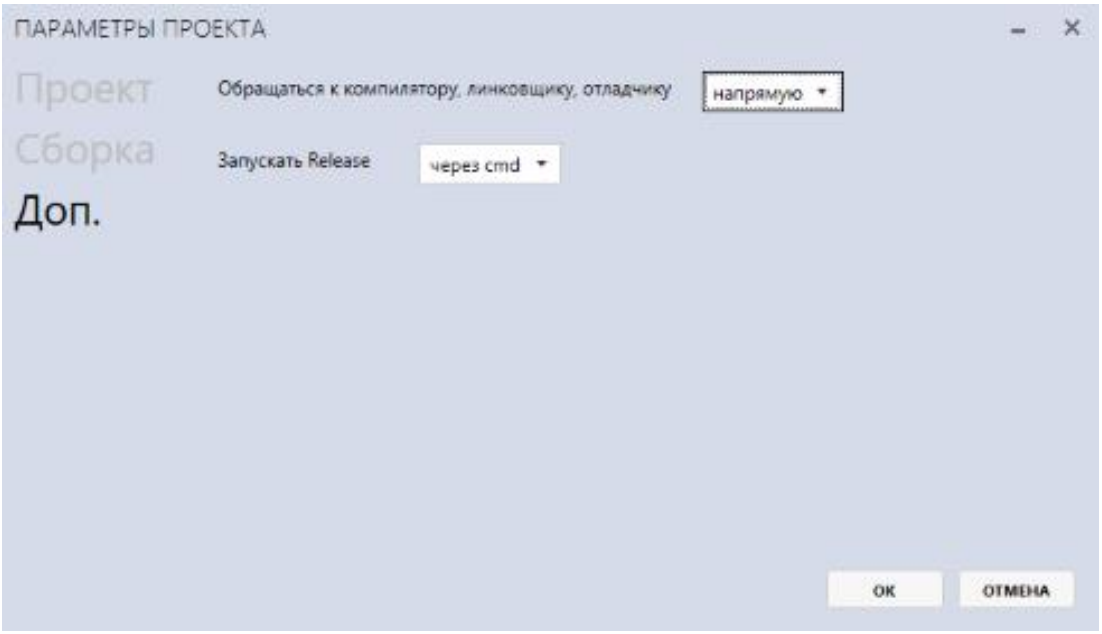


Рисунок 3.8 – Вкладка «Ещё»

Для работы с проектом, можно использовать "Обозреватель проектов" (рисунок 3.9).

В нём можно осуществлять навигацию по файлам проекта, добавлять в проект и удалять из него файлы.

Рисунок 3.9 – «Обозреватель проектов»

6. Работа с кодом.Среда ASM VISUAL поддерживает все основные функции редактора кода: включает автодополнение и всплывающие подсказки. Обеспечивает подсветку кода ассемблера и позволяет выполнить рефакторинг существую-

щего кода. Также можно использовать панель "Code Explorer" для обзора структуры существующего кода и навигации по ней (рисунок 3.10).

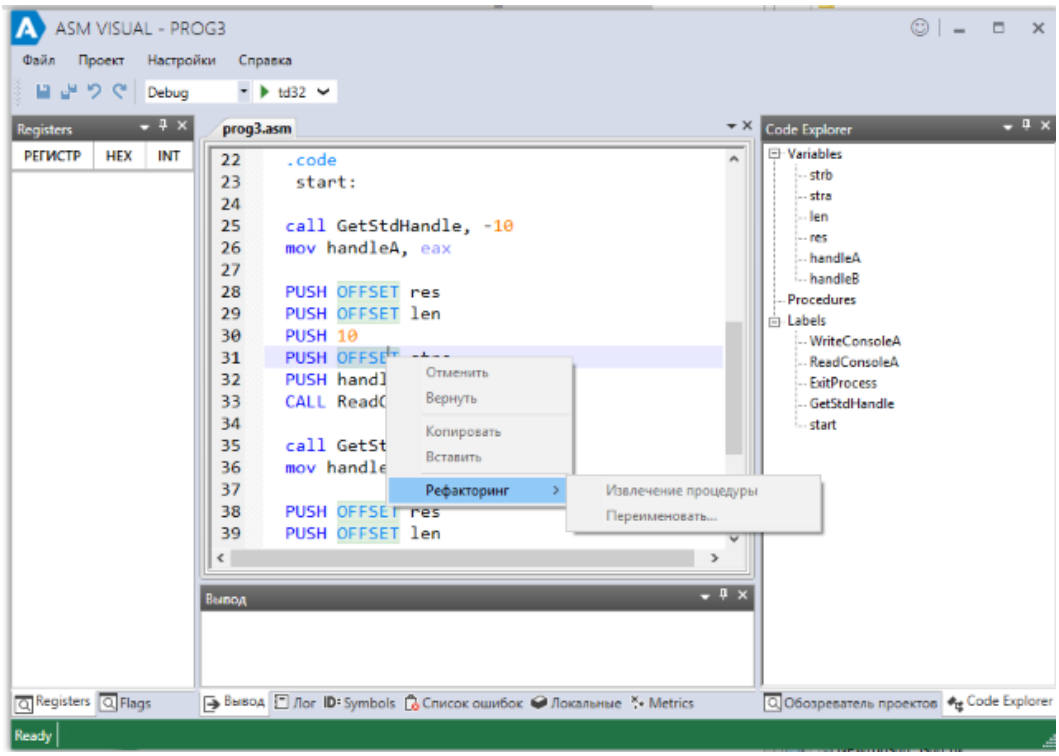


Рисунок 3.10 – Работа с кодом

7. Сборка программ. Компоновка программы – кульминация всего процесса создания программы, далее непосредственно осуществляется ее запуск. Среда ASM VISUAL позволяет выполнить компоновку и запуск программы как поочередно, так и сразу одним действием. Для выполнения операции сборки программы необходимо перейти к подменю «Проект» и выбрать пункт «Собрать проект» (рисунок 3.11).

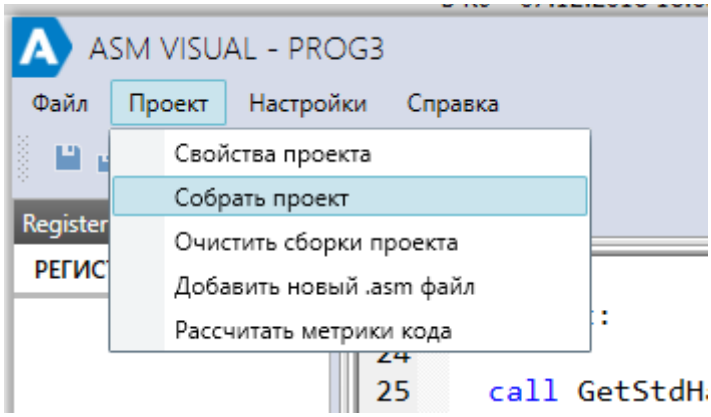


Рисунок 3.11– Сборка программы

Перед этим в тулбаре необходимо выбрать режим сборки (рисунок 3.12).

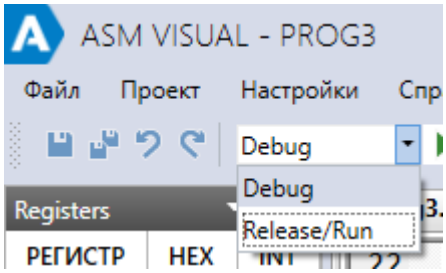


Рисунок 3.12– Выбор режима отладки

Если выбран режим «Debug», то в окошке рядом можно указать используемый отладчик (рисунок 3.13).

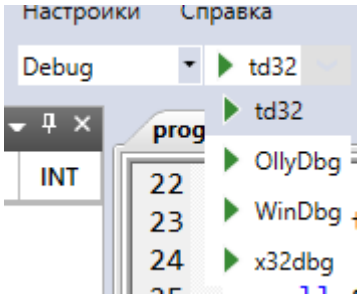


Рисунок 3.13– Выбор отладчика

Ход сборки проекта отображается в панели "Вывод" (рисунок 3.14).

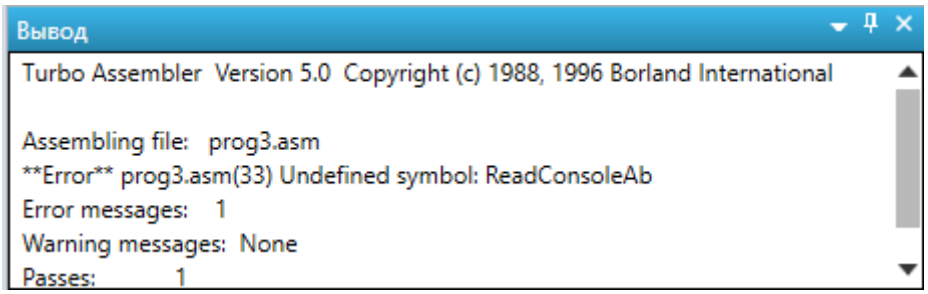


Рисунок 3.14 – Ход сборки проекта

Список ошибок можно посмотреть в соответствующей панели (рисунок 3.15).

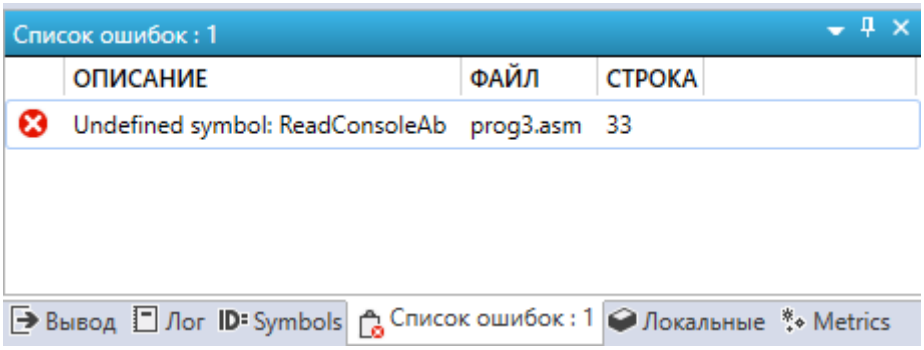


Рисунок 3.15 – Выдача списка ошибок

8. Отладка. ASM VISUAL предоставляет пользователю различные возможности отладки приложения. Пользователь может использовать такие внешние отладчики, как TD32, OllyDbg, WinDbg, x32dbg.

Для MASM и FASM возможно вести отладку непосредственно в IDE.

В случае необходимости пользователь может просмотреть значение регистров, памяти и локальных переменных, поставить точки останова и осуществить построчную отладку с учетом и без учета входа в подпрограммы Ассемблера (рисунок 3.16).

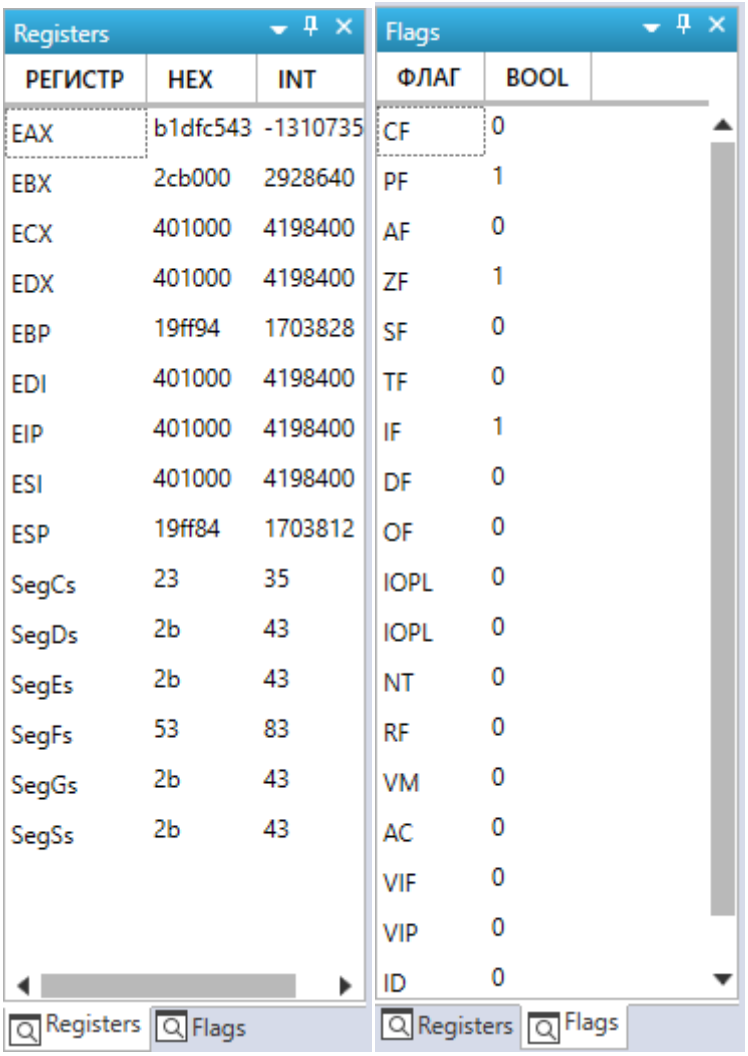


Рисунок 3.16 – Пример отображения содержимого регистров

Ход исполнения программы можно посмотреть в панели "Лог".

9. Настройка среды. Среда обладает всеми необходимыми настройками для удобной разработки программ. Настройка параметров программы ASM VISUAL осуществляется в меню «Настройки».

В первой вкладке настройки «Настройки Основные», можно установить ассоциации файлов (рисунок 3.17)

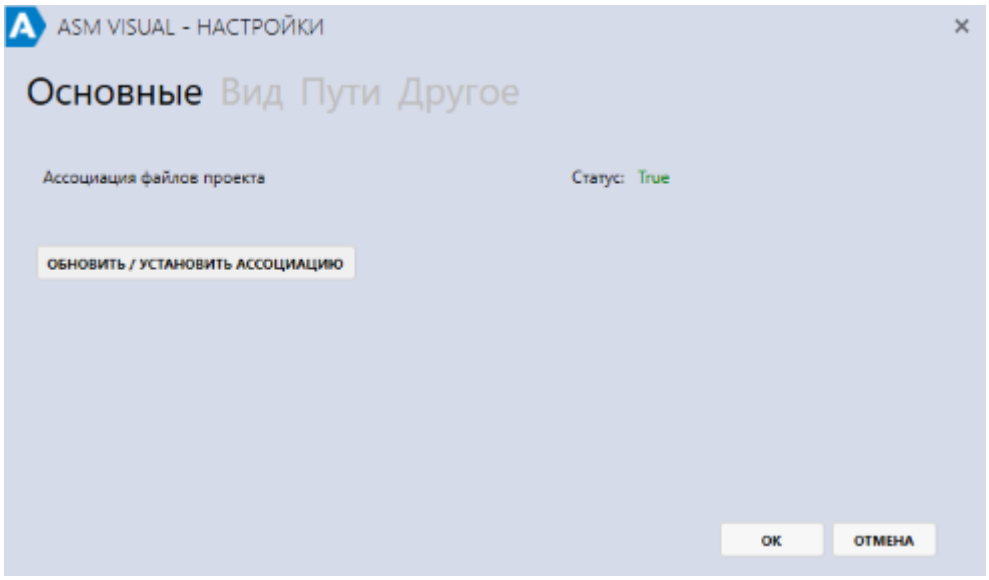


Рисунок 3.17 – Установка ассоциации файлов

Во второй вкладке можно сбросить настройки отображения окон и выбрать язык интерфейса (рисунок 3.18).

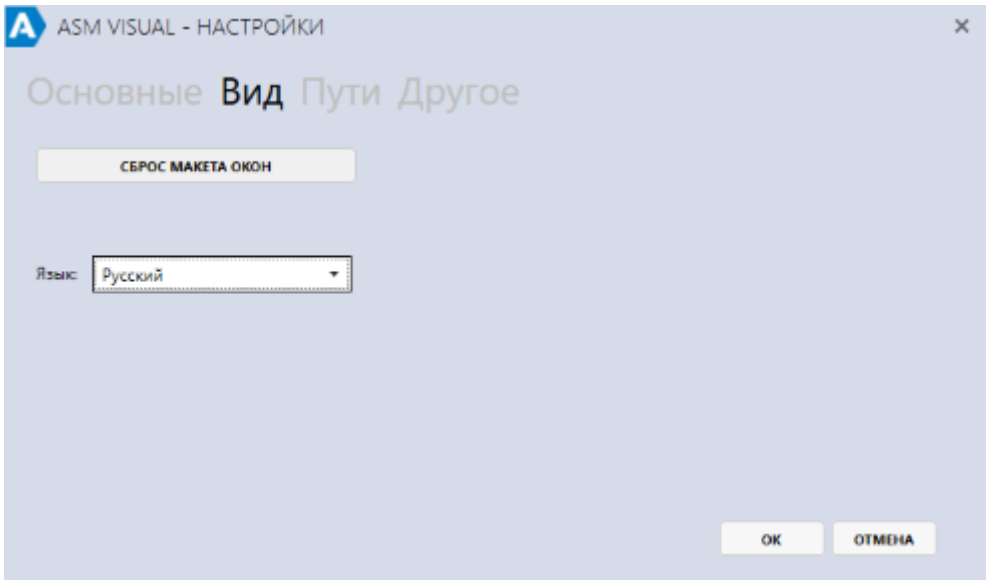


Рисунок 3.18 – Выбор языка интерфейса

На вкладке «Пути» окна «Настройки» можно задать пути для программ взаимодействующих с IDE (рисунок 3.19).

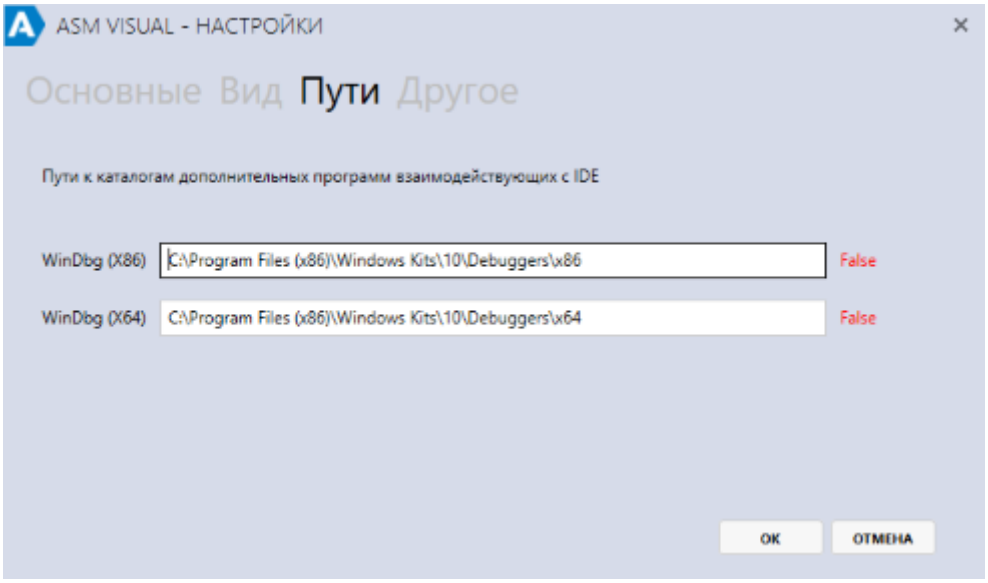


Рисунок 3.19 – Задание пути для программ взаимодействующих с IDE

В пункте «Настройка Темы» можно задать цвета основных элементов IDE (рисунок 3.20).

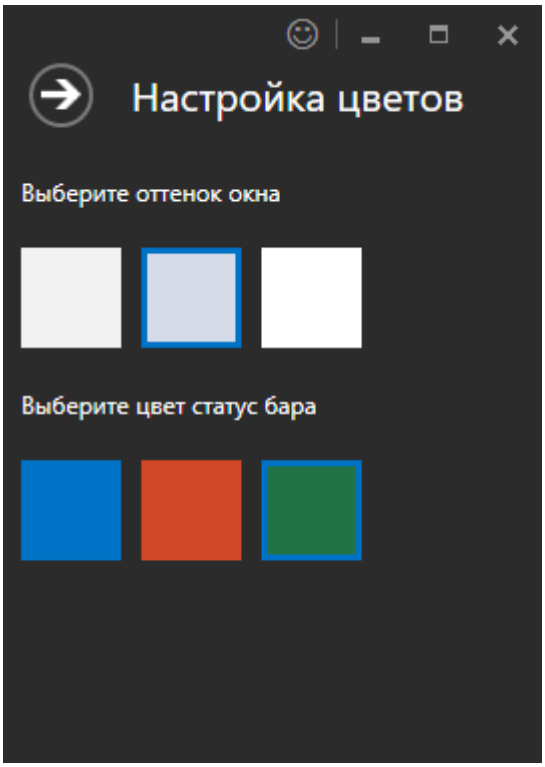


Рисунок 3.20 – Задание цвета основных элементов IDE

4 Выполнение программ в пошаговом режиме с использованием Turbo Debugger

Для отладки полученного exe-файла необходимо выполнить команду DOS Td имя_файла.exe.

Произойдет запуск отладчика и загрузка exe-файла в него. Появляется окно, показанное на рисунке 4.1.

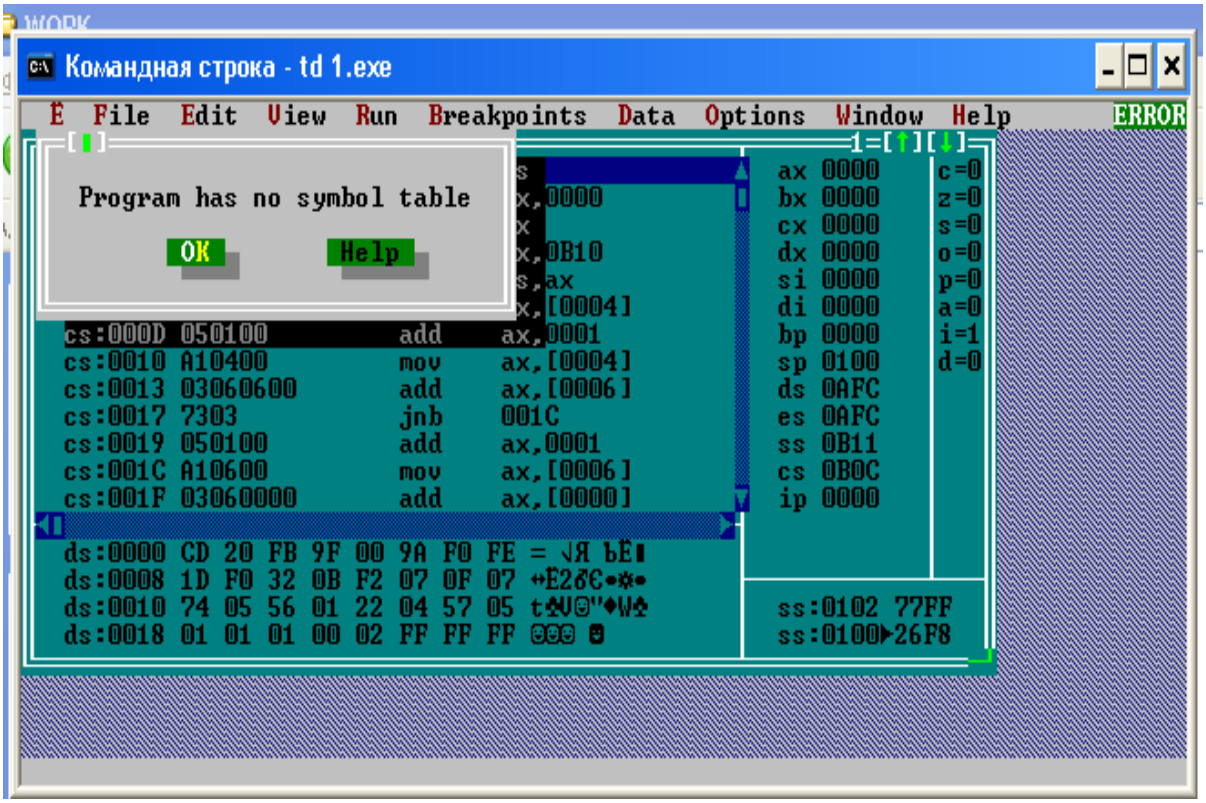


Рисунок 4.1– Главное окно отладчика Turbo Debugger

Рассмотрим области окна отладчика, показанные на рисунке 4.2.

Окно кода. В нем отображаются команды, их адреса и машинные коды.	Окно значений регистров	Окно значений битов регистра FLAGS
Окно данных. Отображает содержимое сегмента данных.	Окно стека. Отображает содержимое стека	

Рисунок 4.2 - Структура окна отладчика Turbo Debugger

На сообщение об отсутствии дополнительной отладочной информации следует нажать **ОК.** Появляется окно, показанное на рисунке 4.3.

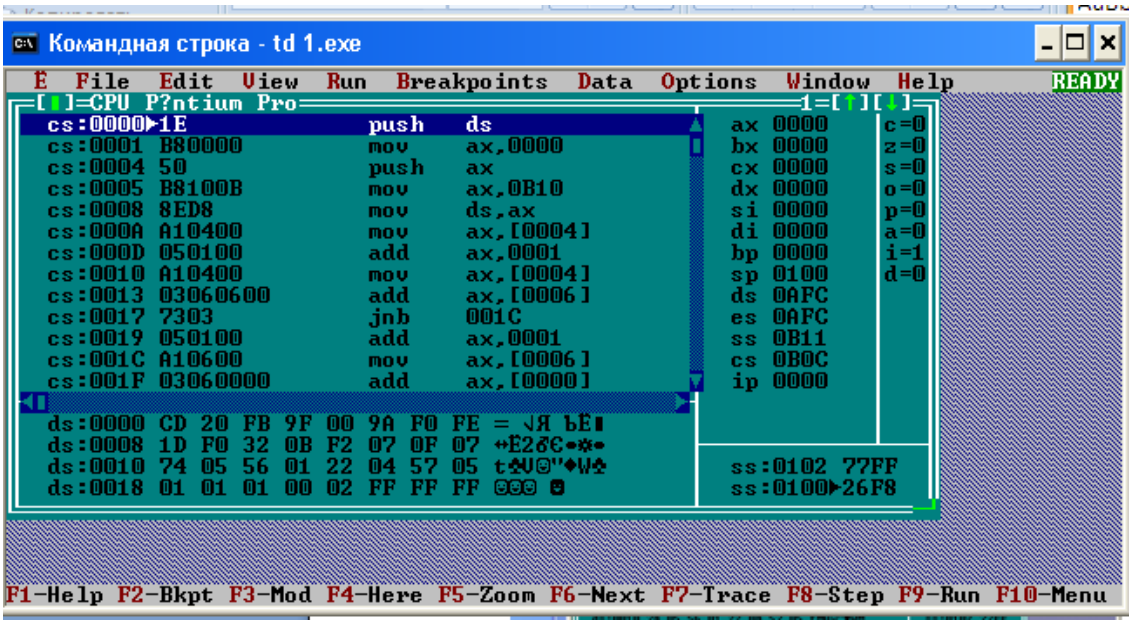


Рисунок 4.3– Начальное положение курсора в сегменте кода

Нажимая F7 выполняем по шагам первые команды. Включая команду MOV DS,AX. Появляется окно, показанное на рисунке 4.4.

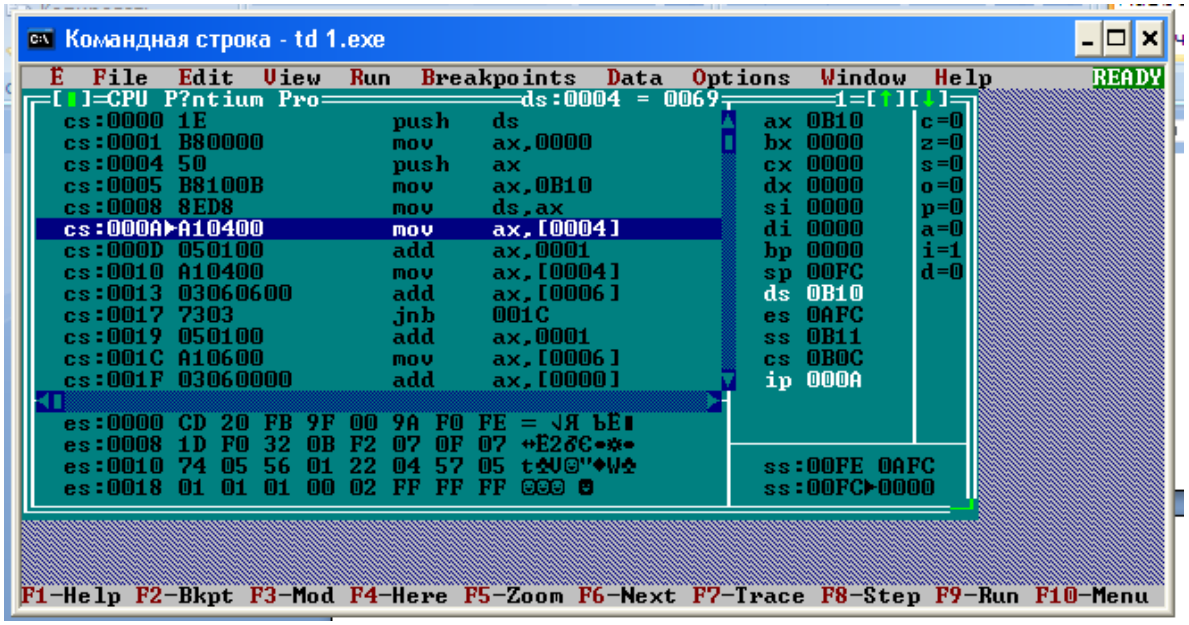


Рисунок 4.4– Вид окна после загрузки начального адреса в регистр DS

При открытии окна отладчика окно данных отображает начало PSP, так как DS в этот момент указывает именно туда. Но при перенастройке DS в команде MOV DS,AX это окно не меняет содержимого. Чтобы отобразить в нем сегмент

данных после выполнения этой команды, необходимо перейти в это окно (с помощью мыши или клавиши Tab). Появляется окно, показанное на рисунке 4.5.

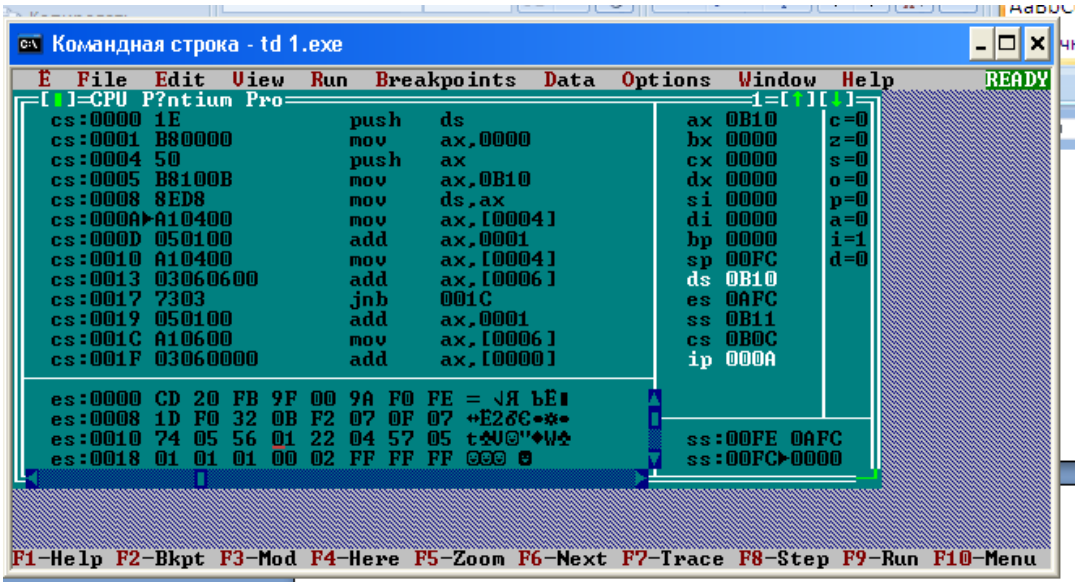


Рисунок 4.5– Переход в сегмент данных

Нажать комбинацию клавиш Ctrl+G. Появляется окно, показанное на рисунке 4.6.

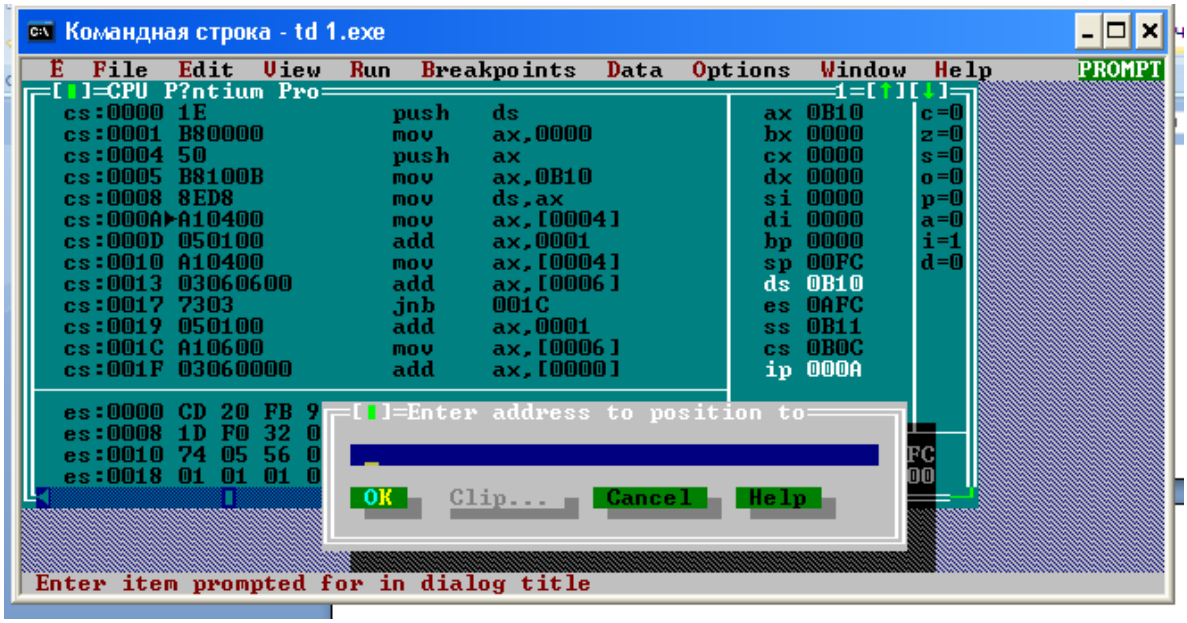


Рисунок 4.6 – Настройка сегмента данных на начало

В появившемся окне ввести ds:0. Появляется окно, показанное на рисунке 4.7.

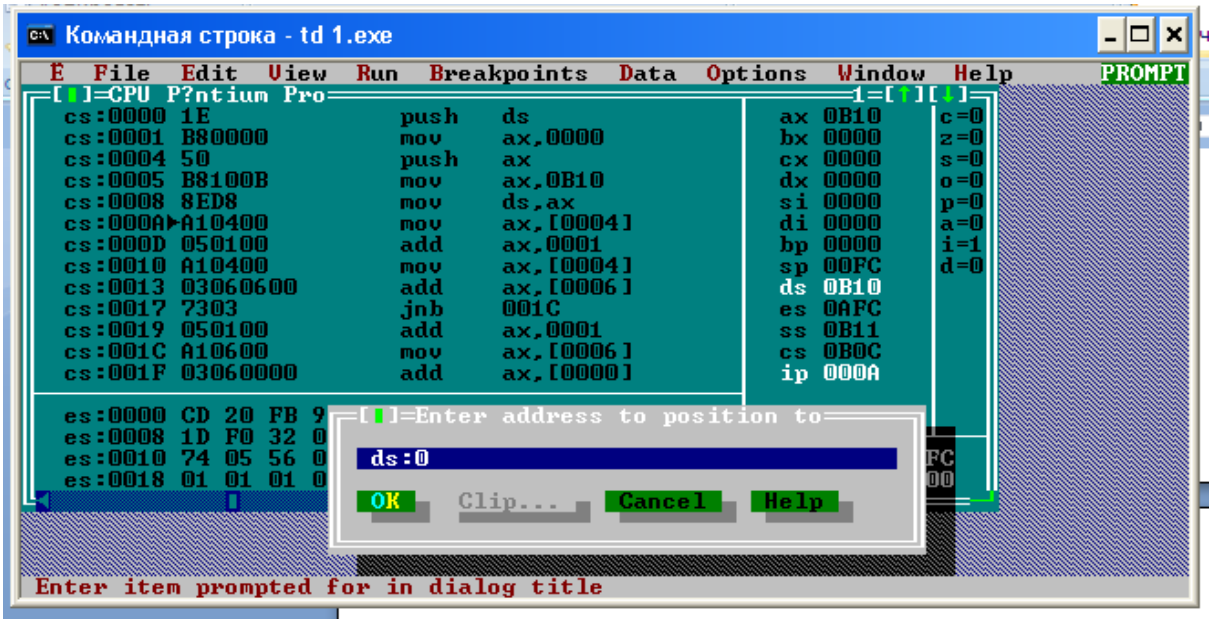


Рисунок 4.7 – Ввод начального адреса сегмента данных

Нажать ОК и вы увидите изменения в сегменте данных, там будут располагаться именно Ваши объявленные переменные. Появляется окно, показанное на рисунке 4.8.

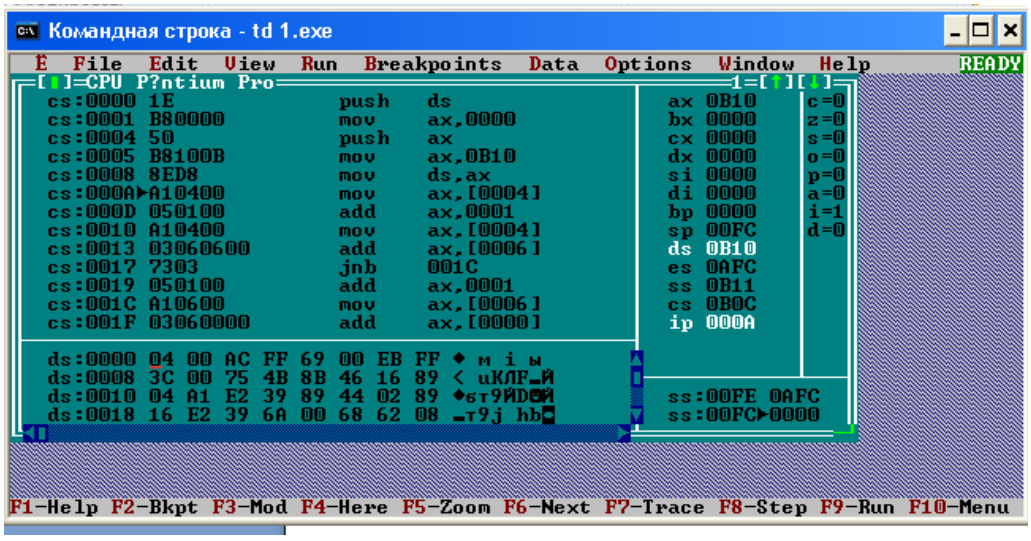


Рисунок 4.8 –Отображение нужных данных в сегменте

Возвращаемся в сегмент кода, где находится Ваша программа. Появляется окно, показанное на рисунке 4.9.

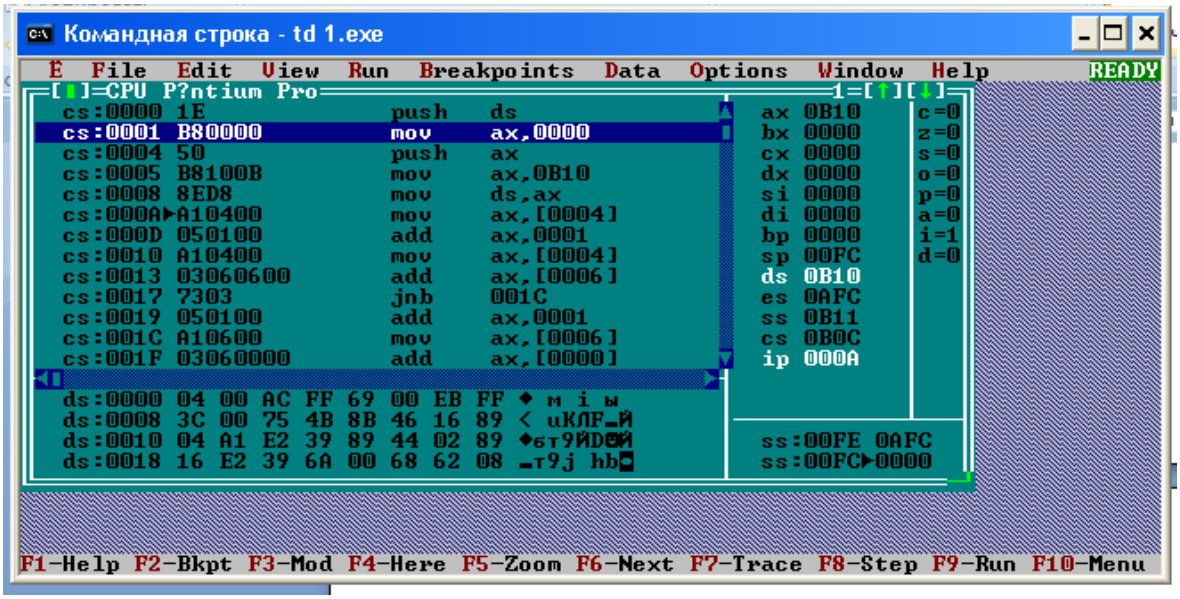


Рисунок 4.9– Возврат в сегмент кода

Для выполнения программы по шагам предназначены клавиши F7 и F8. Разница между ними заключается в обработке команды CALL. Нажатие F7 приводит к переходу на первую команду вызываемой подпрограммы, а нажатие F8 – к переходу к следующей за CALL команды вызывающей подпрограммы. Комбинация клавиш Ctrl+F2 позволяет заново начать выполнение программы с ее первой команды. Клавиша F4 позволяет выполнить программу до текущей строки.

В соответствующих окнах также можно просматривать содержимое регистров и стека.

5 Задание №1. Представление данных. Арифметико-логические операции

Цель работы: изучение архитектуры МП Intel 8086, изучение структуры простейшей ассемблерной программы, ознакомление с системой арифметико-логических команд процессора, организация вычислений на языке ассемблера.

5.1 Методические указания

При выполнении арифметико-логических команд наибольшего быстродействия и удобства программирования можно достичь за счет использования аккумулятора (регистра AX) для хранения промежуточных результатов. Например, вычисление выражения Y=X1+X2-X3 можно записать так

```
MOV AX, X1
ADD  AX, X2
SUB  AX, X3
MOV Y, AX.
```

При реализации операций деления необходимо помнить о том, что если результат не помещается целиком в регистре-приемнике (например, при делении 8B00h в регистре AX на 3 в регистре BL), возникает ошибка «деление на ноль» и программа аварийно завершается. Чтобы избежать подобных ситуаций, следует увеличивать размерность делимого и делителя. В нашем примере следует командой CWD расширить разрядность делимого и делить на BX, а не на BL (естественно, при этом BH должен быть равен 0).

Деление и умножение на степень двойки следует выполнять с помощью команд сдвига. Эти команды наиболее эффективны при использовании регистра AX.

5.2 Практическая часть

В практической части необходимо выполнить следующие действия:

– сформировать числовые значения в соответствии с индивидуальным заданием, определить минимальный формат представления исходных данных;

– по заданному алгоритму (согласно варианту задания) составить и выполнить программу для работы с данными.

Правильность разработки и выполнения программ арифметико-логической обработки данных контролируется путем ручной трассировки заданных алгоритмов с последующим сравнением результатов работы программ с результатами ручной трассировки.

5.3 Пример программы на языке Ассемблера

Приведем пример программы, которая находит сумму трех чисел: X1=8, X2=2, X3=-17 и сохраняет результат в ячейку памяти с именем RESULTAT (переменную RESULTAT).

```
.MODEL SMALL
; Объявляем все сегменты: стека, данных, кода
; Сегмент стека
Stseg SEGMENT STACK ‘stack’
DB 100 DUP (?)
Stseg ENDS
; Сегмент данных
Datseg SEGMENT ‘data’
X1 DB 8
X2 DB 2
X3 DB -17
RESULTAT DB ?
Datseg ENDS
; Сегмент кода
Codseg SEGMENT ‘code’
ASSUME CS: Codseg, DS: Datseg, SS:Stseg
; Программа
; Настроим DS на наш сегмент данных
Begin:
MOV AX, Datseg
```

```
MOV DS, AX
; Делаем вычисления
MOV AL, X1              ; Заносим переменную X1 в регистр AL
ADD  AL, X2             ; Складываем X1 и X2, результат в AL
ADD  AL, X3               ; Добавляем к сумме  переменную X3, результат в
AL
MOV RESULTAT, AL     ; Переписываем полученную сумму в память
Codseg ends
 End begin
```

5.4 Варианты заданий

5.4.1. Значения исходных данных, которые должны храниться в сегменте данных, приведены в таблице 5.1

Таблица 5.1– Исходные данные

№	X1	X2	X3	X4
1	-4	-12	25	51
2	-2	-56	33	24
3	-5	-30	48	55
4	-62	100	7	-60
5	-75	-84	18	26
6	-22	- 96	44	3
7	-68	-38	5	24
8	-106	-6	92	21
9	-4	-61	12	79
10	-6	-83	74	90
11	42	- 60	27	-77
12	99	12	-25	-60
13	28	55	-18	-34
14	32	50	-24	-67
15	49	52	-20	-66
16	38	45	-8	-37
17	4	12	-25	-51
18	2	56	-33	-24
19	5	30	-48	-55
20	62	100	-7	-60
21	-99	-12	25	60
22	68	38	-5	-24

5.4.2. Варианты алгоритмов программ приведены на рисунках 5.1 и 5. 2.

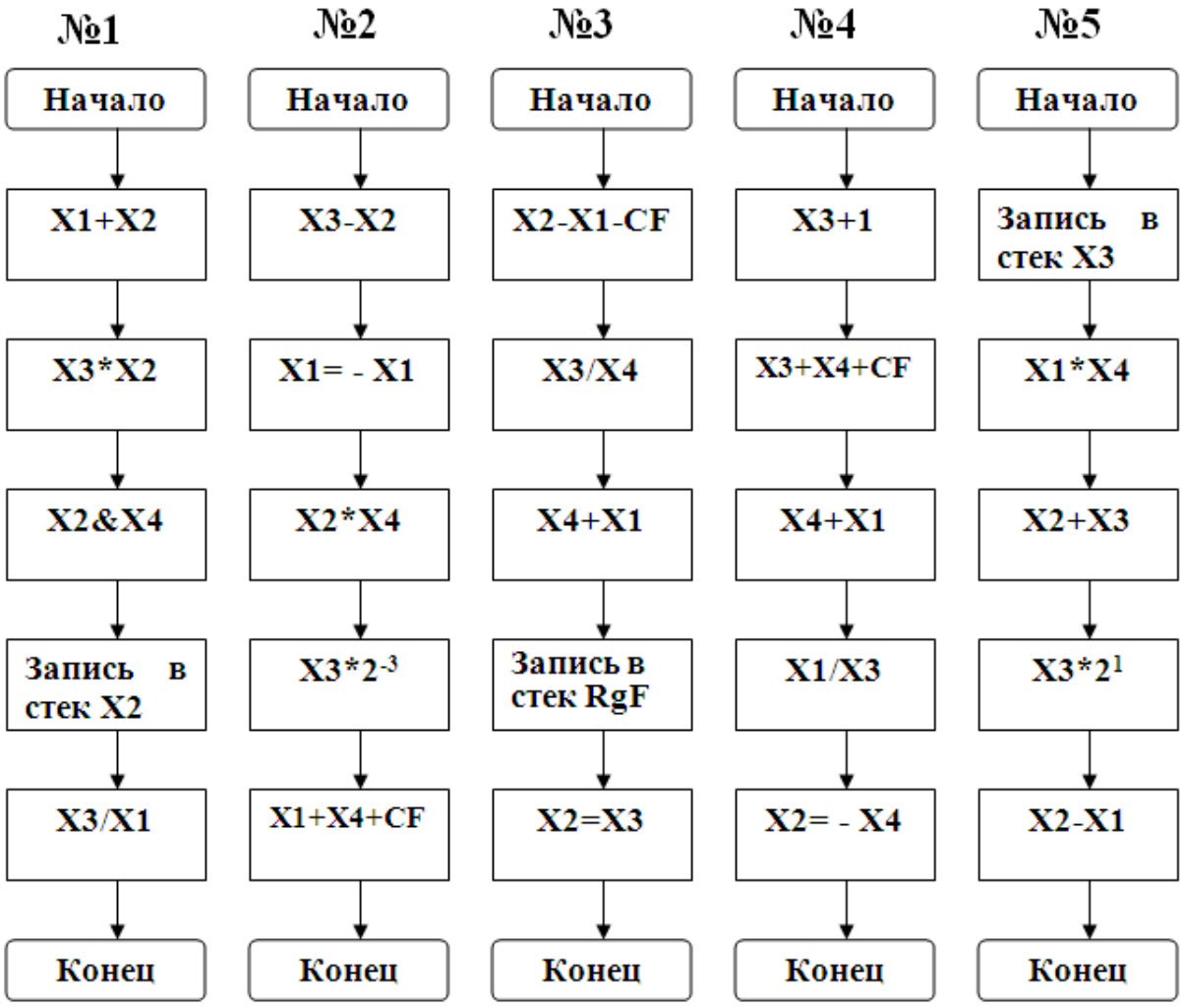


Рисунок 5.1– Варианты 1-5 алгоритмов программ

5.4.3. Порядок выполнения работы.

1. Определить исходные данные в соответствии с номером варианта.

2. Перевести значения величин X1-X4 в шестнадцатеричную систему счисления.

3. Провести ручную трассировку заданного алгоритма с использованием заданных исходных данных.

4. Составить программу заданного алгоритма в мнемокодах.

5. Проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

6. Оформить отчет по лабораторной работе, включив скрины, полученные в программе-отладчике.

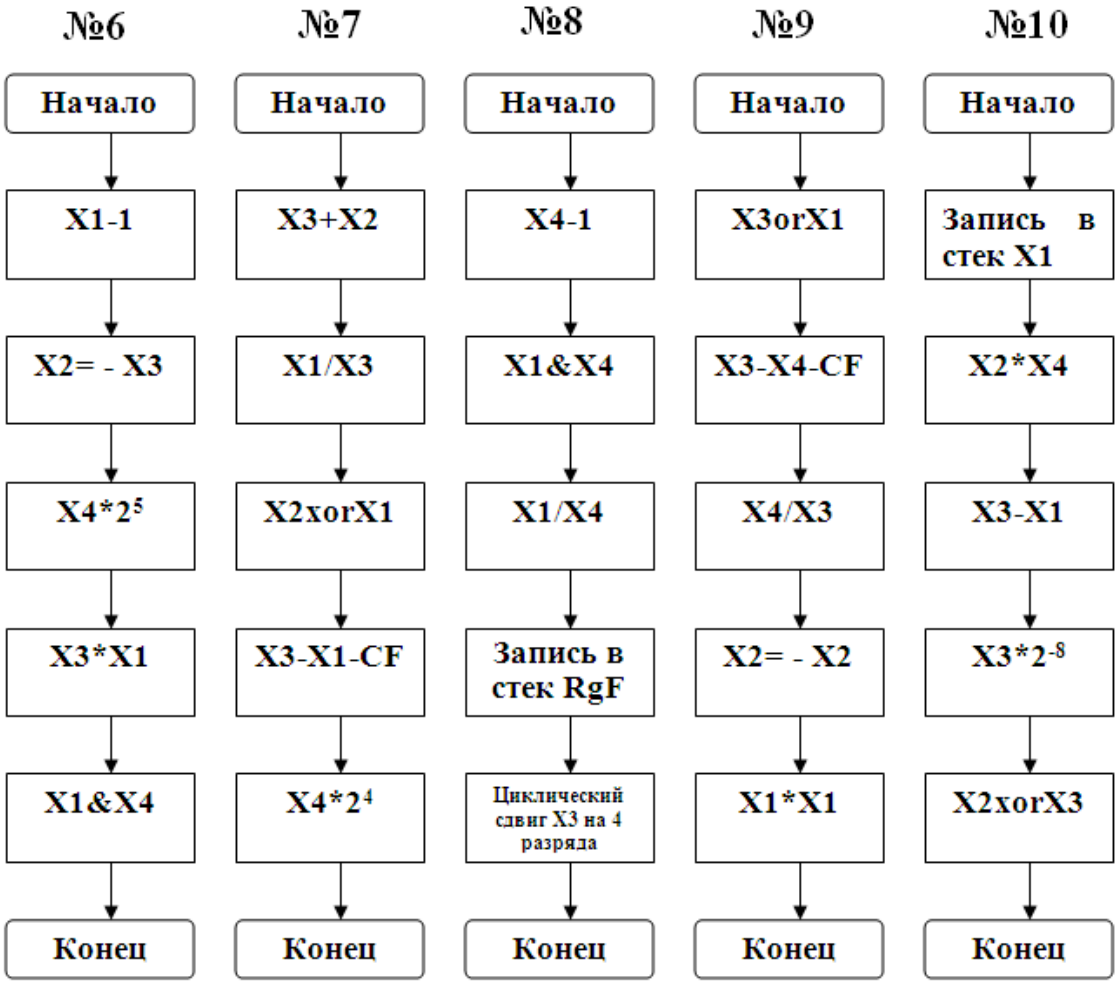


Рисунок 5.2 – Варианты 6-10 алгоритмов программ

5.4.4. Содержание отчета.

1. Текст задания.

2. Алгоритм программы.

3. Ручная трассировка

4. Текст программы на ассемблере с комментариями.

5. Результат машинной трассировки программы.

6. Для команд типа MOV в написанной Вами программе составить машинные форматы для этих команд и сравнить с полученными результатами в отладчике Turbo Debugger.

6 Задание №2. Исследование команд ветвлений и переходов, циклических и строчных команд МП i80x86 на примере программ на языке Ассемблера

Цель работы: изучение команд условного перехода, организации условных операторов и итеративных циклов.

6.1 Методические указания

Условный оператор с одноальтернативным ветвлением организуется в языке ассемблера следующим образом

```
Jусловие End_if1
Операторы
End_if1:
```

Обратите внимание, что в отличие от языков высокого уровня, операторы в данном случае выполняются при невыполнении условия. Например, оператор языка Pascal

```
If X1=0 then
    X1=5,
```

на языке Ассемблера будет выглядеть так

```
        CMP  WORD PTR X1, 0
        JNZ   End_if1
        MOV WORD PTR X1, 5
End_if1:  …
```

Как видно, в операторе If языка ассемблера условие заменяется на противоположное тому, что использовалось бы в языке высокого уровня.

Двухальтернативный условный оператор записывается в ассемблере так:

```
        Jусловие    If_1
        Операторы ветви Else
        JMP End_if2
If_1:
```

```
                Операторы ветви If_1
 End_if2:
```

Здесь условие менять на противоположное не надо, так как при его выполнении выполнится блок If, а при невыполнении – блок Else.

Циклы с предусловием записываются в языке ассемблера следующим образом

```
Cycle_1:    Jусловие    End_cycle_2
            Операторы
            JMP  Cycle_1
End_cycle_2:        …
```

Условие, как и для одноальтернативных условных операторов, нужно изменить по сравнению с языками высокого уровня на противоположное.

Пример: цикл языка Pascal

```
While X1<>0 do
        X1=X1 shr 1,
```

на ассемблере запишется так

```
 Cycle_1:   CMP  BYTE PTR X1, 0
            JZ      End_cycle_2
            MOV AL, X1
            SAR  AL, 1
            MOV X1, AL
            JMP   Cycle_1
 End_cycle_2:       …
```

Циклы с постусловием записываются на ассемблере так

```
Cycle_1:            …
            Тело цикла
            Jусловие    Cycle_1.
```

Для циклов языка Pascal условие необходимо изменить на противоположное. Что касается языка С, то в нем используется именно такая конструкция цикла с постусловием.

6.2 Практическая часть

В практической части необходимо выполнить следующие действия:

– сформировать исходные числовые значения;

– в соответствии с заданием составить алгоритм программы;

– написать программу на языке Ассемблера и выполнить ее.

Правильность разработки и выполнения контролируется путем ручной трассировки составленного алгоритма с последующим сравнением результатов работы программы с результатами ручной трассировки.

6.3 Варианты заданий

Для всех заданий исходное число (числа) хранится в двухбайтовой ячейке (ячейках) сегмента данных, результат необходимо сохранить в однобайтовую ячейку сегмента данных. Под словосочетанием «сохранить результат» понимается запись результата в однобайтовую ячейку в сегменте данных. Во всех заданиях следует использовать только итерационные циклы и условные операторы.

1. Подсчитать вес двоичного вектора и сохранить результат.

2. Если число состоит менее чем из 4 десятичных цифр, сохранить их сумму, иначе сохранить 0.

3. Найти и сохранить сумму десятичных цифр заданного числа

4. Найти и сохранить сумму четных десятичных цифр заданного числа.

5. Найти и сохранить сумму нечетных десятичных цифр заданного числа.

6. Найти и сохранить количество десятичных цифр в числе.

7. Найти максимальную цифру в числе и сохранить ее.

8. Найти и сохранить минимальную цифру в числе.

9. Найти и сохранить номер максимальной цифры в числе, считая, что младшая цифра имеет номер 1, и номера увеличиваются в сторону старших цифр.

10. Найти и сохранить номер минимальной цифры в числе, считая, что младшая цифра имеет номер 1, и номера увеличиваются в сторону старших цифр.

11. Сохранить 1, если число содержит данную цифру, иначе сохранить 0.

12. Найти и сохранить индекс первой со стороны младших цифр четной цифры числа, учитывая, что индекс вычисляется по правилам из задания №9.

13. Найти и сохранить индекс первой со стороны младших цифр нечетной цифры числа, учитывая, что индекс вычисляется по правилам из задания №9.

14. Найти сумму четных цифр заданного числа, которые делятся на четыре, и сохранить ее.

15. Найти сумму нечетных цифр заданного числа, которые делятся на три, и сохранить ее.

16. Найти сумму четных цифр заданного числа, которые делятся на три, и сохранить ее.

17. Определить, есть ли цифра два в заданном числе, и если есть, то удвоить результат и сохранить его.

18. Определить, есть цифра восемь в заданном числе, и если есть, то уменьшить результат в два раза и сохранить его.

19. Если число состоит более чем из 3 десятичных цифр, сохранить их сумму, иначе сохранить 0.

20. Найти произведение двух младших цифр заданного числа и сохранить его.

6.3.1. Порядок выполнения работы.

1. Сформировать исходные данные в соответствии с вариантом.
2. Составить алгоритм программы.
3. Провести трассировку заданного алгоритма.
4. Составить программу заданного алгоритма в мнемокодах.
5. Оформить отчет по лабораторной работе.
6. Проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

6.3.2. Содержание отчета.

1. Текст задания.
2. Алгоритм программы и ручная трассировка.
3. Текст программы на ассемблере с комментариями.
4. Машинная трассировка программы.

7 Задание №3. Изучение регистровых, загрузочных и стековых команд МП i80x86 на примере программ на языке ассемблера

Цель работы: изучение организации вызова подпрограмм в языке ассемблера, передачи параметров через стек и возврат значений из функции.

7.1 Процедуры, их назначение и применение

В программах на языке Ассемблер можно использовать процедуры. Процедуры позволяют сделать программу более наглядной и компактной.

При вызове процедур в нее необходимо передавать параметры. К сожалению, команда вызова процедура CALL не позволяет передавать параметры в процедуру явно. Для работы с процедурами выполняются следующие шаги:

− спроектировать процедуру, что включает как минимум: определение функционального назначения процедуры, перечень входных и выходных параметров процедуры;

− создать программу процедуры и оформить ее на языке Ассемблера;

− отладить процедуру автономно;

− написать в основной программе и отладить команды вызова процедуры.

Рассмотрим эти основные составляющие шаги для создания и применения процедур.

7.2 Описание процедур

Описание процедур выполняется на основе следующего синтаксиса

```
<Имя>   PROC [ FAR | NEAR ]
        <Команда>
        <Команда>
        …
        <Команда>
         RET[N | F]
```

<Имя> ENDP

Процедуры могут быть двух типов: NEAR (близкая) и FAR (далекая). Это указывается при описании процедуры в директиве PROC. По умолчанию процедура является NEAR. Параметры <имя> в операторах PROC и ENDP должны быть одинаковыми. Они отличаются от меток, на них нельзя передавать управление. Оператор RET (RETN , RETF) осуществляет корректный выход из процедуры (для NEAR и FAR соответственно). Таких операторов выхода может быть несколько, и они не обязательно должны стоять в конце процедуры. Но должен быть, по крайней мере, один выход из процедуры. Описание процедуры можно разместить практически в любом месте программы. Но лучше их размещать в конце программы. Число процедур не ограничивается. Не разрешается вложенное описание процедур, хотя допустим вызов вложенных процедур.

7.3 Параметры процедур и вызов процедур

Вызовы процедур выполняются командой CALL

; Вызов процедуры

CALL <имя процедуры> .

При вызове процедуры параметры не задаются. Параметры в/из процедуры входные и возвращаемые могут быть переданы следующими способами:

- через глобальные переменные программы;
- через регистры при вызове процедуры;
- через стек программы.

Передача через глобальные переменные выглядит так (глобальная переменная SIMV)

```
; Глобальная переменная
    SIMV  DB  'R'
      …
; Вызов процедуры
    CALL   SIMVOL
      …
```

```
; Процедура вывод символа
        SIMVOL   PROC
        MOV  DL , SIMV
        MOV  AH , 02H
        INT 21H
        RET
        SIMVOL   ENDP
```

Передача через регистры выполняется так (регистр DL):

```
; Вызов процедуры
         MOV  DL , SIMV
         CALL    SIMVOL
                 …
; Процедура вывод символа
        SIMVOL   PROC
        MOV  AH , 02H
        INT 21H
        RET
        SIMVOL   ENDP.
```

Передача через стек несколько сложнее для одного параметра и выполняется так

```
; Вызов процедуры
         MOV  DL , SIMV
         PUSH DX
         CALL  SIMVOL
         POP DX
              …
; Процедура вывод символа
        SIMVOL   PROC
        MOV  BP ,  SP
        MOV  DX , 2 + [BP]
        MOV  AH , 02H
```

```
        INT 21H
        RET
        SIMVOL   ENDP .
```

Так как команды CALL и RET используют стек, то приходится самостоятельно получать данные из стека (MOV DX , 2 + [BP]), предварительно получив значение BP на основе SP.

Как и процедуры, бывают короткие и дальние вызовы. Короткие вызовы мы уже рассмотрели выше, так как процедуры были по умолчанию объявлены как короткие. Длинный вызов может быть сделан так

```
; Поле для хранения длинного адреса
        PAR_1     DW ?
             …
;Подготовка длинного адреса
        LEA  BX ,  ADR_DL
        MOV PAR_1 , BX
        MOV PAR_1 + 2 , DS
; Задание параметра
        MOV DL , 'B'
        PUSH DX
; Дальний вызов
         CALL  DWORD PTR CS:[PAR_1]
         POP DX
             …
; Процедура для дальнего вызова
        ADR_DL   PROC   FAR
        MOV  BP ,  SP
        MOV  DX , 4 + [BP]
        MOV  AH , 02H
        INT 21H
        RETF
        ADR_DL   ENDP .
```

Для дальнего вызова процедуры можно воспользоваться следующей конструкцией оператора CALL

```
; Дальний вызов
    CALL FAR PTR ADR_DL.
```

7.4 Вложенные вызовы процедур

Из одной процедуры можно вызывать другие процедуры. Ограничений числа вложенных вызовов практически нет. Число вызовов ограничивается размером стека. Пример вложенных вызовов процедур приведен ниже

```
; Процедура перевода строки
    CLIR_1    PROC
    MOV  DL , 10
    CALL SIMVOL
    MOV  DL , 13
    CALL SIMVOL
    RET
    CLIR_1    ENDP .
```

В этой процедуре для вывода на экран символов перевода строки (10) и возврата каретки (13) дважды используется процедура SIMVOL.

Примечание. При вложенных вызовах нужно следить за регистрами и стеком. При необходимости регистры нужно сохранять. Число записей в стек должно четко соответствовать числу выборки данных из стека.

7.5 Пример программы с процедурами

Рассмотрим пример программы с процедурами. Ниже приводится исходный текст этого примера.

```
    MYCODE SEGMENT 'CODE'
    ASSUME CS:MYCODE, DS:MYCODE
    SIMV  DB  'R'
```

```
        PAR_1   DW ?
        begin:
; Загрузка регистра сегмента данных
        PUSH CS
        POP  DS
; Вывод символа
        MOV  DL , SIMV
        CALL SIMVOL
; Перевод строки
        CALL CLIR_1
; Подготовка длинного адреса
        LEA  BX ,  ADR_DL
        MOV PAR_1 , BX
         MOV PAR_! + 2 , DS
; Задание параметра
         MOV DL , 'B'
         PUSH DX
; Дальний вызов
        CALL  DWORD PTR CS:[PAR_1]
        POP DX
; Перевод строки
        CALL CLIR_1
; Ожидание завершения программы
        MOV AH, 01H
        INT  21H
; Выход из программы
        MOV AL, 0         ; выход из программы с возвращением errorlevel 0
        MOV AH, 4CH
; Активизация системной функции для завершения программы
        INT 21H
; Процедура перевода строки
```

```
CLIR_1     PROC
 MOV  DL , 10
 CALL SIMVOL
 MOV  DL , 13
 CALL SIMVOL
 RET
 CLIR_1     ENDP
; Процедура вывод символа
 SIMVOL PROC  NEAR
 MOV  AH , 02H
 INT 21H
 RETN
 SIMVOL ENDP
; Процедура для дальнего вызова
 ADR_DL PROC   FAR
 MOV  BP ,  SP
 MOV  DX , 4 + [BP]
 MOV  AH , 02H
 INT 21H
 RETF
 ADR_DL ENDP
; Конец сегмента
 mycode  ENDS
 END begin.
```

Результат работы данной программы очень простые:

R

B .

После вывода 2-х символов, каждый на отдельной строке, программа ожидает нажатия любой клавиши и затем заканчивается.

7.6 Работа со стеком

Передача параметров в подпрограммы языков высокого уровня обычно производится через стек. Поэтому вызов подпрограммы выглядит обычно следующим образом

```
PUSH пар_1
…
PUSH    пар_N
CALL    subrtn
ADD sp, N*2.
```

Рассмотрим, что происходит со стеком (рисунок 7.1).

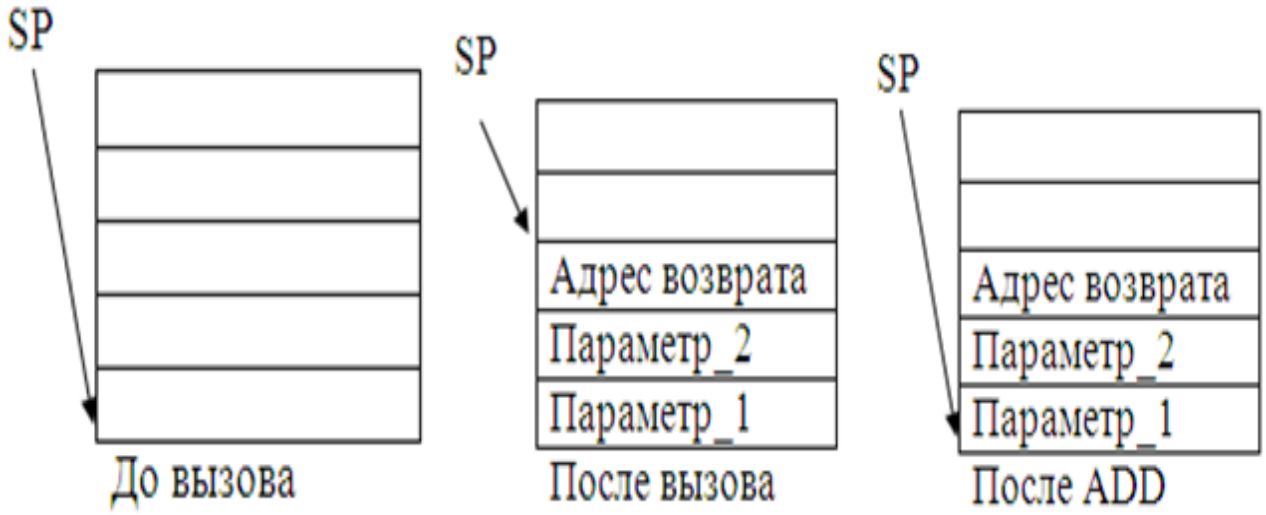


Рисунок 7.1 – Состояние стека

После вызова стандартная подпрограмма с параметрами, передаваемыми через стек, на языке ассемблера выглядит так

```
Subrtn          PROC            NEAR
                PUSH    BP      ; Сохранить старое значение BP
                MOV BP, SP
                …
                Тело подпрограммы
                …
                POP   BP      ; Восстановить значение BP
Subrtn          ENDP.
```

Рассмотрим стек после выполнения первых двух операторов подпрограммы (рисунок 7.2).

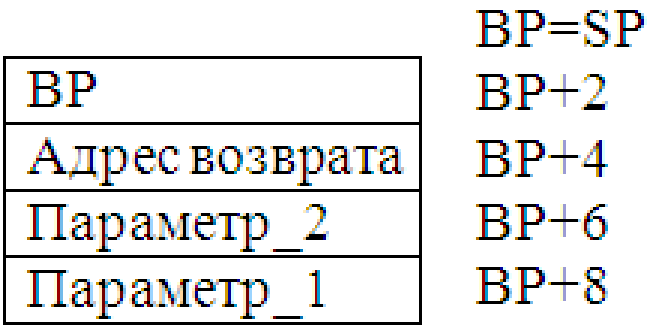


Рисунок 7.2 – Состояние стека после выполнения первых двух операторов подпрограммы

Видно, что первый параметр находится по адресу SS:[BP+8], а второй – по адресу SS:[BP+6].

Возврат значения в функциях языков высокого уровня осуществляется через регистр AX (AL, DX:AX), в зависимости от размера возвращаемого значения.

Приведем пример функции, складывающей значения параметров и возвращающей результат через AX

```
Sum   PROC      NEAR
      PUSH      BP
      MOV BP, SP
      MOV AX, SS:[BP+8]
      ADD  AX, SS:[BP+6]
      POP   BP
      RET
Sum   ENDP.
```

Заметим, что все, что было записано в стек внутри подпрограммы, должно быть извлечено из него, так как в противном случае команда RET возвратит управление не в ту точку, откуда была вызвана подпрограмма.

7.7 Практическая часть

Практическая часть работы включает выполнение следующих действий:

− в соответствии с индивидуальным заданием составление алгоритма основной программы и подпрограммы;

− по алгоритму составление и выполнение программы;

− контроль результатов работы программы.

Правильность разработки и выполнения контролируется путем ручной трассировки составленного алгоритма с последующим сравнением результатов работы программы с результатами ручной трассировки.

7.7.1. Варианты заданий.

Для всех заданий входные данные передаются в подпрограмму через стек, а результат возвращается через регистр AL. Для массивов входными данными являются адрес массива и число элементов в нем.

1. Найти максимум в заданном массиве.

2. Найти минимум в заданном массиве.

3. Найти сумму четных элементов массива.

4. Найти сумму нечетных элементов массива.

5. Подсчитать число ненулевых элементов массива.

6. Найти индекс минимального элемента в массиве.

7. Найти индекс максимального элемента в массиве.

8. Вычислить значение функции $F(x)=x^2+5x+7$

9. Вернуть 1, если числа являются сторонами треугольника Пифагора, иначе вернуть 0.

10. Вернуть 1, если точка лежит внутри окружности и 0 иначе.

7.7.2. Порядок выполнения работы.

1. Сформировать исходные данные в соответствии с вариантом.

2. Составить алгоритм подпрограммы и основной программы для решения поставленной задачи.

3. Провести ручную трассировку заданного алгоритма с использованием заданных исходных данных.

4. Составить программу заданного алгоритма в мнемокодах.

5. Оформить отчет по работе.

6. Проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

7.7.3. Содержание отчета.

1. Текст задания.

2. Алгоритм программы и ручная трассировка.

3. Текст программы на ассемблере с комментариями.

4. Машинная трассировки программы.

8 Задание №4. Изучение команд сдвига и приращений МП i80x86 на примере программ на языке ассемблера

Цель работы: изучение работы с массивами, организации арифметических циклов в языке ассемблера.

8.1 Методические указания

Работа с массивами возможна при использовании нескольких способов адресации: косвенной и индексной (как со смещением, так и без него).

Арифметические циклы на языке ассемблера организуются следующим образом

```
                MOV CX, число_итераций
Cycle1:         …
                Тело цикла
                …
                LOOP    Cycle1
```

Следует помнить, что данный цикл в ассемблере всегда имеет форму

```
For cx:=число_итераций downto 1 do
    Тело_цикла.
```

Если необходимо использовать другой цикл, например, for i:=1 to число_итераций do тело_цикла, нужно дополнительно использовать ячейку памяти или регистр для использования в роли i. Рассмотрим пример.

```
                MOV SI,1
                MOV CX, число_итераций
Cycle1:         …
    Тело цикла
    …
    INC  SI
    LOOP Cycle1.
```

Если необходим шаг цикла, отличный от единицы, следует вместо INC SI использовать ADD SI, шаг_цикла.

8.2 Практическая часть

Практическая часть работы включает выполнение следующих действий:

- в соответствии с индивидуальным заданием составление алгоритма программы обработки массивов, содержащих 10 элементов;
- по алгоритму составление и выполнение программы;
- контроль результатов работы программы.

Правильность разработки и выполнения контролируется путем ручной трассировки составленного алгоритма с последующим сравнением результатов работы программы с результатами ручной трассировки.

8.2.1. Варианты заданий.

Для всех заданий исходный массив хранится в сегменте данных, результаты необходимо сохранить в регистры общего назначения.

1. Найти логическую сумму положительных элементов массива и записать ее в Rg AX, и логическую сумму отрицательных элементов массива, записать ее в Rg BX (формат элементов массива - байт).
2. Найти логическую сумму отрицательных элементов массива и записать ее в Rg AX, и логическое произведение отрицательных элементов массива, записать ее в Rg BX (формат элементов массива - слово).
3. Найти сумму элементов массива, значение которых ≥ 3, и записать ее в Rg AL (формат элементов массива - байт).
4. Посчитать количество элементов массива, равных нулю, и записать их в Rg AX (формат элементов массива - слово).
5. Найти результат умножения максимального элемента массива на 25 и записать его в Rg BX, Rg CX (формат элементов массива - слово).
6. Найти результат деления числа 100 на минимальный элемент массива и записать в Rg BX (формат элементов массива - байт).

7. Найти количество положительных, нулевых и отрицательных элементов массива и записать в Rg AL, Rg BL и Rg DL соответственно (формат элементов массива - байт).

8. Найти минимальный положительный элемент массива (его номер и значение) и записать в Rg BX и Rg DX соответственно (формат элементов массива - слово).

9. Найти арифметическую сумму элементов массива, значения которых лежат в интервале -10 < X(i) < 20, и записать ее в Rg DH (формат элементов массива - байт).

10. Найти количество элементов массива, имеющих отрицательное значение и четный номер, и записать в Rg AX (формат элементов массива - слово).

11. Найти элемент массива, имеющий максимальное абсолютное значение, и записать в RgCX (формат элементов массива - байт).

12. Найти отрицательный элемент массива, имеющий максимальное абсолютное значение, и записать в Rg DX (формат элементов массива - слово).

13. Найти количество элементов массива, значения которых лежат в интервале - 20<X(i)< 50, и записать в RgBX (формат элементов массива - слово).

14. Найти сумму модулей элементов массива и записать в RgDX (формат элементов массива - байт).

15. Найти результат умножения индекса минимального элемента на 55 и записать его в RgBX (формат элементов массива - байт).

16. Найти произведение элементов массива, значения которых лежат в интервале -10 < X(i) < 10, и записать ее в Rg BL (формат элементов массива - байт).

17. Найти минимальный четный элемент массива (его номер и значение) и записать в Rg BLи Rg BH соответственно (формат элементов массива – байт).

18. Найти максимальный нечетный элемент массива (его номер и значение) и записать в Rg BX и Rg AX соответственно (формат элементов массива - слово).

8.2.2. Порядок выполнения работы.

1. Сформировать исходные данные в соответствии с вариантом.

2. Составить алгоритм программы для решения поставленной задачи.

3. Провести трассировку заданного алгоритма с использованием заданных исходных данных.

4. Составить программу заданного алгоритма в мнемокодах.

5. Оформить отчет по лабораторной работе.

6. Проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

8.2.3. Содержание отчета.

1. Текст задания.

2. Алгоритм программы и ручная трассировка.

3. Текст программы на ассемблере с комментариями.

4. Машинная трассировка программы.

9 Задание №5. Использование команд вызова прерываний

В данной лабораторной работе необходимо написать и отладить ассемблерную программу, выводящую на экран заданную информацию.

Управление программой осуществляется с помощью заданных клавиш. Список заданий к лабораторной работе приводится в конце описания. Для решения поставленной задачи рекомендуется использовать встроенные функции DOS и BIOS. Далее приводится описание некоторых из этих функций (далеко не всех). Рассмотренных ниже функций вполне достаточно для выполнения любого из заданий к данной лабораторной работе.

9.1 Прерывания DOS для работы с клавиатурой

9.1.1. Рассмотрим пример проверки нажатия клавиши.

```
ah=1                ; функция 1  21-го прерывания
int 21h.
```

Ввод символа (его ASCII-кода) из буфера клавиатуры. Если буфера пуст (никакая клавиша не нажата), подпрограмма ждет нажатия клавиши. Есть эхо-отображение вводимого символа на экран и проверка на нажатие Ctrl/C.

В результате в регистре al возвращается ASCII-код символа. Если нажата несимвольная клавиша (например F2) в al вернется ноль. При этом чтобы прочитать расширенный скенкод этой клавиши надо повторно вызвать int 21h с функцией 1.

Пример. Проверить нажата ли клавиша q.

```
mov ah,1
int 21h
cmp al,'q'          ; сравнение полученного кода с кодом q
jne m1              ; переход на метку m1 если "не равно".
```

9.1.2. Рассмотрим пример выхода из программы по нажатию какой-либо клавиши.

```
ah=7                ; функция 7  21-го прерывания
```

```
int 21h.
```

Аналогична функции 1, но нет проверки на Ctrl/C и эхо-отображения.

Пример. Выйти из программы по нажатию любой клавиши.

```
mov ah,7
int 21h          ; здесь программа будет "висеть" пока не будет нажата ка-
```

кая-либо клавиша.

```
mov ah,4ch       ; выход в NORTON
int 21h.
```

9.1.3. ah=8 ; функция 8 21-го прерывания

```
int 21h
```

Аналогична функции 7, но есть проверка на Ctrl/C.

9.1.4. ah=6 ; функция 6 21-го прерывания

```
dl=0ffh
int 21h
```

Если в буфере есть символ, то его код возвращается в регистре al, а флаг процессора zf устанавливается в 0. Если в буфере нет символа, zf устанавливается в 1, а в al-"мусор". Т.е. в отличии от предыдущих функций здесь не ждут нажатия клавиши (так называемый ввод без ожидания). Эхо-отображения и проверки на Ctrl/C нет.

Пример. Проверка на нажатие Esc (код Esc = 1bh).

```
mov dl,0ffh
mov ah,6
int 21h
jz m1        ; никакая клавиша не нажата (zf=1)
or al,al
jz m2        ; нажата несимвольная клавиша (в al из int 21h вернулся 0)
cmp al,1bh
je m3        ; нажата Esc.
```

9.1.5. ah = 0ah

dx = начальный адрес буфера в оперативной памяти (в сегменте данных для EXE-программ)

int 21h.

Ввод строки символов с клавиатуры в созданный заранее буфер. Вводимая строка набирается на клавиатуре и заканчивается нажатием клавиши ENTER. В результате в буфер помещается следующая информация:

- байт 0 – ожидаемая длина строки;
- байт 1 – фактическая длина строки;
- байт 2 и далее – строка, заканчивающаяся ASCII-кодом клавиши ENTER (0dh).

Примечание. Рассмотренные выше функции не исчерпывают возможности DOS при работе с клавиатурой. В частности, для ввода с клавиатуры широко используется функция **3Fh** прерывания **21h** с дескриптором равным нулю (дескриптор задается в регистре bx).

9.2 Прерывания BIOS для работы с клавиатурой

9.2.1. Рассмотрим пример проверки на нажатие клавиши

ah=0

int 16h.

Чтение символа из буфера. Если клавиша символьная, в al возвращается ее ASCII-код, а в ah-ее скен-код. Если клавиша не символьная, в al возвращается 0, а в ah - ее расширенный скен-код. Если в буфере нет символа подпрограмма ждет нажатия клавиши. Нет проверки на Ctrl/C и эхо- отображения.

Пример. Проверка на нажатие клавиши "стрелка-вверх". Это несимвольная клавиша и её расширенный скен-код = 48h.

```
mov ah,0
int 16h
or al,al
jnz m1                    ; нажата символьная клавиша
```

```
cmp ah,48h
je m2 .                    ; нажата "стрелка-вверх".
```

Примечание. Прерывание int 16h имеет еще две функции: 1 и 2. Обе эти функции также предназначены для работы с клавиатурой.

9.3 Прерывания DOS для работы с экранов

9.3.1. Рассмотрим пример ввода символа в текущую позицию курсора

```
ah=2
dl= ASCII-код символа
int 21h.
```

Выводит заданный символ в текущую позицию курсора. Курсор после вывода смещается на позицию вправо. Коды 7,8,0ah и 0dh на экран не выводятся, а управляют перемещением курсора:

- 8 – на символ влево;
- 0ah – на строку вниз;
- 0dh – на начало строки;
- 7 – звонок.

Пример. Вывести букву A в текущую позицию курсора.

```
mov ah,2
mov dl,'A'
int 21h.
```

9.3.2. Рассмотрим пример ввода строки символов в текущую позицию курсора

```
ah = 9
dx = адрес начала строки в оперативной памяти (вернее смещение относительно базового адреса сегмента данных)
int 21h.
```

Выводит начиная с текущей позицией курсора строку символов из оперативной памяти. Конец строки задается символом $. Коды 7,8,0ah и 0dh являются управляющими.

Пример.

```
data segment para public 'data'
…
stroka  db 'Я, ребята, студент',0dh,0ah,'$'
…
data ends
…
mov ah,9
mov dx,offset stroka
int 21h.
```

Перечисленные выше функции не исчерпывают возможности DOS по выводу информации на экран. Используются также функция 6 (при dl не равном 0ffh) и функция 40h с дескриптором (в bx) равным 1 или 2. Однако, все равно возможности BIOS при выводе на экран значительно более широкие и гибкие.

9.4 Прерывания BIOS для работы с экраном

9.4.1. Рассмотрим пример гашения курсора.

```
ah = 1
ch = 20
cl = 0
int 10h.
```

Фрагмент программы с такими параметрами гасит на экране курсор.

9.4.2. Рассмотрим пример установки курсора в заданную позицию.

```
ah = 2
bh = номер видеостраницы (у нас 0)
dh = номер строки (0-24)
dl = номер столбца (0-79)
int 10h.
```

Устанавливает курсор в заданную позицию.

Пример. Установить курсор в центр экрана.

```
mov dx,0c28h          ; 12-я строка (0ch), 40-й столбец (28h).
mov bh,0              ; устанавливаем нулевую видеостраницу
mov ah,2              ; задаем вторую функцию
int 10h               ; вызываем десятое прерывание.
```

9.4.3. Прокрутка заданной прямоугольной области экрана на заданное число строк вверх. Такая процедура называется "скроллинг".

```
ah=6
ch=  строка           ; в cx задается левый верхний угол
cl=  столбец          ; области
dh=  строка           ; в dx задается правый нижний угол
dl=  столбец          ; области
al=  на сколько строк поднимать
bh=  атрибуты для заполнения освобождающихся строк
int 10h.
```

Атрибуты цвета экрана и цвета символа задаются в регистре bh, согласно следующему представлению задания цветов, показанному на рисунке 9.1.

Цвет фона				**Цвет символа**			
7	6	5	4	3	2	1	0
M	**R**	**G**	**B**	**I**	**R**	**G**	**B**

Рисунок 9.1 – Формат регистра для задания атрибутов

Разряды регистра имеют следующие значения:

– M – мерцание символа или интенсивность фона, в зависимости от выбранного режима;

– I – интенсивность символа;

– R – красный цвет;

– G – зеленый цвет;

– B – синий цвет.

Комбинация нулей и единиц дает различные цвета, например:

а) bh =7 – белый символ на черном фоне (87h-еще и мигает);

б) bh =70h – черный на белом (0F0h-еще и мигает);

в) bh =01001010b – на красном фоне ярко зеленые буквы.

Пример. Очистить экран.

```
mov cx,0          ; левый верхний угол экрана. Строка=0, столбец=0.
mov dx,184fh      ; правый нижний угол экрана. Строка=24 (18h), столбец=79 (4fh)
mov bh,7          ; белый по черному
mov ax,619h       ; функция 6. Поднять на 25 строк (19h), то есть на весь экран
int 10h.
```

9.4.4. Вывод заданного символа в текущую позицию курсора.

ah=9

bh = номер видеостраницы (у нас 0)

bl = атрибуты символа (см. предыдущую функцию)

al = ASCII-код символа

cx = число повторений

int 10h.

Выводит заданный символ в текущую позицию курсора. Курсор при этом не перемещается. В cx помещается число x (x>=1).

При выводе символ распространяется на x позиций вправо от курсора. То есть если x=1, то будет напечатан один символ, при x=2 – два символа (одинаковых) и.т.д. Коды 7,8,0ah и 0dh являются управляющими.

Пример. Забить верхнюю строку экрана символом "*". Вывод произвести черным по белому.

```
mov ah,2          ; устанавливаем курсор
mov bh,0          ; видеостраница 0
mov dx,0          ; левый верхний угол экрана (строка=столбец=0)
int 10h
mov ah,9          ; вывод символа
mov bh,0          ; видеостраница 0
mov bl,70h        ; черным по белому
```

```
mov al,'*'
mov cx,80          ; заполняем всю строку
int 10h.
```

9.4.5. Вывод символа в текущую позицию курсора

```
ah = 0eh
al = ASCII-код символа
int 10h.
```

Выводит символ в текущую позицию курсора, после чего курсор сдвигается на позицию вправо. Символ выводится с текущими атрибутами.

Примечание. Перечисленные выше функции далеко не исчерпывают возможностей прерывания **int 10h** по выводу информации на экран.

Существует еще немало других функций этого прерывания. Кроме того возможности некоторых рассмотренных выше функций приведены не полностью.

9.5 Варианты задания

1. Программа очищает экран, вырезает в центре экрана инверсное окно разумных размеров и выводит внутрь этого окна Ф.И.О. всех членов бригады и номер группы. Выход из программы по нажатию клавиши TAB.

2. Программа очищает экран и выводит в центр экрана рамку разумных размеров. Символы, которыми рисуется рамка, выбирает студент в соответствии со своим вкусом. Программа выводит внутрь рамки Ф.И.О. всех членов бригады и номер группы. Надписи внутри рамки должны мигать. Выход из программы по нажатию клавиши F10.

3. Программа очищает экран и выводит в центр экрана Ф.И.О. всечленов бригады и номер группы. Нажатие клавиши I инвертирует все цвета на экране. Нажатие клавиши M включает/выключает мигание надписей. Выход из программы по нажатию клавиши F6.

4. Программа очищает экран и вырезает в его центре инверсное окно, разумных размеров. При нажатии клавиши "1" в окно выводится Ф.И.О. первого члена бригады, при нажатии клавиши "2"- второго члена бригады и так далее.

При нажатии клавиши G в окно выводится номер группы. В любой момент времени в окне располагается не более одной надписи, то есть при нажатии любой из вышеперечисленных клавиш вся старая информация из окна удаляется и туда вводится новая информация. Выход из программы по нажатию клавиши F5.

5. Программа очищает экран и вырезает в его центре инверсное окно разумных размеров. Внизу экрана печатается строка с просьбой ввести Ф.И.О. первого члена бригады. Ф.И.О. вводится с клавиатуры, а затем по нажатию клавиши V выводится в окно. Далее процесс повторяется для остальных членов бригады и номера группы. В конце концов в окне должна располагаться полная информация о бригаде. Выход из программы по нажатию клавиши F7.

6. Программа очищает экран и выводит в ее центр нечто вроде рамки. Контуры этой рамки составляет написанная несколько раз фамилия одного из членов бригады. Количество повторов определяется, естественно, размерами рамки. Внутрь рамки помещаются Ф.И.О. остальных членов бригады. Либо рамка, либо ее внутреннее содержимое должны мигать. Переключение процесса мигания между рамкой и ее внутренним содержимым осуществляется по нажатию клавиши P. Выход из программы по нажатию клавиши F3.

7. Программа очищает экран и вырезает на нем два инверсных окна. В 1-е из них выводится Ф.И.О. одного, а во 2-е - второго члена бригады. При нажатии клавиши C информация в окнах меняется местами. Выход из программы по нажатию клавиши TAB.

8. Программа очищает экран. В центре экрана печатается звездочка. С помощью клавиш-стрелок можно перемещать эту звездочку по всему экрану, не выходя, однако, за его пределы. В любой момент на экране находится только одна звездочка. Выход из программы по нажатию клавиши TAB.

9. Программа очищает экран. При нажатии любой символьной клавиши на экране появляется соответствующий символ, который в дальнейшем можно распространять по экрану с помощью клавиш-стрелок. В любой момент можно сменить символ нажав другую символьную клавишу. Выход из программы по нажатию клавиши ESC.

10 Задание №6. Написание программ на языке ассемблера для управления вводом-выводом

Рассмотрим управление вводом-выводом непосредственно на уровне порта (генерация звука). В состав любой ПЭВМ фирмы IBM входит микросхема таймера i8253(54). Микросхема имеет три канала, соответственно каналы 0,1 и 2. Канал 0 отводится для службы системного времени, канал 1 - для регенерации памяти, а канал 2 - для управлением работой динамика.

Порты таймера имеют следующие системные адреса: канал 0 - 40h; канал 1 - 41h; канал 2- 42h; регистр управляющего слова (РУС)- 43h.

Через РУС производится настройка каналов, через остальные порты-загрузка/считывание информации в/из соответствующих каналов.

Для работы со звуком нам нужен канал 2 и РУС. Каналы 0 и 1 перезагружать и перенастраивать категорически запрещено так как система выйдет из строя (во всяком случае до перезапуска).

Работа канала 2 заключается в том, что он делит опорную частоту (fоп=1.19... МГц. для XT) на коэффициент пересчета Кпр, который заранее загружается в канал. Получаемая fвых=fоп/Кпр подается на динамик. Нао учитывать что работой канала 2 и подачей fвых на динамик управляют два младших бита порта 61h. Если бит 0 порта 61h равен единице, то работа канала 2 (счет) разрешается. Если бит 1 порта 61h равен единице, то разрешается подача fвых на динамик. Таким образом звук будет воспроизводиться только если оба этих бита установлены в единицу.

Коэффициент пересчета для любой ноты можно определить исходя из выражения: Кпр=fоп/fноты.

При этом учитываются следующие соотношения:

- частота ноты "до" 1-й октавы (fдо)= 32.625 Гц;
- частота ноты "до" 2-й октавы = 2*fдо;
- частота ноты "до" 3-й октавы = 4*fдо и т.д;
- частота "до-диез" = a*fдо;
- частота "ре" = a*fдо-диез и т.д.,

где a=1.06 (приблизительно).

Если вам необходимо сыграть на компьютере мелодию, то проще посчитать коэффициенты всех нот заранее или взять их готовыми, так как они давно известны. Таким образом для мелодии желательно подготовить следующие данные:

а) коэффициент пересчета и время звучания для каждой ноты;

б) длительности всех пауз между нотами.

После того как все эти числа известны надо последовательно выполнять следующие действия:

а) запретить звучание, для чего установить в ноль оба младших бита порта 61h (не меняя при этом остальные биты этого порта);

б) настроить канал 2 выполнив две команды

```
mov al,0b6h
out 43h,al;
```

в) за две передачи в канал 2 загрузить Кпр для текущей ноты

```
mov al,младший байт Кпр
out 42h,al
mov al,старший байт Кпр
out 42h,al;
```

г) разрешить звучание, установив в единицу оба младших бита порта 61h;

д) ввести задержку, равную длительности ноты;

е) запретить звучание

ж) выдержать паузу требуемой длительности и перейти к пункту а.

10.1 Задание к работе

В соответствии с музыкальными способностями и вкусом реализовать на ПЭВМ любую известную мелодию начиная от "гаммы до мажор" до "Полонеза" Огинского. Единственным требованием является несовпадение вашей мелодии с мелодиями других студентов.

11 Задание №7. Разработка и использование макрокоманд Ассемблера

Целью выполнения лабораторной работы является изучение языка и возможностей Макроассемблера, способов написания и использования макрокоманд, приемов их отладки и тестирования. В работе студенты разрабатывают собственные макрокоманды, проверяют их работу, получают навыки создания систем макрокоманд и их отладки. По результатам работы оформляется отчет.

11.1 Требования к выполнению лабораторной работы

При разработке программы с макрокомандами и их отладки, студент должен выполнить следующие требования. Нужно спроектировать, запрограммировать и отладить следующие макрокоманды.

1. Макрокоманду **вывода на печать одного символа** (на консоль) с двумя параметрами: выводимый символ и признак перевода строки (для него назначить текстовую константу "**PER**"). Нужно продемонстрировать использование макрокоманды во всех режимах, включая и пример ошибочного задания параметров. При наличии ошибки выдается сообщение оператором макроассемблера **%OUT** с включенным режимом контроля ошибок - **.ERR**. Контроль пустого параметра выполнять условным оператором **IFNB**. Макрокоманду назвать **PRINT.** Данное задание является общим для всех студентов и не имеет вариантов. Контроль правильности генерации макрокоманд проверяется по листингу с макрорасширениями, выдаваемому макроассемблером. Для управления листингом, с этой целью, нужно использовать псевдооператоры управления листингом макроассемблера**: .LALL , .SALL, .XALL**.

2. Макрокоманду **описания и заполнения массива** (назвать макрокоманду **MAS**). В зависимости от варианта массив может быть байтовым (**DB**) или двухбайтовым (**DW**). Макрокоманда должна иметь четыре параметра: размер массива, название массива, базовое значение для заполнения и параметр заполнения массива. В зависимости от варианта, заполнение массива выполняется: натураль-

ными числами, возрастающими по значению, натуральными числами, убывающими по значению, последовательностью арифметической прогрессии, последовательностью геометрической прогрессии. Для заполнения массива использовать вложенные макрокоманды и целые макро переменные (целые переменные декларируются с помощью "=", а константы с помощью **EQU**). Числа для заполнения массива должны быть сгенерированы на этапе макропроцессора, быть положительными и возможные значения размерности должны быть не менее 10. Продемонстрировать использования описания массивов для разных случаев, в том числе и при ошибочном задании параметров. При разработке макроопределения использовать операторы макроассемблера: "=", "%" и "&". При написании макрокоманды MAS необходимо использовать оператор **REPT**. Название массива должно устанавливаться у его первого элемента.

3. Результат генерации массива должен быть примерно таким (рисунок 11.1) (здесь показано байтовое заполнение массива, и используется арифметическая прогрессия для заполнения целыми числами).

```
Вызов макрокоманды создания описания массива
MAS 15, NAME, 1 , 1.
Результаты макрогенерации с заданными параметрами
1 NAME DB  1
1      DB  2
1       ...
1      DB  15.
```

Рисунок 11.1 – Результат генерации массива

Массив имеет название “NAME”. Значение заполнения переменных зависят от варианта. Имя массива передается в качестве параметра. В примере показано заполнение натуральными числами с базовым значением 1.

4. Макрокоманду **подсчета** агрегированных характеристик массива (суммы или произведения). В макрокоманде (назвать макрокоманду **AGREGATE**) нужно предусмотреть следующие три параметра: название массива, размерность массива для подсчета, переменная, в которую возвращается результат вычисления. В зависимости от варианта в макрокоманде выполняется подсчет: **суммы** заданно-

го числа переменных массива, их **произведения**, числа **нечетных** элементов массива и числа элементов, в которых **бит четности** для числа равен нулю (не путать с четными и нечетными переменными). При организации цикла в макрокоманде использовать локальные метки и переменные (оператор **LOCAL**). Предположить для массива из слов (DW), что результат расчета сможет поместиться в переменную типа слово (DW). В случае переполнения результата возвращать "-1" (для суммы и произведения). При выполнении макрокоманды сохранять все используемые регистры и восстанавливать их при ее завершении (PUSH, POP). Результат расчета распечатать в десятичном виде, для чего нужно воспользоваться процедурами перевода и печати, разработанными в предыдущих лабораторных работах по Ассемблеру.

5. Макрокоманду распечатки массива (назвать макрокоманду **PRINT_MAS**). Распечатка выполняется в десятичном формате, для этого используются процедуры из предыдущих лабораторных работ по ассемблеру. В зависимости от варианта распечатка выполняется по строкам, с указанием количества чисел в строке и по колонкам, с указанием числа столбцов. Макрокоманда должна содержать следующие параметры: название массива, число элементов, которые должны быть распечатаны, параметры распечатки (число строк или столбцов). При распечатке между отдельными числами должны быть пробелы, лидирующие нули должны заменяться на пробелы. При организации циклов должны определяться локальные метки и переменные. Вид по строкам и столбцам показан на рисунке 11.2.

```
Распечатка по строкам (число чисел в сроке 5, размер 8)
1   2   3   4   5
6   7   8.
Распечатка по столбцам  (число столбцов 6, размер 11)
1   3   5   7   9   11
2   4   6   8  10
```

Рисунок 11.2 – Распечатка массива по строкам и столбцам

6. Продемонстрировать использование в программе операторов макроассемблера **IRP и IRPC**. Для этого написать текст программы с использованием этих

псевдооператоров макроассемблера и получить их макрорасширение (рисунок 11.3).

```
mycode segment 'code'

 assume cs:mycode, ds:mycode, ss: stseg
; Макрокоманда печати символа с переводом строки
; 1-й параметр= символ, второй параметр признак перевода "PER"
PRINT MACRO  ch, CR
  mov dl, ch
  mov ah, 2
  int 21h
     IFIDN <CR>,<PER>
  mov dl, 0Ah
  mov ah, 2
  int 21h
  mov dl, 0Dh
  mov ah, 2
  int 21h
     ELSE
  IFNB <CR>
  .ERR
  %OUT ERRROR Ошибка параметра перевода строки
  ENDIF
     ENDIF
     ENDM
main proc
;Занесение регистра DS
     PUSH CS
     POP DS
     PRINT '5', PER
; Ошибка - print 'A', AAA
     print 'A'
     PRINT ' ', PER
; выход с ожиданием
;  Запрос символа с клавиатуры
    mov ah, 08h
    int 21h
    mov al, 0
;  Выход в ДОС
    mov ah, 4ch
    int 21h
main endp
;-----------------
mycode ends
;-------
stseg segment stack 'stack'
 dw 256 dup(0)
stseg ends
 end main
```

Рисунок 11.3 – Пример программы с макрокомандой

Если макрокоманда вызвана с ошибочным параметром, в тексте примера этот вызов закомментирован, то макроассемблер сформирует сообщение об ошибке на этапе работы макропроцессора. Здесь внесена ошибка параметра “AAA” вместо “PER”(рисунок 11.4).

```
...
; Ошибка -
                              print 'A', AAA
0014  B2 41             1        mov dl, 'A'
0016  B4 02             1        mov ah, 2
0018  CD 21              1               int 21h
                        1        .ERR
i:\2008_2~1\kaf\sp_2009\asm_met\mac1s.asm(33): error A2089: forced error
...
```

Рисунок 11.4 – Пример сообщения об ошибке

Программа откомпилирована и отлажена в среде QC25.

11.2 Задание к лабораторной работе

Задание на лабораторную работу заключается в разработке программы, в которой студенты дают описание собственных макрокоманд по темам определенным в вариантах и демонстрируют их правильное использование на конкретных примерах. Должны быть разработаны макрокоманды с параметрами, включающие условную компиляцию, локальные переменные и вложенные макрокоманды.

В ходе работы студенты разрабатывают: макрокоманду вывода текстовых данных, макрокоманду описания массива с конкретным заполнением (по варианту), макрокоманду расчета агрегированных характеристик массива (сумма, произведение и т.д.) и макрокоманду вывода на печать результатов работы программы.

Варианты заданий представлены в таблице 11.1

Таблица 11.1– Варианты заданий

№	Переменные массива	Печать массива	Подсчет	Заполнение
1	DB	По столбцам	Числа нечетных элементов массива	Арифметическая прогрессия целых чисел (задается параметр команды)
2	DW	По столбцам	Числа нечетных элементов массива	Геометрическая прогрессия целых чисел (задается параметр команды)
3	DB	По строкам	Числа нечетных элементов массива	Натуральные числа в порядке возрастания (1,2,3, …)
4	DW	По строкам	Числа нечетных элементов массива	Натуральные числа в порядке убывания (15,14,13, …)
5	DW	По строкам	Числа нечетных элементов массива	Арифметическая прогрессия целых чисел (задается параметр команды)
6	DW	По строкам	Суммы элементов массива	Арифметическая прогрессия целых чисел (задается параметр команды)
7	DB	По столбцам	Суммы элементов массива	Геометрическая прогрессия целых чисел (задается параметр команды)
8	DW	По столбцам	Суммы элементов массива	Натуральные числа в порядке возрастания (1,2,3, …)
9	DB	По строкам	Суммы элементов массива	Натуральные числа в порядке убывания (15,14,13, …)
10	DW	По строкам	Суммы элементов массива	Геометрическая прогрессия целых чисел (задается параметр команды)
11	DW	По строкам	Числа элементов массива, для которых вычисленные бит четности равен 0	Арифметическая прогрессия целых чисел (задается параметр команды)
12	DW	По столбцам	Числа элементов массива, для которых вычисленные бит четности равен 0	Геометрическая прогрессия целых чисел (задается параметр команды)

Продолжение таблицы 11.1

№	Переменные массива	Печать массива	Подсчет	Заполнение
13	DB	По строкам	Числа элементов массива, для которых вычисленные бит четности равен 0	Натуральные числа в порядке возрастания (1,2,3, …)
14	DB	По столбцам	Числа элементов массива, для которых вычисленные бит четности равен 0	Натуральные числа в порядке убывания (15,14,13, …)
15	DW	По столбцам	Числа элементов массива, для которых вычисленные бит четности равен 0	Натуральные числа в порядке возрастания (1,2,3, …)
16	DW	По строкам	Произведения элементов массива	Арифметическая прогрессия целых чисел (задается параметр команды)
17	DW	По столбцам	Произведения элементов массива	Геометрическая прогрессия целых чисел (задается параметр команды)
18	DB	По столбцам	Произведения элементов массива	Натуральные числа в порядке возрастания (1,2,3, …)
19	DB	По строкам	Произведения элементов массива	Натуральные числа в порядке убывания (15,14,13, …)
20	DW	По столбцам	Произведения элементов массива	Натуральные числа в порядке убывания (15,14,13, …)

11.3 Требования к оформлению отчета по ЛР

В отчет по лабораторной работе должны входить ниже перечисленные пункты.

1. Постановка задачи (общие требования и требования варианта)
2. Блок-схемы макрокоманд программы.
3. Листинг программы командного файла с расширениями вызовов макрокоманд.
4. Результаты запуска программы при проверке работы макрокоманд.

5. Перечень основных ошибок, которые возникали и были исправлены при отладке макрокоманд. Отсутствие перечня ошибок, или копирование его у других студентов, для меня дает дополнительную информацию о самостоятельности работы над заданием лабораторной работы конкретного студента.

11.4 Контрольные вопросы к лабораторной работе

1. На каком этапе компиляции работает макропроцессор?
2. В каких случаях выгоднее использовать макрокоманды по сравнению с процедурами?
3. В каких случаях выгоднее использовать процедуры по сравнению с макрокомандами?
4. Каково основное преимущество макрокоманд?
5. Для чего нужен оператор LOCAL?
6. Какую операцию выполняет оператор макроассемблера "&"?
7. Какую операцию выполняет оператор макроассемблера "%"?
8. Что нужно сделать для отмены или включения печати макрорасширений в распечатке ассемблера?
9. Можно ли из одной макрокоманды вызывать другую макрокоманду с параметрами?
10. Как определить локальную переменную в макрокоманде (не метку)?
11. Для чего используется оператор REPT?
12. Как завершить выполнение макрокоманды?
13. Как можно изменить значение текстовой переменной этапа компиляции?
14. Как можно изменить значение целочисленной переменной этапа компиляции?
15. Как и куда можно вывести сообщение об ошибках этапа компиляции, определяемых пользователем?
16. Как можно подключить библиотеку внешних макрокоманд?
17. Как можно связать выполнение различных макрокоманд?
18. Что такое условная компиляция?
19. Перечислите основные операторы макроассемблера.
20. Можно ли считать, что макрокоманда – это процедура, которая выполняется на этапе компиляции программы?

12 Построение резидентных программ

Резидентные программы (TSR – Terminate and Stay Resident) – это программы, которые остаются в оперативной памяти (ОП) после их специального завершения. Эти программы могут работать параллельно (псевдопараллельно) с другими программами операционной системы. Обращение к этим программам выполняется через механизм прерываний (программные или аппаратные прерывания), как показано на рисунке 12.1.

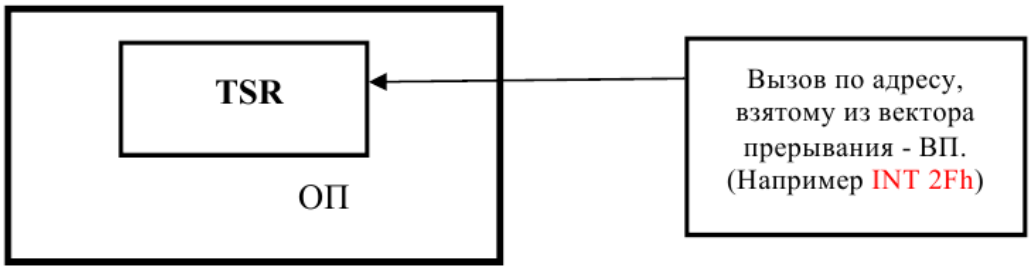


Рисунок 12.1 − Вызов прерываний

Для правильной работы резидентной программы (TSR) необходимо выполнить следующие действия и процедуры:

− проверить повторную загрузку данного нового резидента;

− загрузить резидентную программу в оперативную память;

− выдать сообщение о корректной загрузке резидента в память;

− записать в вектор прерываний адрес резидента;

− прочитать и правильно использовать параметры запуска резидента;

− обеспечить вызов старого обработчика прерываний по данному адресу, если такое необходимо;

− правильно отработать в процедуре резидента заданные функции;

− обеспечить связь с резидентной программой с помощью клавиатуры, программно или другим способом;

− обеспечить, при необходимости корректную выгрузку резидентной программы и освобождение занимаемой резидентом памяти;

− корректно восстановить работу старых обработчиков данного прерывания,

восстановить адреса старых обработчиков в векторе прерываний;

− выдать сообщение о завершении работы резидентной программы.

В данном разделе мы рассмотрим основные действия и процедуры, необходимые для построения резидентной программы и ее корректной работы.

12.1 Таблица векторов прерываний

Таблица векторов прерываний (ТВП) – этот область оперативной памяти, содержащей вектора прерываний (адреса обработчиков прерываний). Каждый вектор занимает 4 байта в формате сегмент:смещение (адрес сегмента – 2 байта и адрес смещения – 2 байта). Таблица векторов прерываний находится в начале ОП (для реального режима) по адресам 0000:0000 – 0000:03FF, то есть имеет размер 1024 байта. В ТВП максимально можно записать 256 векторов, т.е. возможно обрабатывать 256 различных прерываний с номерами от 0 до 255. Расположение ТВП в ОП показано на рисунке 12.2.

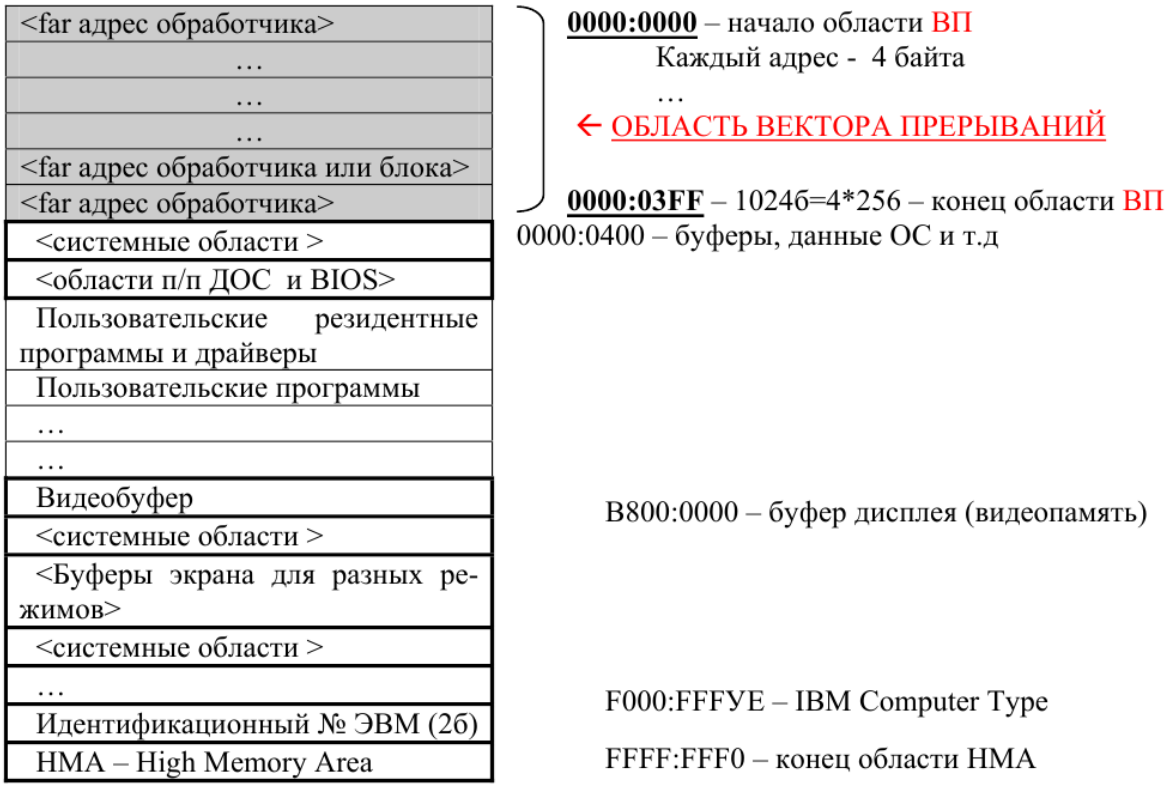


Рисунок 12.2 – Расположение таблицы векторов прерываний

Вычисление адреса конкретного обработчика в таблице векторов прерываний выполняется умножением его номера (прерывания) на четыре: например, для прерывания 09H мы получим – 0000:0024H (или десятичное смещение 36). Доступ к адресам обработчиков прерываний может быть прямым (вычисляем адрес) или с помощью системных функций (25h и 35h прерывания DOSc номером 021h). Эти возможности будут рассмотрены ниже.

12.3 Структура резидентной программы

Обычно резидентная программа (резидент) первоначально состоит из двух частей, показанных на рисунке 12.3.

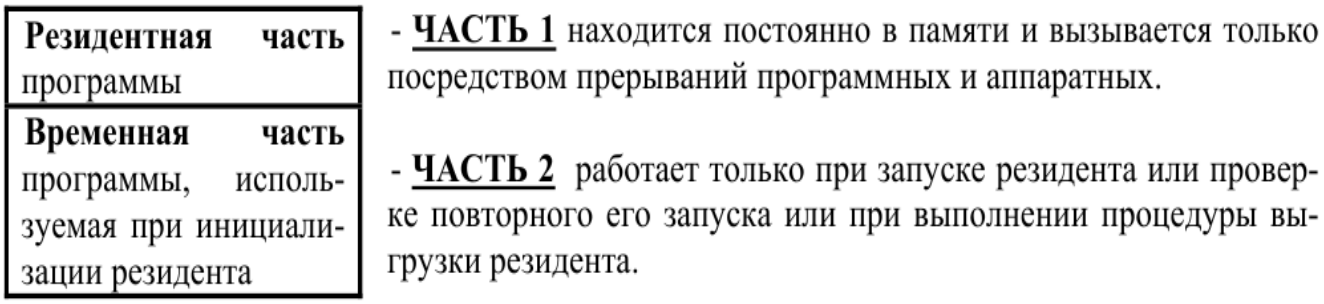


Рисунок 12.3 – Структура резидентной программы

При завершении установки резидента оперативная память, отведенная под временную часть, обычно освобождается, поэтому команды и данные из этой части нельзя использовать в процедурах резидентной части. В состоянии проверки повторности загрузки резидентной программы в ОП может находиться одновременно две части, так как программа запускается повторно.

12.4 Обработка прерываний в процессоре

Прерывания – это специальный механизм, позволяющий операционной системе выполнять свои функции (управление ресурсами). Они обеспечивают связь аппаратной части компьютера и обрабатывающих программ. Кроме того, меха-

низм прерываний обеспечивает выполнение множества стандартных функций для управления программами и устройствами.

Прерывания – это автоматически инициируемый программный процесс, временно переключающий микропроцессор на выполнение другой программы. После обработки прерывания прерванный процесс автоматически будет продолжен.

В момент возникновения прерывания в стеке запоминается точка возврата (адрес следующей команды, которая бы выполнялась, если бы прерывания не было) и содержимое регистра флагов. После завершения обработки прерывания из стека восстанавливается регистр флагов (Flags) и прерванная программа продолжает свое выполнение.

Прерывания подразделяются на программные и аппаратные. Данная классификация основана на причине инициирования прерывания. В случае программных прерываний они возникают после выполнения специальной команды (INT – interrupt – прерывание), которая выполняется в самой прерываемой программе. Программа фактически сама себя прерывает. При аппаратных прерываниях срабатывают специальные схемы контроля компьютера, и сигнал прерывания поступает через специальное устройство (КП – контроллер прерываний) в центральный процессор (CPU), после чего работающая программа прерывается.

Аппаратные прерывания могут возникать от любых устройств, которые могут связываться с процессором (клавиатура, мышь, диски и т.п.). Устройства с помощью прерываний сообщают процессору о своем состоянии, выполнении действий и исправности или неисправности.

Аппаратные прерывания инициируются устройствами, подключенными к компьютеру, для этого они подключены к специальной микросхеме – контроллеру прерываний КП. Эта микросхема в свою очередь подключена к центральному процессору и позволяет по специальным линиям передать: сигнал прерываний и специальный номер прерывания.

Аппаратные прерывания подразделяются также на маскируемые и немаскируемые прерывания. Если прерывание может быть временно запрещено, то оно относится к группе маскируемых прерываний. Программные прерывания не мо-

гут быть замаскированы, и они подразделяются на прерывания DOS и прерывания BIOS зависимости от места расположения процедур обработки прерываний.

При инициировании программных прерываний в программе выполняется специальная машинная команда:

INT <номер прерывания>,

где <номер прерывания> – число в пределах от 0 до 255 (один байт). Данный номер определяет адрес процедуры обработки прерывания в таблице векторов прерываний.

В РГЗ по дисциплине «Программирование микропроцессорных систем» необходимо разобраться с механизмом инициации и обработке аппаратных и программных прерываний и написать собственные процедуры обработки прерываний в виде резидентных модулей.

На рисунке 12.4 представлена обобщенная и укрупненная схема обработки программных и аппаратных прерываний. Данная схема может быть доработаны для каждого конкретного случая.

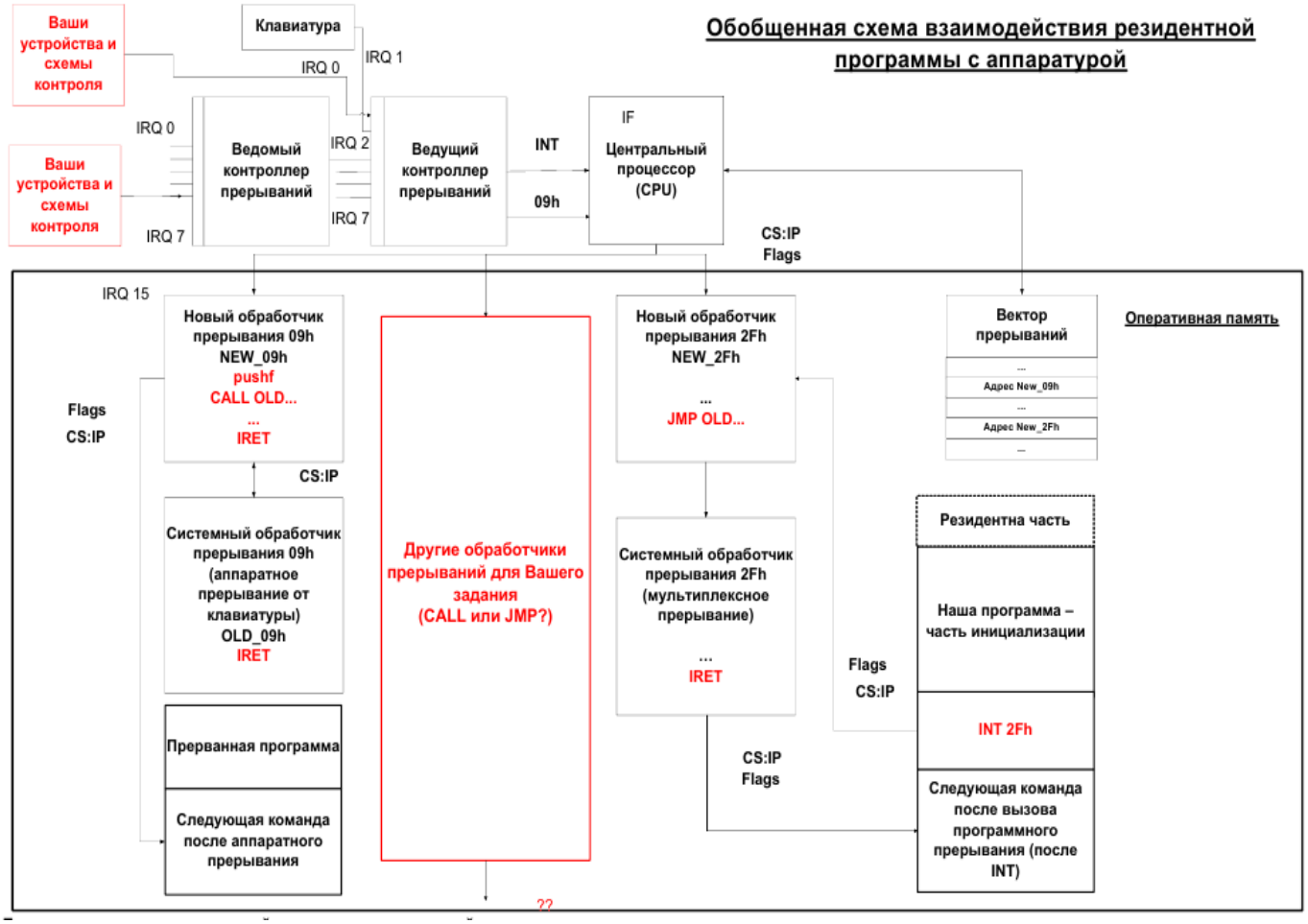


Рисунок 12.4 − Схема обработки программных и аппаратных прерываний

Поясним кратко последовательность действий при обработке аппаратных и программных прерываний.

Аппаратные прерывания (в нашем случае обработка прерывания от клавиатуры – номер – 09H):

− нажата любая клавиша на клавиатуре;

− инициируется прерывание (IRQ1) и сигнал поступает в контроллер прерывания.

12.4.1 Для обработки 15-ти прерываний, на нашей схеме каскадом соединены 2 контроллера прерываний (КП), каждый из которых способен обработать только по 8 сигналов прерываний, причем один из входов ведущего контроллера задействован для подключения выхода INT ведомого контроллера

− из контроллера прерываний в центральный процессор (CPU) поступает сигнал INT и номер прерывания 09H;

− по номеру прерывания процессор выбирает адрес обработчика прерывания

из вектора прерывания, в нашем случае этот сдвиг в таблице векторов прерываний равен 36 (9*4);

− процессор запоминает в стеке регистр флагов (Flags) и дальний адрес возврата в прерванную программу (CS:IP);

− на основе адреса обработчика устанавливаются новые CS:IP, что фактически

приводит к вызову новой процедуры обработчика прерываний;

− новый обработчик прерываний выполняет необходимые (по варианту задания) действия;

− в новом обработчике необходимо запомнить в стеке регистры общего назначения, которые используются в его программе;

− при необходимости может быть вызван старый обработчик прерываний по методу CALL или методу JMP. При использовании вызова старого обработчика с возвратом нужно в стек поместить регистр флагов (команда PUSHF) для того, чтобы корректно работала команда IRET в этом обработчике. Вызов с возвратом может быть выполнен в начале нового обработчика, при его окончании или в се-

редине. При вызове с командой JMP выполняется безусловный переход и в стек ничего не заносится. Перед вызовом по методу JMP нудно восстановить все запомненные регистры общего назначения;

− восстанавливаются регистры общего назначения в новом обработчике;

− выполняется возврат в прерванную программу командой IRET, которая из стека выбирает регистр флагов и дальний адрес возврата CS:IP;

− продолжает работать прерванная программа.

Программные прерывания (в нашем случае обработка прерывания запускается командой INT 2FH):

− выполняется команда INT 2FH в части инициализации резидентной программы;

− процессор, выполняя эту команду, определяет адрес нового обработчика прерывания (2F16 = 4710 , смещение 47*4 = 188). Получаем дальний адрес нового обработчика;

− запоминается в стеке регистр флагов и текущие значения CS:IP в качестве точки возврата в прерванную программу;

− на основе адреса обработчика устанавливаются новые CS:IP, что фактически приводит к вызову новой процедуры обработчика прерываний;

− новый обработчик прерываний выполняет необходимые (по варианту задания) действия;

− в новом обработчике необходимо запомнить в стеке регистры общего назначения, которые используются в его программе;

− восстанавливаются регистры общего назначения в новом обработчике;

− при необходимости может быть вызван старый обработчик прерываний по методу CALL или методу JMP. При использовании вызова старого обработчика с возвратом нужно в стек поместить регистр флагов (команда PUSHF) для того, чтобы корректно работала команда IRET в этом обработчике. Вызов с возвратом может быть выполнен в начале нового обработчика, при его окончании или в середине. При вызове с командой JMP выполняется безусловный переход и в стек ничего не заносится. Перед вызовом по методу JMP нудно восстановить все запомненные регистры общего назначения;

− выполняется возврат в прерванную программу из старого обработчика прерывания командой IRET, которая из стека выбирает регистр флагов и дальний адрес возврата CS:IP;

− продолжает работать прерванная программа в части инициализации.

В РГЗ студенты должны приспособить данную схему для своего варианта и описать последовательность обработки прерываний.

12.5 Установка резидента

В конце части инициализации резидентной программы необходимо указать операционной системе, что часть программы останется резидентной (Часть 1), то есть выполнить установку резидента.

Для установки резидента используется специальное прерывание DOS (Terminate and Stay Resident – TSR) **031H – 21H,** или BIOS прерывание **027H.**

Эти варианты отличаются способом задания размера памяти резидента («обрезания») и отдельными свойствами (более подробную информацию можно найти в справочниках DOS). Ниже приведены способы установки резидента.

12.5.1 Первый способ (ОП >64kb и старшие версии ДОС).

MOV DX , <размер резидента в параграфах>

MOV AX, 3100H

INT 21H.

12.5.2 Второй способ.

MOV DX , <адрес конца резидента от PSP>

INT 27H .

Для правильной установки необходимо рассчитать в программе его размер.

12.6 Расчет размера резидента

Для случая с прерыванием 031H – 21H нужно определить в параграфах размер оставляемой памяти. Он определяется для программы

ORG 100h

BEGIN: …

…

INIT: , так MOV DX, (INIT – BEGIN + 10fh)/16 .

Размер вычисляется как разница начала части инициализации (INIT), которая будет удалена (Часть 2) и адреса точки входа в программу (BEGIN). Кроме того, добавляется размер области PSP и корректировочная константа. Полученное значение нужно разделить на 16 для получения размера в параграфах (16 байт). Причем: 100h – размер PSP, а 0Fh – коррекция при делении нацело (16) для округления целого числа параграфов.

Для прерывания 027H указывается адрес конца резидента, как адрес начала части инициализации (INIT)

MOV DX, OFFSET INIT

INT 027h.

12.7 Запуск части инициализации

Резидентная программа обычно оформляется в формате *.COM. Поэтому точка входа должна быть расположена непосредственно после области выделяемой под PSP процесса (ORG 100H). Поэтому точка входа должна быть помечена в первой строке после предполагаемой области PSP (метка BEGIN). В директиве конца программы END эта метка также должна быть задана. После запуска программы из командной строки выполняется безусловный переход на метку инициализации (JMP INIT). Начало резидента может быть оформлено и в виде процедуры (BEGIN PROC … BEGIN ENDP), но это не имеет особого значения, так как в эту процедуру возврат не осуществляется.

Ниже приводится фрагмент запуска части инициализации резидентной программы.

```
; Начало резидента
CODEPR SEGMENT PARA
ASSUME CS:CODEPR , DS:CODEPR
; Начало части резидента (Часть 1)
ORG 100H; установка счетчика команд (байт) на 100H для последующей
```

```
; вставки PSP
BEGIN:
JPM INIT
; это метка начала части инициализации
; Определение данных резидента
…
; Начало части инициализации (Часть 2)
INIT:
…
END BEGIN;  Определение точки входа при запуске в командной строке.
```

Область данных резидента обычно располагается после этой метки, хотя эти данные можно расположить в любом месте резидентной части (часть 1 – располагается между метками **BEGIN** и **INIT**).

12.8 Определение и запоминание старого обработчика

Для каждого прерывания, которое будет обрабатываться в резиденте, нужно определить и запомнить адрес старого обработчика прерывания. Это делается для того, чтобы после завершения работы резидентной программы модно было восстановить старый обработчик и, при необходимости, вызвать его из собственного обработчика для выполнения стандартных действий (например, при обработке прерывания от клавиатуры). Эти действия могут быть выполнены либо с помощью специально прерывания (35H – 21H) либо с помощью непосредственного обращения в область вектора прерывания , как приведено ниже.

```
; область данных части резидента
OLD_09 dd ?
; адрес старого обработчика 09h
…
; часть инициализации
…
MOV AH, 35h
```

```
MOV AL , 09h
; номер обработчика
INT 21h
MOV WORD PTR CS:OLD_09, BX
MOV WORD PTR CS:OLD_09+2, ES.
```

Адрес обработчика считывается в поле длинной в 4 байта (длинный адрес). Нужно помнить, что двойные слова сохраняются в памяти в обратном порядке (сначала младшее слово – смещение, а затем старшее слово – сегмент). После обращения к прерыванию 035H, в регистрах мы получим дальний адрес из вектора прерывания (ES:BX). При этом на AL указывается номер прерывания. Для приведения адресного выражения к слову используется спецификация WORD PTR.

Такие действия необходимо выполнить в программе для каждого обработчика прерывания, задаваемого в программе.

12.9 Задание нового обработчика прерывания

Для каждого прерывания, которое обслуживает резидентная программа нужно определить новый обработчик прерывания. Это делается с помощью функции **025h– 21h** или с помощью непосредственного запись в область вектора прерывания (см. ниже). В приведенном ниже фрагменте предполагается, что в регистре CS записан адрес сегмента кода программы. Новый обработчик располагается в резидентной части программы. На регистре AL должен быть задан номер нового обработчика прерывания, а на регистре DX смещение обработчика – NEW_09 (для CS). Ниже приведены фрагменты для части резидентной и части инициализации для установки нового прерывания.

```
; часть резидента
NEW_09 PROC
…
IRET
NEW_09 ENDP
 …
```

```
; часть инициализации
MOV AH, 25h
MOV AL , 09h
; номер прерывания обработчика
LEA DX, NEW_09
INT 21h .
```

12.10 Вызов старого обработчика прерывания

Вызов старого обработчика прерывания может быть выполнен как с помощью безусловного перехода **(JMP)**, так и как перехода с возвратом **(CALL)**. И в первом и во втором случае адресом вызова должен быть запомненный в резидентной части адрес старого обработчика (например: OLD_09).

При безусловном переходе вызов выполняется так

```
; тело нового обработчика прерывания
…
; вызов старого обработчика прерывания без возврата
JMP DWORD PTR CS:OLD_09.
```

При переходе с возвратом

```
; тело нового обработчика прерывания
…
; вызов старого обработчика прерывания с возвратом
PUSHF
CALL DWORD PTR CS:OLD_09
; продолжение тела нового обработчика прерывания
…
IRET.
```

Различие в этих вызовах заключается, в том числе, в наличии команды PUSHF, запоминающей в стеке регистр флагов. Это необходимо для того, чтобы при выполнении команды IRET стек корректно освобождался. Кроме этого, при вызове без возврата нет необходимости задавать команду IRET, так как заверше-

ний работы резидента и возврат к прерванной программе выполняется в старом обработчике.

12.11 Пример простейшего резидента

На основе материала изложенного выше можно построить простейшую резидентную программу. В этой программе не будет некоторых важных возможностей (проверки повторности, выгрузки резидента, обработки параметров и т.д.), но для простоты понимания задачи пока их опустим.

12.11.1 Написать резидентную программу, которая при нажатии клавиши ESC выводит на экран цепочку одинаковых символов (в программе ‘B’). При загрузке резидента должно выдаваться сообщение “Start TSR”.

Текст программы такого резидента приведен ниже.

```
;     Символы по нажатию ESC
        CODEPR SEGMENT PARA
            ASSUME CS:CODEPR , DS:CODEPR
           ORG 100H
; Область PSP
; Резидентная часть
        BEGIN:  JMP  INIT; Переход к части инициализации
; Данные резидента
        SAVEINT9 DD ?         ; Сохранение старого обработчика
;  Новый обработчик прерывания 09H
        NEWINT9: PUSH  AX; Сохранение используемых регистров
          PUSH  CX
          PUSH  BX
;   Запрос скан-кода из клавиатуры
             IN   AL, 60H    ; Взять из порта 60 H на регистр AL
             CMP  AL, 1  ; Проверка скан-кода ESC он равен 1
             JNE  EXIT     ;  Переход если не ESC
;   Вывод на экран 10–ти символов ’B’ с помощью BIOS функции 0AH – 010H
```

```
        PUSH  AX
        PUSH  BX
        PUSH  CX
        MOV AH , 0AH
          MOV AL, 'B'
          MOV BH , 0
          MOV CX, 10      ; Число символов
          INT 010H
          POP   CX
          POP   BX
          POP   AX
; Восстановление регистров и вызов старого обработчика без возврата
      EXIT:  POP   BX
          POP   CX
          POP   AX
          JMP CS:SAVEINT9   ; Вызов старого обработчика
; Часть инициализации
      INIT:  CLI       ; Запрет прерываний
; Установка DS
          PUSH CS
          POP DS
; Получение адреса старого обработчика
          MOV AH, 35H
          MOV AL, 09H  ; Номер прерывания
          INT 21H
; Сохранение адреса старого обработчика
          MOV   WORD PTR SAVEINT9 , BX
          MOV   WORD PTR SAVEINT9+2 , ES
; Установка нового обработчика в вектор прерывания
          MOV   AH,25H
          MOV   AL, 09H  ; Номер прерывания
```

```
        MOV   DX, OFFSET  NEWINT9
        INT   21H
; Вывод сообщения о загрузке резидента
        MOV AH , 09H
        MOV DX, OFFSET MSG
        INT  21H
; Завершить и оставить резидентной (TSR)
        MOV   AX, 3100H
        MOV   DX, (init –  begin +10FH)/16  ; Размер резидента
        STI        ; Разрешение прерываний
        INT    21H
; Данные части инициализации
    MSG   DB 'Start TSR', 10,13,'$'
    CODEPR  ENDS
      END   BEGIN.
```

В данной программе вывод сообщения о старте резидента выводится с помощью функции DOS (09H – 21H), а вывод цепочки символов с помощью прерывания BIOS (0AH – 10H), так как использование функций DOS в резидентной части ограничено и практически недопустимо (!!!).

Компиляция такой программы (fast.asm) выполняется командным файлом, содержащем следующие директивы

```
tasm /l /zi fast.asm
tlink /v /t /l fast.obj
PAUSE.
```

Директива PAUSE необходима для контроля наличия ошибок в программе. После запуска резидента при нажатии клавиши **ESC** на экране будет выведена следующая строка

```
C:\work>fast.com
Start TSR .
```

После нажатия клавиши ESC на экран в командную строку будет выведено:

```
C:\work>\BBBBBBBBB .
```

Должно было быть выведено 10 символов, а на самом деле всего 9. Это объясняется тем, что при нажатии ESC на экран выводиться символ '\', а так как в BIOS прерывании с повторением курсор не перемещается, то символ '\' заменяет первую букву 'B'.

Чтобы исправить эту “ошибку” необходимо выполнить полностью обработку прерывания, включая выдачу сигналов клавиатуре о прочитанном символе, а контроллеру прерывания сигнала о завершении обработки прерываний. Для обработки клавиатурных прерываний это существенно. В следующем фрагменте показано, как это сделать.

```
; Сохранение регистров перед выводом на экран цепочки
        PUSH AX
        PUSH BX
        PUSH CX .
; Выдача сигнала через порт контроллера 8255 о чтении скан-кода
        IN AL, 61H
        OR AL, 10000000b   ; установим бит 7 порта B
        OUT 61H, AL
        AND AL, 01111111b
        OUT 61H, AL     ; сбросим бит 7 порта B (символ прочитан)
;  Вывод цепочки 10 символов
        MOV AH , 0AH
        MOV AL, 'B'     ; символ ’B’
        MOV BH , 0
        MOV CX, 10      ; Число символов
        INT 010H        ; прерывание BIOS DOS напрямую
        POP CX          ; восстановление регистров
        POP BX
        POP AX
;  Сигнал контроллеру прерываний через порт 20H сигнал (EOI = 20H)
        MOV AL,20H
        OUT 20H , AL     ; в порт ведущего контроллера 8259A
```

POP AX

IRET ; завершение нового обработчика при вводе ESC .

В результате мы получим строку, которая расположена ниже, причем курсор будет располагаться в первой позиции, так как 0A– 010H не переводит курсора

C:\work>BBBBBBBBBB.

Теперь продолжим рассмотрение других особенностей построения и реализации резидентных программ.

12.12 Работа с вектором прерываний напрямую

Работать с вектором прерываний можно напрямую. В реальном режиме доступна вся память, достаточно задать длинный адрес (сегмент смещение). Адрес расположения зависит от номера прерывания, достаточно этот номер умножить на 4 и получим необходимое смещение. Значение сегментного регистра (ES) в этом случае должно быть нулевым. Необходимо также учесть, что в памяти двойное слово по словам располагается в обратном порядке, с начала смещение, а затем сегмент. Текст этих фрагментов программы приведен ниже.

Запоминание старого обработчика

MOV AX, 0

PUSH AX

POP ES ; задание в ES значения 0

MOV BX, WORD PTR ES: 09h*4 ; считаем первое слово

MOV WORD PTR CS:OLD_09, BX ; запомним его в поле старого обработчика

MOV BX, WORD PTR ES: 09h*4 + 2 ; считаем второе слово

MOV WORD PTR CS:OLD_09+2, BX ; запомним его в поле старого обработчика.

Запись нового обработчика.

MOV AX, 0

PUSH AX

```
POP ES
LEA BX , NEW_09
MOV WORD PTR ES: 09h*4, BX
MOV BX, CS
MOV WORD PTR ES: 09h*4 + 2, BX.
```

Несмотря на рассмотренную возможность, более корректно использовать для этих целей функции 035H и 025H прерывания 21H.

12.13 Выгрузка резидента

При выгрузке резидентной программы из оперативной памяти должны быть выполнены следующие действия:

− выполняются необходимые действия для завершения резидента (закрытие файлов, сброс системных флагов и другие необходимые действия);

− восстановлены все старые обработчика в векторе прерывания;

− освобождена оперативная память, выделенная под резидентную программу и под сопровождение процесса (окружение программы).

Отдельные действия конкретного резидента зависят от существа задачи, поэтому здесь мы не будем рассматривать. Действия по восстановлению векторов прерываний аналогичны действиям по установке резидента и базируются на сохраненных адресах старых обработчиков. Они выполняются так

```
 PUSH DS
; Восстановление 9H
 mov AX,2509H ; Восстановление обработчика прерывания 05H
 lds DX ,CS:SAVEINT9
 int 21 H
; Восстановление 2FH
 mov AX ,252FH
; Восстановление обработчика прерывания 2FH
 lds DX,CS:SAVEINT2F
 int 21 H
```

POP DS .

Команда **LDS** позволяет загрузить одновременно на основе длинного адреса регистры DS и DX, необходимые для выполнения функции 25H. Так как регистр DS затирается его нужно сохранить в стеке и восстановить.

Действия по освобождению памяти выполняются с помощью функции 49H прерывания DOS. При этом на регистре ES должен быть установлен сегментный адрес начала выделенной области (или блока – ОС выделяет память блоками, которые идентифицируются по началу блока). В начале резидента располагается область PSP (Program Segment Prefix см. в справочниках), которая содержит сегментный адрес области окружения DOS со смещением 2CH (DOS environment). Выгрузка самой программы выполняется на основе сегментного регистра кода, который сформирован при вызове самого резидента. Код ниже иллюстрирует эти операции.

```
; получим из PSP адрес собственного окружения и выгрузим окружение DOS
    MOV ES,CS:2CH
    MOV AH,49H
    INT 21H
; выгрузим теперь саму программу
    PUSH CS
    POP      ES
    MOV AH,49H
    INT 21H
    IRET
```

После действий по освобождении памяти никакие действия в обработчике нельзя считать корректными.

12.14 Разбор параметров командной строки

Ниже приведены требования при инициализации параметров командной строки.

1. Размер командной строки задан в программе со смещением 80H (область PSP). Это однобайтовое поле. Если его значение равно 0, то параметров в командной строке нет. Максимальный размер этого поля параметров равен 127 байт.

2. Начиная с CS:81H расположена сама строка параметров, которые нужно проверять, включая, кстати, и сброс пробелов. Просмотр строки лучше выполнять в цикле. В алгоритмах рекомендую использовать команды LOASB и STOSB.

3. Алгоритм просмотра реализуется в цикле и с учетом строгой и нестрогой последовательности параметров в задании, а так же разных регистров.

4. Для доступа к параметрам окружения DOS (SET переменные, подробнее в справочнике) необходимо использовать дальний адрес, расположенный в PSP со смещением 2CH (точнее CS:2CH).

Ниже приводится фрагмент программы для анализа наличия параметров в командной строке.

```
MOV AL ,  CS:80H
CMP AL, 0      ; проверить число байт параметров командной строки
JNE INIT_UND     ; перейти на метку если параметров нет.
```

12.15 Контроль наличия резидента (другой способ)

Проверка наличия может быть выполнена и другим способом. По адресу резидента проверяем заданную константу – пароль. Адрес поля пароля можно вычислить на основе адреса обработчика прерывания (например, **INT_09**).

```
; это часть резидента
…
PASSW DW 62627
INT_09: …
```

Далее в программе инициализации вычисляем адрес резидента с помощью функции DOS 35H. Адрес получаем на регистрах – ES:BX.

```
; это часть инициализации
MOV AH, 35h
MOV AL , 09h ; номер обработчика
```

```
INT 21h
SUB BX , 2
MOV AX , WORD PTR ES:BX
CMP AX , PASSW
JNE NOLOAD ; метка обработки отсутствия в памяти.
```

Если код, записанный в поле PASSW и код, полученный из резидента по этому прерыванию совпадают, то с большой долей вероятности можно считать, что резидент уже в памяти. Если совпадения нет, то резидент отсутствует.

Такая проверка возможна потому, что в оперативной памяти может одновременно находиться две резидентные части: одна – загруженный резидент, а вторая часть основной программы.

12.16 Связь с резидентом с помощью клавиатуры

В каждом варианте РГЗ предусмотрена обработка прерывания от клавиатуры. Прерывание от клавиатуры является аппаратным (BIOS прерывание – 09H), для его обработки необходимо установить собственный обработчик. Обработка этого прерывания может быть выполнена:

− с использованием портов ввода/вывода (60H , 61H и 20H);

− с использованием стандартного обработчика прерываний;

− с помощью комбинированного использования портов и стандартного обработчика прерываний.

При использовании стандартного обработчика прерываний можно использовать специальный буфер клавиатуры и специальные функции прерывания BIOS для работы с клавиатурой (16H).

Для инициации обработки клавиатуры необходимо, основываясь на изложенной выше информации, установить собственный обработчик для прерывания 09H. При работе с буфером клавиатуры можно использовать область системной памяти, которая расположена в начале, ОП с сегментным адресом 040H (смещение – 0400H) сразу после вектора прерываний. Буфер клавиатуры является циклическим, каждый его элемент содержит 2 байта. Таким образом, максимально в

нем может быть расположено 16 символов (точнее нажатий клавиш, включая и их комбинации). Такое необходимо, так как обработчики прерывания могут не успеть обработать все нажатия. При переполнении циклического буфера выдается сигнал через динамик и новые нажатия клавиш в нем не фиксируются. Для работы с буфером клавиатуры предусмотрено два указателя на начало (голову списка – Head) и на конец (хвост списка – Tail). Если эти указатели равны, то список пуст.

Кроме того, в системной области фиксируется информация о нажатии управляющих клавиш (таких как Alt, Ctrl и др.) Расположение специальных полей в системной области (0400H), позволяющих работать с буфером клавиатуры (более подробно содержание системной области смотрите в справочниках – General Memory Map) показано ниже:

− по адресу **0417h** – находится информация (размером в 2 байта) о нажатии специальных управляющих клавиш (Ctrl, Alt, Shift и др.), подробнее о содержании смотрите в справочниках;

− по адресу **041Ah** – находится адрес головы циклического буфера (2 байта);

− по адресу **041Ch** – находится адрес головы циклического буфера (2 байта);

− по адресу **041Eh** (32 – байта) – циклический буфер клавиатуры.

На основе этой информации можно установить код символа, стоящего в очередь на обработку, и информацию о нажатии управляющих клавиш. Использование буфера клавиатуры возможно после вызова стандартного обработчика прерывания от клавиатуры.

12.16.1 Работа через порт. При использовании порта ввода/вывода можно прочитать код нажатой клавиши в порта А контроллера 8255 (60H). Это может быть выполнено в программе собственного обработчика с помощью следующей команды:

IN AL , 060H ; чтение скан–кода клавиши.

Полученный скан– код (см. раздел – 23.5. SCAN – коды), можно использовать в программе. Для получения признака нажатия управляющих клавиш (Shift,

Ins и др. см. в справочнике) можно использовать специальное прерывание BIOS. В примере, расположенном ниже проверяется нажатие клавиши Shift совместно с клавишей ‘Q’.

```
IN   AL, 60H    ; Взять из порта 60H скан-код на регистр AL
CMP AL , 10H  ; Проверка нажатия ’Q’, скан код не зависит от регистра
JNE  OLD       ; Переход на стандартный обработчик
; проверка клавиши Shift (первый бит в байте состояния управляющих клавиш)
MOV AH , 02H
INT 016H       ; Получим управляющий байт на AL
TEST AL , 00000010b  ; Проверим на 1
JZ OLD    ; Переход на стандартный обработчик, если 0 .
```

Для завершения самостоятельной обработки необходимо через порт B контроллера 8255 (61H) передать информацию о считывании скан-кода. Для этого нужно выполнить следующие команды:

```
IN AL, 61H      ; Считаем старое состояние
OR AL, 10000000b    ; Установим 7– й бит в 1
OUT 61H, AL      ; Запишем в порт
AND AL, 01111111b   ; Сбросим 7– й бит
OUT 61H, AL     ; Запишем в порт.
```

Данные команды позволяют подтвердить прием скан-кода из клавиатуры. Если далее нам не нужно обрабатывать стандартным обработчиком данное прерывание, то необходимо передать сигнал (20H – EOI END OF INTERUPT) в порт контроллер прерываний. Это делается командами

```
MOV AL,20H
OUT 20H,AL
POP AX
IRET
```

Если нужно завершить обработку с системным обработчиком прерываний, то нужно ему передать управление:

```
OLD: POP AX
STI
```

```
JMP   CS:SAVEINT9   ; Вызов старого обработчика.
```

12.16.2 Работа с буфером клавиатуры. При работе с буфером клавиатуры целесообразно вызвать старый обработчик предварительно. Это можно сделать так:

```
NEWINT9:
PUSH  AX       ; Сохранение используемых регистров
PUSH CS        ; для работы с данными
POP  DS
PUSHF          ; Это обязательно для корректного возврата
CALL CS:SAVEINT9   ; Вызов старого обработчика.
```

Для полноты работы с клавиатурой нужно иметь возможность получить информацию об управляющих клавишах и значение введенного кода (скан-кода, ASCII кода и расширенного ASCII кода). Для проверки нажатия управляющих клавиш можно прочитать специальный байт по адресу 0417 из системной области. Это можно сделать так, как показано ниже.

```
; проверка левого шифта
    MOV AX , 40H   ; сегментный адрес системной области
    MOV ES , AX
    MOV AL , ES:17H  ; смещение для управляющего байта состояния клавиш
    TEST AL , 00000010b ; проверка второго бита
    JZ PROD    ; переход если левый шифт не нажат.
```

Такие действия можно выполнить и по-другому, основываясь на прерывании BIOS 02H – 16H.

```
    MOV AH , 02H
    INT  16H   ; BIOS прерывание для получения управляющего байта
    TEST AL , 00000010b ; проверка второго бита
    JZ PROD    ; переход если левый шифт не нажат.
```

Если символ, прочитанный с клавиатуры, не должен появляется на экране (например, в окне COMMAND.COM) необходимо обнулить буфер клавиатуры. Это можно сделать, установив одинаковые значения указателей циклического буфера. Например, так

```
MOV AX , 40H
MOV ES , AX
MOV BX , ES:1AH
MOV ES:1CH , BX.
```

После такой операции введенный символ не будет выводиться на экран командного процессора. Это можно сделать и так

```
MOV AH , 00H  ; код команды чтения символа и выборки из буфера
INT  16H   ; BIOS прерывание для получения управляющего байта.
```

Если буфер клавиатуры пуст, то программа будет ожидать нажатия клавиши. Из буфера клавиатуры можно прочитать коды нажатой клавиши или их комбинации напрямую. Это может быть сделано так

```
MOV AX , 40H
MOV ES , AX
MOV BX , ES:1AH
MOV AX, ES:[BX].
```

После такой операции на регистрах AH и AL заносится комбинация кодов, на основе которых можно выделить следующее: ASCII символ, скан-код или расширенный ASCII код. Это можно сделать и так

```
MOV AH , 01H  ; код команды чтения символа и без выборки из буфера
INT  16H   ; BIOS прерывание для получения управляющего байта.
```

Если регистр AL содержит 0, то это значит, что в регистр AH занесен расширенный ASCII код, который можно анализировать. Иначе регистр AL содержит ASCII введенного символа, а регистр AH содержит скан-код нажатой клавиши.

Предварительно необходимо вызвать системный обработчик прерываний. Анализ введенного кода для проверки нажатой клавиши может быть проведен так

```
MOV AX , 40H
MOV ES , AX
MOV BX , ES:1AH
MOV AX, ES:[BX] ; на AX 2 байта их буфера клавиатуры
; Проверка расширенного кода
CMP AL , 0   ; проверка отсутствия ASCII
```

```
    JE PRINT_EXT  ; переход, если расширенный код
    CMP AL ,  61H ; Проверка символа ’a’
    JNE EXIT
    CMP AH, 1EH   ; Проверка нужного скан-кода для клавиши ’A’
    JNE EXIT
  …
  PRINT:
   …
; Проверка комбинации расширенного кода
  PRINT_EXT:
    CMP AH , 5EH  ; Ctrl + F1
    JNE EXIT   ; Переход если не наша комбинация
    …
    JMP PRINT.
```

В данном примере сначала на AX из буфера клавиатуры получаем два байта. Затем проверяем AL на 0. Нулевое значение свидетельствует о наличии расширенного кода ASCII и мы переходим на метку для его анализа (PRINT_EXT). Там мы проверяем код комбинации Ctrl + F1(код – 5E). Если код ASCII (AL != 0), то проверяем нужный код в регистре AL (у нас 61H – код строчной буквы ‘a’). Для иллюстрации наличия скан-кода клавиши ‘A’ делаем дополнительную проверку регистра AH, куда помещен скан-код этой клавиши – 1EH. Если код не соответствует, то уходим на выход из процедуры.

В практике программирования и в РГЗ можно выбрать любой из вариантов для обработки прерываний от клавиатуры.

12.17 Освобождение памяти внешнее из отдельной программы

Освобождение памяти под резидент и восстановление векторов прерываний старых обработчиков прерываний. Вы можете выполнить и вне резидента, в отдельной программе. Для этого нужно вычислить адрес начала резидентной программы и PSP, а затем выполнить действия по восстановлению старых векторов

прерываний и освобождению оперативной памяти. Трудности может составить получение адресов сохраненных старых адресов вектора прерываний.

Если разместить переменную для сохранения старого адреса обработчика непосредственно перед точкой входа в процедуру обработки прерываний, то получить этот адрес несложно, вычитая из этого адреса четыре.

```
SAVEINT9 DD ?
; Сохранение старого обработчика
;  Новый обработчик прерывания 09H
NEWINT9:
PUSH  AX
; Сохранение используемых регистров.
```

Начальная точка расположения резидентной программы и адрес ее PSP могут быть получены аналогично.

12.18 Завершение основной программы при проверке повторной загрузки

При проверке повторности загрузки в части инициализации, программа завершается обычным образом, если она не остается резидентной. Аналогичное завершение должно быть при выдаче справке о программе. Завершение основной программы в этих случаях может быть выполнена так

```
MOV AX , 4C00H
 INT 21 H ,
```

на AL задается код возврата программы (ERRORLEVEL), или

```
 MOV AX , 0000H
 INT 20 H.
```

12.19 Проверка загрузки и выгрузки с помощью утилиты mem.exe

Для контроля наличия резидента в ОП нужно использовать утилиту ОС **MEM.EXE.** Эта утилита используется несколько раз: с ее помощью уточняется объем свободной памяти до запуска резидента; проверяется наличие резидента в

ОП и проверяется освобождение ОП под модули резидента после его выгрузки. Для этого необходимо сначала уточнить параметры ее запуска, так как для Разных ОС они могут отличаться. Поэтому ее нужно запустить в режиме справки (MEM.EXE/?). Размер свободной памяти до загрузки резидента должен полностью соответствовать размеру свободной памяти после его выгрузки. Запуск в режиме справки для XP:

C:\work>mem /?

Справка на экране:

Вывод сведений об используемой и свободной памяти.

MEM [/PROGRAM | /DEBUG | /CLASSIFY]

/PROGRAM or /P Вывод статуса программ, загруженных в память.

/DEBUG or /D Вывод статуса программ, внутренних драйверов и другой дополнительной информации.

/CLASSIFY or /C Классификация программ по использованию памяти.

Вывод сведений о размерах программ, использовании памяти и максимальном свободном блоке памяти.

Если предварительно загрузить резидент **(TSRKBD.COM)** и проверить распределение памяти командой:

C:\work>mem /p

Мы получим на экране информацию, как показано на рисунке 12.5.

```
Адрес   Имя     Размер   Тип
-------  --------  ------  ------
000000       000400   Вектор прерывания
000400       000100   Область обмена ПЗУ (ROM)
000500       000200   Область обмена DOS
000700  IO     000370   Системные данные
000A70  MSDOS   0017A0   Системные данные
002210  IO     002280   Системные данные
...
0044A0  COMMAND  000B50  Программа
005000  MSDOS   000070  – Свободно –
005080  COMMAND  0005E0  Окружение
005670  DOSX    0087A0  Программа
00DE20  COMMAND  000510  Данные
00E340  COMMAND  000B50  Программа
00EEA0  COMMAND  0004F0  Окружение
00F3A0  TSRKBD   000510  Окружение
00F8C0  TSRKBD   000160  Программа
00FA30  MEM     000510  Окружение
00FF50  MEM     0174E0  Программа
027440  MSDOS   078BA0  – Свободно –
...
0D4310  MSDOS   00BCE0  – Свободно –
 655360 байт – всего обычной памяти
 655360 байт – доступно для MS– DOS
 589968 максимальный размер исполняемой программы
 1048576 байт – всего непрерывной дополнительной памяти
 0 байт – доступно непрерывной дополнительной памяти
  941056 байт – доступной памяти XMS резидентная часть MS– DOS
загружена в сегмент HMA.
```

Рисунок 12.5 – Вывод сведений о размерах программ

На этом примере видно, что под область резидентной программы выделяется два фрагмента ОП, которые необходимо освобождать (программа и окружение). Количество свободной памяти в этом режиме определяется строкой: «589968 максимальный размер исполняемой программы».

12.20 Описание данных и процедур резидента

Описание данных и процедур, используемых в резидентной части, нужно размещать только в области самого резидента

```
BEGIN: …
    JMP INIT
; Описания данных
…
; обработчики прерываний и процедуры
…
; Описания процедур резидента
MY_PROC PROC
…
MY_PROC ENDP
…
; Описания данных
…
MSG DB …
INIT: … .
```

Если данные для резидента случайно описать в области части инициализации, они не будут доступны и программа резидента не сможет работать правильно.

12.21 Русификация сообщений резидента

Для того чтобы резидент мог выдавать сообщения на русском языке необходимо выполнить следующие условия:

− загрузить перед запуском резидента русификатор (например, RKM);

− сообщения резидента закодировать в коде ASCII, для чего их необходимо ввести в текстовом редакторе DOC при включенном русификаторе. В этом случае в текстовом редакторе под WINDOWS данные сообщения правильно читаться не будут.

Кстати, используя русификатор, Вы дополнительно проверяете правильность своей программы в части вызова старого обработчика прерывания. Необходимо корректно вызвать старый обработчик. В тех случаях, когда используется полная обработка прерывания собственным резидентов, ввод русских символов с клавиатуры становиться недоступным.

12.22 Автономная программа для выгрузки TSR

В вариантах курсовой работы предусмотрен случай выгрузки резидента с помощью специальной автономной программы. В этом варианте необходимо создать программу, которая обращается к резиденту с помощью прерывания 2D (2F). С помощью этого прерывания сначала нужно проверить наличие резидента в памяти, а затем выдать сигнал выгрузки. Далее показан пример программы, в которой выполняются такие действия.

```
; Программа выгрузки резидента (unload.asm)
        MYCODE SEGMENT 'CODE'
         ASSUME CS:MYCODE
START:
; Загрузка сегментного регистра данных DS
         PUSH CS
          POP DS
; Проверка наличия резидента в ОП
```

```
        MOV AH, 0EEh
        MOV AL, 1  ; Проверка наличия
        INT 2Dh
        CMP AL, 0FFh
        JNE NO_LOAD ; переход если резидент не(!) в памяти
; Вызрузка
        MOV AH , 0EEH
        MOV AL,2
        INT 2Dh
        CMP AL , 0FFH       ; проверка ответа от нашего резидента должно быть
0FFH
        JNE ERR
; Сообщение об успешной выгрузке
        MOV AH , 09H
        MOV DX, OFFSET MSG_UNLD
        INT  21H
        JMP EXIT
;  Ошибка выгрузки
 ERR:
        MOV AH , 09H
        MOV DX, OFFSET MSG_ERR
        INT  21H
        JMP EXIT
; Сообщение о том, что уже загружен
        NO_LOAD:
        MOV AH , 09H
        MOV DX, OFFSET MSG_NOUNLD
        INT  21H
; Выход из программы
EXIT:
        MOV AL, 0
```

```
        MOV AH, 4CH
        INT 21H
;  Сообщения
        MSG_NOUNLD   DB 'TSR is NO in memory!', 10,13,'$'
        MSG_UNLD DB 'TSR was unloaded (Unload programm)!', 10,13,'$'
        MSG_ERR DB 'Ошибка выгрузки резидента!', 10,13,'$'   ; ASCII!!!
        MYCODE ENDS
        END START .
```

Компиляция данной программы должна быть выполнена в формате .EXE. Мы получим **UNLOAD.EXE.** Запуск этой программы при загруженном резиденте приводит к следующей реакции в командной строке

```
C:\work> unload
C:\work>TSR was unloaded (Unload programm)!
C:\work>_
```

Если резидента в памяти не обнаружено, то получим:

```
C:\work> unload
C:\work> TSR is NO in memory!
C:\work>_ .
```

Выгрузку можно выполнить и другим способом, рассмотренным выше, без использования обработчика собственного прерывания (2D), при этом должны быть восстановлены старые обработчики и освобождена оперативная память.

12.23 Пример резидентной программы

В приложении Б приводиться текст простой (функционально) резидентной программы, которая содержит все системные требования, заданные в расчетно-графическом задании студентов по дисциплине «Программирование МПС».

Данная программа, после выполнения загрузки резидента, при нажатии на функциональную клавишу F1 выводить на экран дисплея сообщение –“ TSR: F1 – pushed!!!”. Перечень основных функций перечислен в заголовке программы

После компиляции и редактирования связей в формате .COM получим исполнимый модуль TSRPOS.COM.

Предварительно необходимо проверить размер свободной памяти и его запомнить

C:\work>mem

…

591664 максимальный размер исполняемой программы

… .

При запуске этого модуля без параметров резидент будет загружен в оперативную память.

C:\work>tsrpos

C:\work>Start TSR program!

C:\work>_ .

Повторный запуск исполнимого файла **TSRPOS.COM** приводить к выдаче сообщения на экран

C:\work>tsrpos

C:\work>TSR already is in memory

C:\work>_ .

Проверка с помощью утилиты MEM должна показать наличие резидента и его окружения в памяти (напомню, что перед запуском резидента необходимо с помощью утилиты MEM проверить количество свободной памяти и его запомнить!)

C:\work>mem /p

Адрес	Имя	Размер	Тип
000000		000400	Вектор прерывания
000400		000100	Область обмена ПЗУ (ROM)
…			
00DE20	COMMAND	000510	Данные
00E340	COMMAND	000B50	Программа

00EEA0 COMMAND 0004F0 Окружение

00F3A0 TSRPOS 000510 Окружение

00F8C0 TSRPOS 000260 Программа

00FA30 MEM 000510 Окружение

00FF50 MEM 0174E0 Программа

…

589968 максимальный размер исполняемой программы.

При запуске программы в режиме справки резидент не загружается и не удаляется из памяти. На экран выводиться следующее сообщение

C:\work>tsrpos /H

Test – sample TSR – 2010

Parameters: /H or /h – help , /U or /u – unload TSR

Информация!

C:\work>_ .

При нажатии на клавишу F1 резидент выдает следующее сообщение в текущую позицию курсора дисплея (курсор расположен после последнего символа)

C:\work>TSR: F1 – pushed!!!_ .

При использовании комбинации Ctrl+F1 резидент будет выгружен с помощью своих процедур

C:\work> TSR was unloaded (Ctrl+F1 – TSR part)!_.

Повторная проверка с помощью MEM должна показать отсутствие резидента в памяти, а размер свободной памяти должен восстановиться к начальному состоянию

C:\work>mem

…

591664 максимальный размер исполняемой программы .

Числовые значения для различных компьютеров могут отличаться от величин, приводимых здесь (это зависит от конкретной конфигурации ОС и установок конкретного компьютера), но значения, полученные до загрузки резидента и после его выгрузки должны совпадать в точности до единицы (у нас в примере 591664 = 591664).

Выгрузку резидента можно выполнить в нашем случае и из командной строки

```
C:\work>tsrpos /U
TSR was unloaded (Init part)!
C:\work>_.
```

Контроль выгрузки с помощью MEM здесь выполняется аналогично. При задании ошибочных параметров будет выдано сообщение

```
C:\work>tsrpos /L
Error –  command line parameters!
C:\work>_ .
```

В отчете по РГЗ должна быть приведена программа данных проверок.

12.24 Задание к РГЗ

1. Через временной интервал (например, 1 минута) на экран выводится какое-либо сообщение. Через 10-20 секунд сообщение снимается и работа ПЭВМ продолжается обычным образом.
2. При нажатии любой клавиши на экран выдается просьба нажать эту клавишу еще раз. При повторном нажатии просьба с экрана снимается и работа ПЭВМ продолжается обычным образом.
3. При нажатии клавиши ENTER в центр экрана выводится сообщение: "Отдыхаю, подождите минутку". Через 10-20 секунд сообщение снимается и работа ПЭВМ продолжается обычным образом.
4. ПЭВМ реагирует на клавишу "стрелка-вверх" как на клавишу "стрелка-вниз" (и наоборот), на клавишу "стрелка-влево" как на клавишу "стрелка-вправо" (и наоборот).
5. При нажатии клавиши F1 программа очищает экран и безостановочно выводит на экран сообщение "Не хочу вам помогать!", прокручивая при этом экран вверх. Секунд через 10-20 этот процесс прекращается, восстанавливается экран и работа ПЭВМ продолжается обычным образом.

6. Через равные промежутки времени (например 30 секунд) резидент блокирует/разблокирует клавиатуру.

7. При нажатии клавиши T сообщается текущее системное время.

8. Нажатие клавиши D замедляет/восстанавливает реакцию системы на нажатие клавиш клавиатуры. Замедление реакции должно быть достаточным для того чтобы его можно было заметить визуально.

9. При нажатии клавиши R очищается правая, а при нажатии L-левая половина экрана. Через 10-20 секунд после нажатия любой из этих клавиш экран восстанавливается.

10. Нажатие клавиши G производит переключение видеоадаптера между текстовым и графическим режимом.

11. Нажатие клавиши I инвертирует цвета экрана.

12. После установки резидента система перестает реагировать на нажатие клавиши F7. На другие клавиши система реагирует обычным образом.

13 Программирование микроконтроллеров

Изначально микроконтроллеры программировались исключительно на различных языках ассемблера, ориентированного на конкретное устройство. Можно сказать, что такие языки представляли собой символьные мнемоники соответствующих машинных кодов, а перевод мнемоники в машинный код выполнялся транслятором. Это требовало знаний всех особенностей архитектуры конкретного типа микроконтроллера и логики его работы, но зато позволяло создавать оптимальные по объему и быстродействию программы.

В процессе развития микроконтроллеров решаемые ими задачи значительно усложнились, а их объем памяти и быстродействие резко увеличились. Это привело к тому, что при разработке программного обеспечения для микроконтроллеров стали в основном использовать языки высокого уровня, в частности, язык Си.

В данном разделе приводятся краткие сведения о языке Си и компиляторе Image Craft, достаточные для того, чтобы приобрести навыки программирования микроконтроллеров и разрабатывать микропроцессорные устройства, требующие несложного программного обеспечения. Для более углубленного изучения языка Си необходимо обратиться к специальной литературе.

13.1 Интегрированная среда разработки ICCAVR

Среда программирования ICCAVR фирмы Image Craft Creations Inc предназначена для разработки программного обеспечения (проекта) и его компиляции в исполняемый AVR-микроконтроллером файл. ICCAVR является одним из простейших и очень удобных компиляторов. Он имеет набор стандартных библиотечных функций и ряд подпрограмм, специально предназначенных для AVR-микроконтроллеров. Их можно использовать в своем проекте, предварительно подключив их к проекту директивой **#include.**

В работе ICCAVR используются следующие типы файлов (по расширениям):

1) C – исходный текст на языке Си;

2) S – исходный текст на ассемблере или выходной ассемблерный файл, генерируемый для каждого исходного С-файла;

3) H – заголовочный (header) файл;

4) PRJ – файл проекта;

5) SRC – список файлов проекта;

6) O – объектный файл, получаемый после компиляции ассемблерного файла;

7) HEX – выходной файл в формате Intel HEX для загрузки в ПЗУ программ микросхемы;

8) EEP – выходной файл в формате Intel HEX для загрузки в ПЗУ данных микросхемы;

9) COF – выходной файл в формате COFF, используется при отладке проекта в AVR Studio или других средах программирования;

10) LST – файл-листинг, содержащий информацию об адресах;

11) MP – MAP-файл, содержащий символическую информацию;

12) DBG – файл с отладочной информацией;

13) A – библиотечный файл.

В среде ICCAVR пользователь разрабатывает проект (файл с расширением PRJ), состоящий из одного или нескольких исходных С-файлов и заголовочных Н-файлов.

В проект могут также входить текстовые файлы с дополнительной информацией, например, об особенностях работы алгоритма программы или какой-то справочной информацией. При успешной компиляции программы образуется HEX-файл, который с помощью специальных средств «прошивается» в память программ микроконтроллера. HEX-файл можно также использовать в системах схемотехнического моделирования (например, в PROTEUS) для проверки правильности функционирования микроконтроллера.

После запуска ICCAVR на экране монитора появляется главное окно, которое разделено на три части (окна), как представлено на рисунке 13.1. Левая верхняя часть – окно редактора.

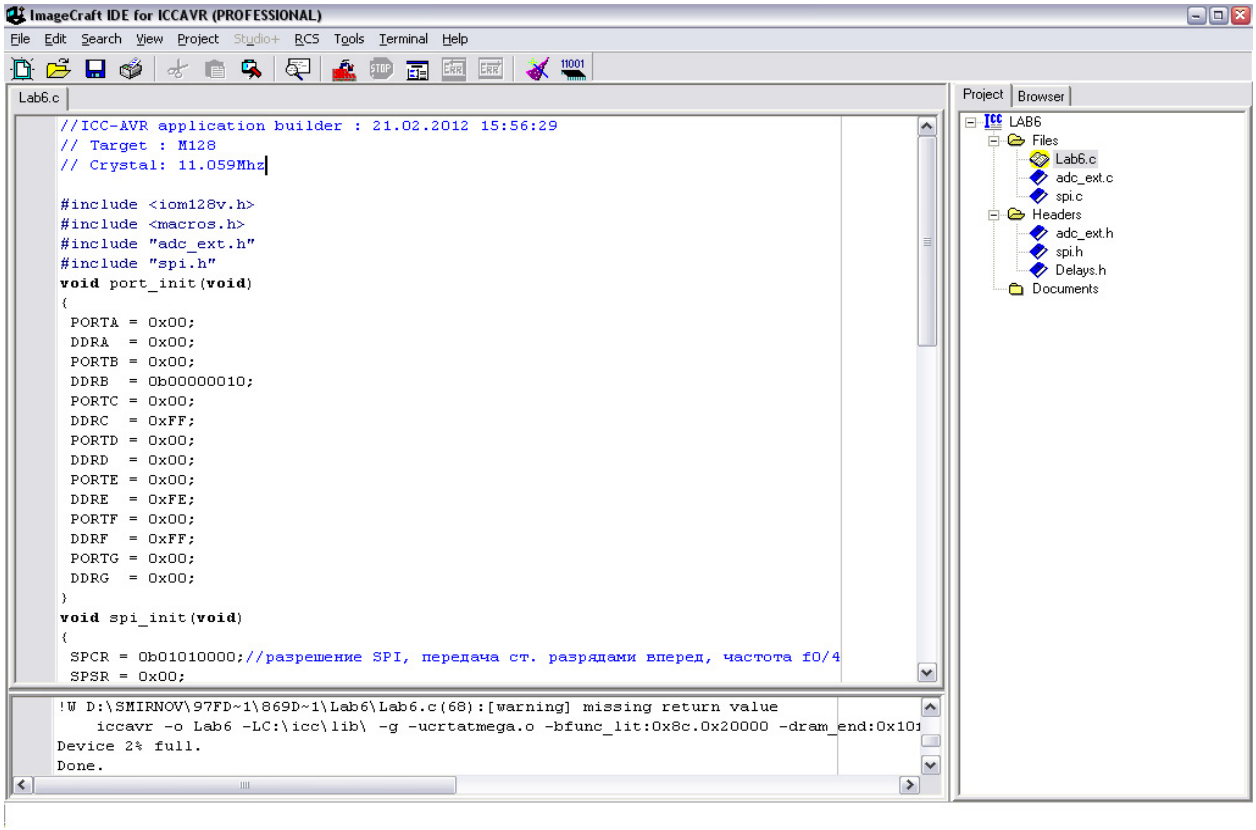


Рисунок 13.1 – Главное окно ICCAVR

В окне редактора представлены тексты программ, которые образуют проект. Именно с этими текстами и работает программист, редактируя их или добавляя новые строки и блоки программ.

Правая часть – окно менеджера проекта, содержащее две вкладки: Project и Browser. В окне Project находится список файлов проекта, каждый из которых открывается двойным щелчком мыши. В окне Browser находится обозреватель кода, показывающий список функций и переменных, определенных в проекте. Если дважды нажать на функцию в обозревателе кода, курсор на определение этой функции в исходном файле.

Нижняя часть – окно состояния. В нем показываются результаты компиляции отдельного файла или всего проекта в целом. Если имеются ошибки, то после компиляции появятся сообщения об обнаруженных ошибках с указанием их места расположения и подсказкой о типе ошибки. Как и в любом языке программирования, компилятор проверяет только ошибки синтаксиса программы, а не правильность ее алгоритма.

Все свои действия по управлению проектом или отдельными файлами пользователь осуществляет, используя пункты меню в верхней части главного окна, кнопки панели инструментов (под пунктами меню) или правую кнопку мыши и всплывающее при этом контекстное меню. Рассмотрим кратко наиболее важные пункты меню.

Пункт меню **File** позволяет открыть какой-либо файл или создать новый, сохранить, закрыть или вывести на печать и т.д. Одним из его подпунктов является пункт **Reopen...**, который позволяет выбрать и загрузить один из последних файлов, с которыми работал пользователь.

Edit позволяет осуществить действия по редактированию текста открытого в окне файла: скопировать в буфер или вставить из него фрагмент текста, «вырезать» выбранный фрагмент текста в буфер или удалить его совсем, отменить последнее исправление и т.д. Все эти операции также можно осуществлять нажатием соответствующих комбинаций клавиш.

Search позволяет находить заданные пользователем слова в тексте открытого файла проекта или во всем проекте, заменять отдельные фрагменты текста на другой текст, перемещать курсор на нужный номер строки текста открытого файла или на нужную метку, добавлять или удалять метки и т.д.

Project позволяет открыть какой-либо проект или создать новый; открыть или закрыть все файлы проекта; открыть один из последних проектов, с которым работал пользователь; закрыть проект или сохранить его под новым именем; добавить открытый в окне файл в проект или удалить из проекта файл, который выбран в окне менеджера проекта. Подунктом **Make Project** позволяет осуществить компиляцию открытого в окне редактора файла, а подпункт **Rebuild All** – компиляцию всех файлов проекта. Важным пунктом является **Option...**, который открывает диалоговое окно, которое представлено на рисунке 13.2.

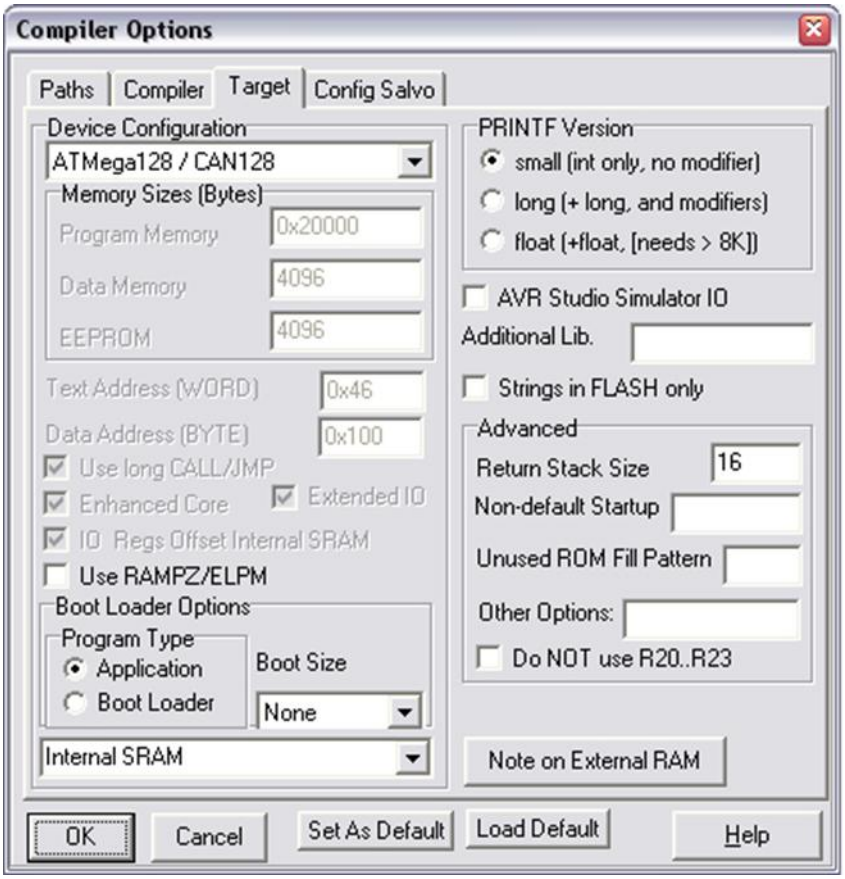


Рисунок 13.2 – Окно при выборе пункта меню Option

В пункте **Target** в окне **Device Configuration** следует задать микроконтроллер, для которого разрабатывается проект, например ATmega128/CAN128 (см. рис. 2.2). В пункте **Paths** необходимо установить пути к библиотечным файлам и файлам, которые пользователь подключает к проекту, например, для библиотечных файлов **c:\icc\lib**. В пункте **Compiler** в окне **Output Format** установить формат выходных файлов – COFF/HEX.

В пункте **Tools** главного окна важными являются три пункта: **Editor Options, Application Builder** и **In System Programmer**. Пункт **Tools>Editor Options** позволяет установить нужные опции для редактора текста. Здесь лучше все оставить без изменений. Но если возникает необходимость писать комментарии в программе на русском языке, то в пункте **Tools>Editor Options> Highlighting** необходимо в окне **Characters** установить опцию **Russian.**

Пункт **Tools>Application Builder** предназначен для инициализации периферийных устройств микроконтроллера, а также для настройки системы внешних прерываний. Настройку периферийных устройств можно выполнить и вручную,

присвоив в программе соответствующим регистрам управления и статуса нужные значения. Однако с помощью модуля **Application Builder** эта инициализация осуществляется гораздо быстрее и надежнее. Окно **Application Builder** с опциями настройки портов ввода/вывода представлено на рисунке 13.3.

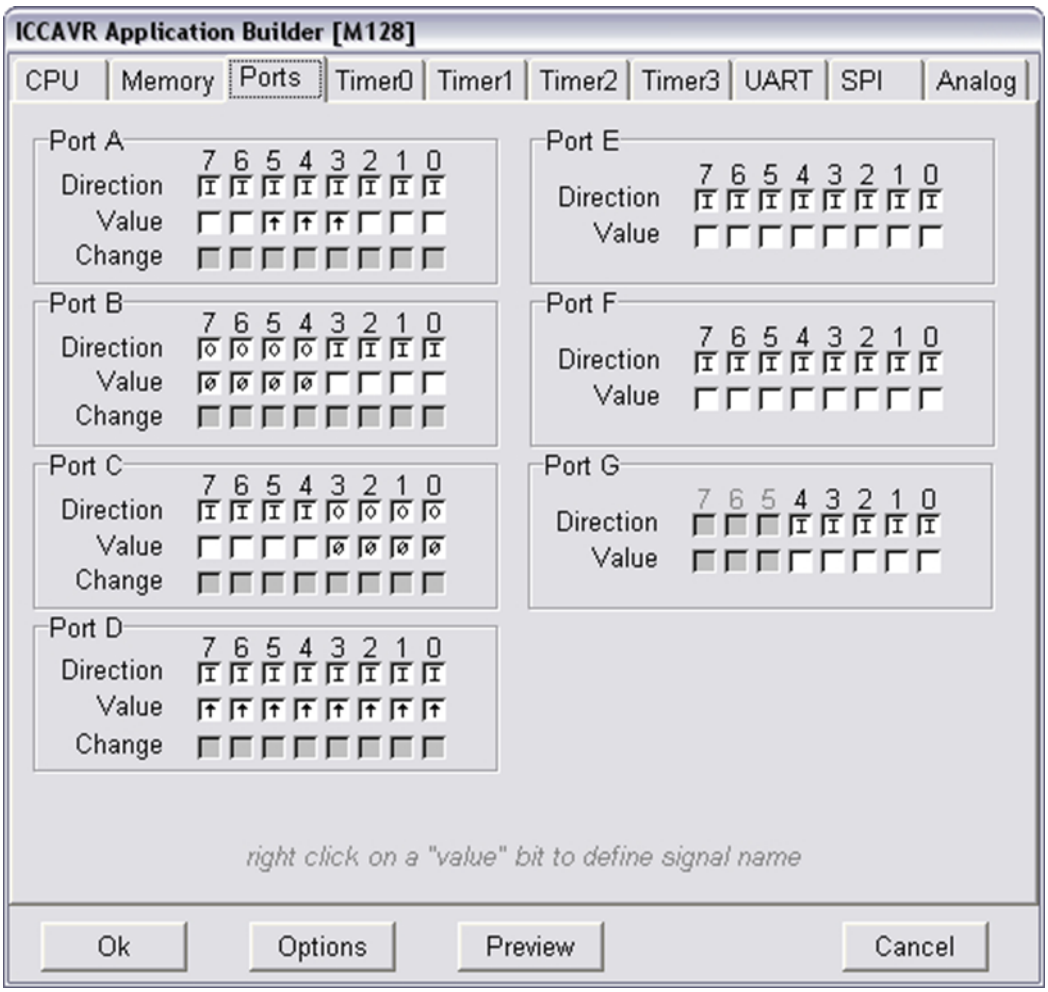


Рисунок 13.3 – Окно Application Builder с опциями настройки портов ввода/вывода

В качестве примера на рисунке 2.3 показана инициализация портов A, B, C и D, а для портов E, F и G состояние регистров направления **DDRx** и данных **PORTx** (x = E, F, G) оставлено без изменений. Видно, что все выводы портов (за исключением 4-х старших у порта B и 4-х младших у порта C) настроены на вход. Кроме того, у выводов 3, 4 и 5 порта A и у всех выводов порта D подключены подтягивающие резисторы. В результате **Application Builder** сгенерирует программный код с функцией void port_init(void), как представлено на рисунке 13.4,

при выполнении которой всем регистрам **DDRx** и **PORTx** будут присвоены значения, соответствующие заданным пользователем установкам (см. рис. 13.3).

```
void port_init(void)
{
  PORTA = 0x38;      // в двоичной системе равно 0b00111000
  DDRA  = 0x00;
  PORTB = 0x00;
  DDRB  = 0xF0;      // в двоичной системе равно 0b11110000
  PORTC = 0x00;
  DDRC  = 0x0F;      // в двоичной системе равно 0b00001111
  PORTD = 0xFF;      // в двоичной системе равно 0b11111111
  DDRD  = 0x00;
  PORTE = 0x00;
  DDRE  = 0x00;
  ……….               // порты F и G инициализированы также, как и порт E
}
```

Рисунок 13.4 – Программный код с функцией void port_init(void)

Подобным образом с помощью **Application Builder** можно настроить работу системы внешних прерываний, таймеров/счетчиков, всех интерфейсов (USART, SPI, TWI), АЦП, компаратора, а также решить ряд других задач, связанных с работой памяти EEPROM, сторожевого таймера и т.д. Если разрешены какие-нибудь прерывания, например, от таймеров или USART, то **Application Builder** генерирует программный код, содержащий шаблоны функций для обработки этих прерываний.

Пункт **Tools> In System Programmer** предназначен для «прошивки» HEX-кода программы в память микроконтроллера с помощью программаторов. После настройки программатора (типа и интерфейса) клавишей Program FLASH/EEPROM инициирует запись кода в память микроконтроллера. Можно также прочитать из памяти микроконтроллера Fuse-биты, определяющие особенности его работы. Можно произвести корректировку Fuse-битов в соответствующем окне, после чего записать скорректированные значения в память микроконтроллера. Следует соблюдать особую осторожность при работе с Fuse-битами, так как запись ошибочного значения в память может создать серьезные проблемы по возвращению микроконтроллера в исходное состояние.

Выполнение наиболее важных пунктов меню можно осуществить с помощью кнопок на панели инструментов или, используя правую кнопку мыши и всплывающее при этом контекстное меню. Это, в частности, относится к вызову таких пунктов, как **Build Project** (компиляция файлов), **Project Options** (установка опций проекта), **Application Builder** (инициализация периферийных устройств), **ICP Dialog** (прошивка кода программы). Если курсор находится в окне редактора, то с помощью правой кнопки мыши можно вызвать функции редактирования текста, поиска меток или добавления новых файлов в проект.

Если курсор находится в окне менеджера проекта, то можно осуществить компиляцию, добавление или удаление файлов из проекта и т.д.

Рассмотрим последовательность действий при создании нового проекта.

Для начала создадим рабочую папку, в которой будут храниться все файлы проекта. Затем выберем пункт меню **Project>New** и в открывшемся окне зададим имя проекта и путь к нему. В окне менеджера появится имя нового проекта и пустые папки для хранения C-файлов, H-файлов и файлов для документов, например, в текстовом формате.

Далее необходимо произвести настройки проекта, установив с помощью **Project>Options>Target** в окне Device Configuration нужный тип микроконтроллера, например, ATMega128/CAN128. С помощью **Project>Options>Paths** установить в окне **Library Path** пути к стандартным библиотечным файлам и в окне **Include Paths** – пути к подключаемым к проекту файлам. Эти файлы находятся в папках **icc\lib** и **icc\include**, которые, в сою очередь, находятся в папке, где установлен компилятор ICCAVR. Если это не первый проект, создаваемый на данном компьютере, то правильные пути к библиотекам уже установлены, надо только это проверить.

Теперь надо подключить файл с исходным кодом, предварительно выбрав пункт **File>New**. Можно написать исходный код непосредственно в окне редактора. В этом случае очень полезна комбинация клавиш Ctrl+j, которая открывает окно с шаблонами операторов. Использование этой комбинации клавиш поможет избежать синтаксических ошибок при разработке программ.

Если текст программы (С-файл) уже есть и его надо лишь подредактировать, то можно открыть этот файл и произвести необходимые исправления. Для начального обучения полезно поработать с исходными С-файлами, расположенными в папке **icc\examples.avr**. Эта папка создается при установке компилятора ICCAVR. Полезно, например, открыть программу **led.c**, которая позволяет переключать состояние одного из выводов микроконтроллера из низкого состояния в высокое и наоборот.

Если программа относительно сложная, то следует воспользоваться **Tools>Application Builder**. Он осуществит начальную инициализацию периферийных устройств и сгенерирует начальный фрагмент программного кода, который пользователь может дорабатывать по своему усмотрению. Необходимо помнить, что после вызова **Application Builder** в пункте CPU следует установить тип микроконтроллера, например, M128 (сокращение от ATMega128) и частоту синхронизации Xtal speed, например, 11.059MHz. Все остальные периферийные устройства настраиваются (или не настраиваются) в соответствии с решаемыми в данной программе задачами. Обычно этот фрагмент кода заканчивается функцией init_devices. Имя этой функции следует вставить в текст главной функции программы main одной из первых (сразу после объявления переменных в функции main), так как именно с инициализации периферийных устройств начинается выполнение программы. Следует помнить, что в программе обязательно должна быть главная функция с именем main, иначе компилятор будет выдавать сообщения об ошибке.

После создания исходного файла, разработанного на языке Си, его необходимо сохранить в рабочей папке, не забыв про расширение к имени файла, так как автоматически расширение файла не формируется. После этого следует добавить файл к проекту, используя, например правую кнопку мыши и выпадающее контекстное меню. В результате в окне менеджера проекта в папке Files появится имя первого файла проекта. К проекту можно добавлять новые С-файлы или Н-файлы, предварительно выбрав папку Headers. Таким образом, создается проект, включающий в себя несколько файлов. Такая модульная структура проекта позволяет лучше воспринимать его алгоритм функционирования. Для контроля ошибок при

разработке программы следует периодически осуществлять компиляцию отдельного файла или всех файлов проекта. Успешная компиляция заканчивается сообщением Done, неуспешная – сообщениями об ошибках и дополнительной информацией о сути этих ошибок и их месте нахождения в программе.

13.2 Примеры программирования периферийных устройств

Пример 1. Установка отдельных бит в регистрах периферийных устройств (ПУ) в состояние лог.1 или лог.0 (рисунок 13.5).

```
PORTB | = (1<<PB4)|(1<<PB0); // выводы 0 и 4 порта B установлены в лог.1
DDRB & = ~(1<<DDB5);         // установка направления вывода PB5 на вход
```

Рисунок 13.5 – Установка отдельных бит в регистрах ПУ (вариант 1)

Первый оператор устанавливает в состояние лог.1 два вывода порта B, оставляя остальные выводы без изменений. Второй оператор обнуляет пятый бит регистра DDR порта B, настраивая вывод PB5 на вход. Остальные выводы остаются при этом без изменений. Альтернативный вариант этих же операций (рисунок 13.6).

```
PORTB | = 0b00010001; // выводы 0 и 4 порта B установлены в лог.1
DDRB & = 0b11011111;  // установка направления вывода PB5 на вход
```

Рисунок 13.6 – Установка отдельных бит в регистрах ПУ (вариант 2)

Пример 2. Ожидание нажатия кнопки, в результате чего на выводе 1 порта B устанавливается низкое напряжение (бит PINB1 в регистре PINB становится равным 0) и далее программа продолжает выполнять следующие операторы (рисунок 13.7).

```
while (PINB&0b00000010) // ожидание нажатия кнопки: 0 - нажата: 1 - не нажата
{;}                     // пока кнопка не нажата, следующие операторы не выполняются
```

Рисунок 13.7 – Ожидание нажатия кнопки

Данный фрагмент программы обычно используется, кода требуется инициировать какое-то действие путем нажатия кнопки. В ожидании нажатия кнопки программа с помощью оператора **while** «зациклена» в одном месте.

Следующие за ним операторы будут выполняться лишь тогда, когда кнопка будет нажата и 1-й бит регистра PINB установится в лог.0.

Пример 3. Передача данных через универсальный асинхронный приемо-передатчик USART0 с использованием опроса флага освобождения буфера передатчика **UDRE** и предварительной инициализацией USART0 (рисунок 13.8).

```
void uart0_init(void) // инициализации USART0 (генерируется Application Builder)
{
  UCSR0A = 0x00;
  UCSR0C = 0x06; // установка формата данных в посылке: 8 бит, 1 стоп-бит
  UBRR0L = 0x0B; // установка скорости передачи данных: 57600 бод
  UBRR0H = 0x00;
  UCSR0B = 0x08; // разрешение работы передатчика
}
void USART_Transmit( unsigned char data ) // функция передачи данных через USART
{
  while ( !( UCSRA & (1<<UDRE)) ); // ожидание освобождения буфера передатчика
  UDR = data;                      // помещение данных в буфер и их пересылка
}
```

Рисунок 13.8 – Передача данных через универсальный асинхронный приемо-передатчик

Программа будет ожидать, пока первый бит в регистре UCSRA, т.е. бит UDRE, сигнализирующий об освобождении буфера передатчика, станет равным лог.1. После этого будет выполнен следующий за while оператор – в буфер передатчика UDR будет записано некоторое числовое значение, которое затем автоматически будет пересылаться через вывод микроконтроллера TXD0 получателю этой информации, например, в компьютер.

Пример 4. Чтение результата преобразования внутреннего 10-разряжного АЦП. Предварительно в функции инициализации adc_init() произведены все необходимые настройки АЦП, включая установку источника опорного напряжения (ИОН), предделителя частоты и т.д. Операцию чтения АЦП выполняет функция ReadADC(), которой передается результат преобразования (рисунок 13.9).

```
void adc_init(void) //инициализация АЦП (генерируется Aplacation Builder)
{
  ADCSRA = 0x00;       // работа АЦП пока запрещена
  ADMUX = 0b11000000; // подключен внутренний ИОН на 2,56 В
  ACSR = 0x80;         // отключен аналоговый компаратор
  ADCSRA = 0xC6;       // установлены биты ADEN, ADSC (разрешение и старт),
                       //а также ADPS2 и ADPS1 (предделитель на 64)
}
int ReadADC(void) //функция чтения внутреннего АЦП
{
  int data_ADC;                // объявление переменной data_ADC
  ADMUX |= (1<<MUX0);          // выбор канала мультиплексора (канал №1).
  ADCSRA |= (1<<ADSC);         // старт преобразованию установкой бита ADSC в лог. 1
  while (ADCSRA & (1<<ADSC)); // ожидание, пока закончится преобразование и
                               // бит ADSC в регистре ADCSRA обнулится
  data_ADC = ADCL;             // чтение младшего байта результата преобразования
  data_ADC += (int)ADCH << 8; // прибавление к результату старшего байта
  return data_ADC;             // передача результата на выход функции
}
```

Рисунок 13.9 – Чтение результата преобразования внутреннего 10-разряжного АЦП

Если теперь в основной программе, содержащей функцию main, встретится оператор вида

X = ReadADC();

то будет инициирована работа внутреннего АЦП, а результат аналого-цифрового преобразования будет присвоен переменной X. Разумеется, эта переменная должна быть объявлена ранее до вызова функции ReadADC(), причем переменная X должна иметь тип **int**.

Пример 5. Чтение результата преобразования внешнего 16-разрядного АЦП с последовательным выходом, например, AD7683. АЦП взаимодействует с ATmega128 посредством SPI-интерфейса. Особенностью работы АЦП данного типа является то, что до начала выполнения аналого-цифрового преобразования необходимо на тактовый вход АЦП в «ручном» режиме подавать чередующиеся лог.0 и лог.1 и ждать, когда на выходе АЦП появится лог.0. После этого инициируется работа SPI-интерфейса и с большой скоростью происходит чтение результата преобразования. АЦП активен, т.е способен выполнять свои функции под управлением микроконтроллера, если у него на выводе CS (выбор кристалла) установлен лог.0. Это позволяет микроконтроллеру из нескольких устройств, с

которыми он взаимодействует посредством SPI-интерфейса, выбрать тот, который нужен в данный момент (рисунок 13.10).

```
int ReadADC_ext(void) //функция чтения внешнего АЦП с помощью SPI-интерфейса
{
 unsigned int tmp=0; // объявление переменной целого типа tmp, ей будет передан
                     // результат преобразования
 SPCR=(1<<MSTR);     // выбор режима работы "Master", первым передается старший
                     // значащий разряд, скорость передачи fclk/4
 PORTE&=~(1<<PE7);   // сигнал PE7 (сигнал CS у АЦП) - в лог.0
 while (PINB&(1<<PB3))
   {                 // подготовка АЦП к работе, что включает в себя:
     PORTB|=(1<<PB1);   // формирование чередующихся лог.1 и лог.0, ожидание
     PORTB&=~(1<<PB1);  // появления лог.0 на выводе Dout у АЦП
   }
 PORTB|=(1<<PB1);            // установка вывода в лог.1
 SPCR|=(1<<SPE);             // разрешение работы модуля SPI при частоте fclk/4:
 SPSR|=(1<<SPI2X);           // включение бита SPI2X – бита удвоения скорости обмена
 SPDR=123;                   // запись в регистр данных SPDR произвольного числа для
                             // инициализации работы SPI (это особенность работы АЦП)
 while(!(SPSR & (1<<SPIF))); // ожидание конца передачи ст. байта и установки флага SPIF
 tmp=SPDR<<8;                // запись ст. байта рез-тата преобразования в переменную tmp
 SPDR=123;                   // запись в регистр данных SPDR произвольного числа
 while(!(SPSR & (1<<SPIF))); // ожидание конца передачи мл. байта и установки флага SPIF
 tmp|=SPDR;                  // дополнение результата младшим байтом
 SPCR&=~(1<<SPE);            // запрещение работы модуля SPI
 PORTE|=(1<<PE7);            // отключение сигнала CS - выбор микросхемы АЦП
 return tmp;                 // результат выполнения функции передан переменной tmp
}
```

Рисунок 13.10 – Чтение результата преобразования внешнего 16-разрядного АЦП с последовательным выходом

Список использованных источников

1 Рудаков, П.И. Язык ассемблера: уроки программирования. / П.И. Рудаков, К.Г.Финогенов. – М.: Диалог- МИФИ, 2001.- 640 с.

2 Абель, П. Язык Ассемблера для и программирования. / Перевод с английского. - М.: Высшая школа, 1992.-477с.

3 Юров, В.И. Assembler. Учебник для вузов. 2-е изд. / В. И. Юров – СПб.:Питер, 2003. – 637 с.

4 Язык ассемблера для IBM PC. Перевод с английского./П. Нортон, Д. Соухе. – М: Издательство «Компьютер», 1993. – 252 с.

5 Калашников, О.А. Ассемблер - это просто. Учимся программировать. 2-е издание, переработанное и дополненное. Для программистов. / О. А Калашников. – Санкт-Петербург: «БХВ-Петербург», 2011.– 330с.

6 Аблязов, Р. З. Программирование на ассемблере на платформе x86-64. / Р.З. Аблязов. – ДМК Пресс, 2016. –304 с.

7 Скенлон, Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера./Л.Скэнлон Л. – М: Радио и связь, 1991. – 336 с.

8 Ревич, Ю. В. Практическое программирование микроконтроллеров Atmel AVR на языке ассемблера (3-е издание) / Ю.В. Ревич. – БХВ-Петербург, 2015, –354 с.

9 Гуров, В. В. Микропроцессорные системы: Учебник / В.В. Гуров. - М.: НИЦ ИНФРА-М, 2016. - 336 с.: 60x90 1/16. - (Высшее образование: Бакалавриат) ISBN 978-5-16-009950-7. Режим доступа: http://znanium.com/bookread2.php?book=462986

10 Гуров, В.В. Архитектура микропроцессоров / В.В. Гуров. М.: Интернет-Университет Информационных Технологий, 2010. Режим доступа: http://biblioclub.ru/index.php?page=book&id=233074&sr=1

Приложение А

(обязательное)

Основные команды языка Ассемблера

Таблица А1 – Основные команды языка Ассемблера

		Флажки			
Команда	Действие	Z	S	C	O
MOV Оп1, Оп2	Оп1 ←Оп2	-	-	-	-
XCHG Оп1, Оп2	Оп1↔ Оп2	-	-	-	-
NEG Оп1	Оп1:= - Оп1				
ADD Оп1, Оп2	Оп1:= Оп1+Оп2	*	*	*	*
ADC Оп1, Оп2	Оп1:= Оп1+Оп2+Флаг С	*	*	*	*
SUB Оп1, Оп2	Оп1:= Оп1- Оп2	*	*	*	*
SBB Оп1, Оп2	Оп1:= Оп1- Оп2- Флаг С	*	*	*	*
INC Оп	Оп1:= Оп1+1	*	*	-	*
DEC Оп	Оп1:= Оп1-1	*	*	-	*
MUL Оп	DX:AX = AX*Оп	*	-	*	*
IMUL Оп	AX = AL*Оп	*	*	*	*
DIV Оп IDIV Оп	AX = DX:AX div Оп; DX = DX:AX mod Оп	*	-	*	*
	AL = AX div Оп; AH = AX mod Оп	*	*	*	*
CMP Оп1, Оп2	Оп1- Оп2 без сохранения результата	*	*	*	*
NOT Оп1	Оп1:= инверсия Оп1				
AND Оп1, Оп2	Оп1:= Оп1 &Оп2	*	*	0	0
OR Оп1, Оп2	Оп1:= Оп1 ∨Оп2	*	*	0	0
XOR Оп1, Оп2	Оп1:= Оп1 ⊕ Оп2	*	*	0	0
TEST Оп1, Оп2	Оп1 & Оп2 без сохранения результата	*	*	0	0
SHL Оп1,1 SHL Оп1,CL	Логический сдвиг влево на 1 или CL бит	*	*	*	*
SHR Оп1,1 SHR Оп1,CL	Логический сдвиг вправо на 1 или CL бит	*	*	*	*

Продолжение таблицы А1

Команда	Действие	Z	S	C	O
SAR Оп1,1 SAR Оп1,CL	Арифметический сдвиг вправо на 1 или CL бит	*	*	*	0
JMP метка	Переход на метку	-	-	-	-
JC метка JB метка	Переход, если C=1 (если Оп1 в CMP<Оп2 и Оп1 и Оп2 - беззнаковые)	-	-	-	-
JNC метка JNB метка JAE метка	Переход, если C=0 (если Оп1 в CMP>=Оп2 и Оп1 и Оп2 - беззнаковые)	-	-	-	-
JZ метка	Переход, если Z=1 (если Оп1 в CMP=Оп2 или если результат нулевой)	-	-	-	-
JNZ метка	Переход, если Z=0 (если Оп1 в CMP<>Оп2 или если результат ненулевой)	-	-	-	-
JS метка	Переход, если S=1 (если результат отрицательный)	-	-	-	-
JNS метка	Переход, если S=0 (если результат положительный)	-	-	-	-
JBE метка JNA метка	Переход, если C∨Z=1 (если Оп1 в CMP<=Оп2 и Оп1, Оп2 – беззнаковые)	-	-	-	-
JA метка	Переход, если C=0 и Z=0 (если Оп1>Оп2 и Оп1, Оп2 – беззнаковые)	-	-	-	-
JL метка	Переход, если O⊕S = 1 (если Оп1 в CMP<Оп2 и Оп1 и Оп2 – знаковые)	-	-	-	-
JNL метка JGE метка	Переход, если O⊕S = 0 (если Оп1 в CMP>=Оп2 и Оп1 и Оп2 – знаковые)	-	-	-	-

Продолжение таблицы А1

Команда	Действие	Z	S	C	O
JLE метка JNG метка	Переход, если O⊕S = 1 и C=1 (если Оп1 в CMP<=Оп2 и Оп1 и Оп2 – знаковые)	-	-	-	-
JG метка	Переход, если O⊕S = 0 и Z=0 (если Оп1 в CMP>Оп2 и Оп1 и Оп2 – знаковые)	-	-	-	-
JO метка	Переход, если флаг O=1	-	-	-	-
JNO метка	Переход, если флаг O=0	-	-	-	-
LOOP метка	CX := CX-1, переход, если CX<>0	-	-	-	-
CALL метка	Вызов подпрограммы	-	-	-	-
RET	Возврат из подпрограммы	-	-	-	-
PUSH регистр	Запись регистра в стек	-	-	-	-
POP регистр	Восстановление из стека	-	-	-	-
PUSHF	Запись регистра FLAGS в стек	-	-	-	-
POPF	Восстановление FLAGS из стека	-	-	-	-
CLD	Установка флага D в единицу	-	-	-	-
STD	Сброс флага D в нуль	-	-	-	-
CBW	Заполняет AX знаковым битом AL				
CWD	Заполняет DX знаковым битом AX				

Приложение Б

(обязательное)

Пример простой резидентной программы

```
;   Пример простой резидентной программы ( tsrpos.asm )– > tsrpos.com
;   По F1 –  выводиться строка
;   По Ctrl+F1 –  программа выгружается
;   Выполняется проверка повторности запуска TSR
;   Выполняется вызов справки по /H или /h
;   Выполняется выгрузка из части инициализации /U или /u
CODEPR SEGMENT PARA
 ASSUME CS:CODEPR , DS:CODEPR
 PSP:    ORG 100H           ; Область PSP
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Резидентная часть
BEGIN:  JMP  INIT    ; Переход к части инициализации
;    Данные резидента
SAVEINT9 DD ?         ; Сохранение старого обработчика
SAVEINT2D DD ?         ; Сохранение старого обработчика
FLAG DB  0
;    Новый обработчик прерывания 09H
NEWINT9:
    CLI
    PUSH  AX       ; Сохранение используемых регистров
 ; Вызов старого обработчика
    PUSH ES
    PUSH DS
    PUSH CS
    POP  DS
;
    PUSH AX
    PUSH BX
    PUSH CX
    PUSHF
   CALL CS:SAVEINT9   ; Вызов старого обработчика    с возвратом
 ;     Запись в AX и AL кодов из буфера
;    MOV AX , 40H
;    MOV ES , AX
;     MOV BX , ES:1AH
;     MOV AX, ES:[BX]
    MOV AH, 1
    INT  16H
    JZ PROD
;     Проверка расширенного кода
    CMP AL , 0
    JE  EXT
     CMP AH, 1DH
; Для чистого CTRL
     JNE PROD
     MOV AH,00H
    INT 16H
```

```
PROD:
    POP CX
    POP BX
    POP AX
    JMP  END9
;   Проверка нажатия F1
EXT:  CMP AH , 3BH ; Код F1
     JE PRINT1          ; Печать строки   по F1
; Проверка выгрузки
     CMP AH , 5EH    ; Нажата Ctrl + F1
     JNE PROD
UNLD:
; Выгрузка
     PUSH ES
     MOV DX,OFFSET REZMSG1   ; Вывод сообщение о завершении
    CALL PRINT       ; резидента
    POP ES
;
    PUSH DX
    PUSH ES
 ;
; 9H
   mov AX,2509H  ; Восстановление обработчика прерывания 05H
   lds DX,CS:SAVEINT9
   int 21H
; 2FH
   mov AX,252DH  ; Восстановление обработчика прерывания 2FH
   lds DX,CS:SAVEINT2D
   int 21H
; получим из PSP адрес собственного  окружения и выгрузим его
   MOV   ES,CS:2CH
   MOV   AH,49H
   INT   21H
; выгрузим теперь саму программу
   PUSH  CS
   POP       ES
   MOV   AH,49H
   INT   21H
; Код в AL для успешной выгрузки
    MOV AH,00H
    INT 16H
;
    POP ES
   POP DX
    POP CX
    POP BX
    POP AX
;
    POP DS
    POP ES
    POP AX
    STI
    IRET
 ;  Вывод цепочки F1
```

```
PRINT1:
     PUSH ES
     MOV DX,OFFSET REZMSG2   ; Вывод сообщение о F1
    CALL PRINT       ;
    POP ES
 ; Очистка буфера клавиатуры
    PUSH ES
   MOV AX , 40H
   MOV ES , AX
    MOV BX , ES:1AH
    MOV ES:1CH , BX
;
     POP ES
    JMP PROD
 ;
END9:
     POP DS
     POP ES
     POP AX
     STI
    IRET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
NEWINT2D PROC FAR
         CLI
         PUSH DS
;
        PUSH CS
         POP  DS
       CMP AH, 0EEh  ; собственный код
       JE MY_OBR
       JMP OLD_OBR
; Новый обработчик
MY_OBR:  CMP AL , 1 ; Проверка наличия
        JE EXIT_2F
          CMP AL, 2  ; Выгрузка
          JE UNLOAD
        ;;;;;;;;;
EXIT_2F: MOV AL, 0FFH ; Есть в памяти
         STI
         POP DS
         IRET
 ;
 UNLOAD:
 ; Восстановление обработчиков
     PUSH CS
     POP  DS


     MOV DX,OFFSET REZMSG1   ; Вывод сообщение о завершении
;    CALL PRINT       ; резидента

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    PUSH DX
    PUSH ES
```

```
; 9H
   mov AX,2509H  ; Восстановление обработчика прерывания 05H
   lds DX,CS:SAVEINT9
   int 21H
; 2FH
   mov AX,252DH  ; Восстановление обработчика прерывания 2FH
   lds DX,CS:SAVEINT2D
   int 21H
 ; получим из PSP адрес собственного окружения и выгрузим его
   MOV   ES,CS:2CH
   MOV   AH,49H
   INT   21H
; выгрузим теперь саму программу
   PUSH  CS
   POP       ES
   MOV   AH,49H
   INT   21H
; Код в AL для успешной выгрузки
   MOV AL, 0FFH
   POP DX
   STI
   POP ES
   POP DS
    IRET
OLD_OBR: MOV AL, 0  ; наш резидент не установлен
     STI
     POP DS
   JMP   CS:SAVEINT2D   ; Вызов старого обработчика  2FH
NEWINT2D ENDP
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Процедура печати строки для резидентной программы
; т.к. 21H(09H) нельзя
; DX –  adress string NEAR (DS –  БАЗА)
;  Конец строки –  '$'
;
PRINT  PROC
     PUSH DI
     PUSH AX
     PUSH CX
     PUSH DX
     PUSH BX
    MOV   DI,DX
;  Позиция курсора в начале
    MOV   AH,03H              MOV   BH,00H
    INT   10H
;
    MOV CX , 70
; цикл вывода и проверки символов строки
LOOP1:  MOV   AL,DS:[DI]
    CMP   AL,'$'   ; Конец строки для вывода
    JE   FIN
;   Печать символа на экран
     INC DI
    MOV   BH,0
```

```
    MOV   AH,0AH
    PUSH CX
    MOV   CX,0001H
    INT   10H
    POP CX
; Новая позиция курсора
NEXT4:  MOV   AH,02H
     INC  DL
    MOV   BH,00H
    INT   10H
    JMP   LOOP1
; Конец цикла вывода
FIN:
     POP BX
     POP DX
     POP CX
     POP AX
     POP DI   ; Восстановление стека
    RET
PRINT  ENDP
REZMSG1 db 'TSR was unloaded (Ctrl+F1 –  TSR part)!','$'
REZMSG2 db 'TSR: F1 –  pushed!!!', '$'
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Часть инициализации
INIT:  CLI       ; Запрет прерываний
; Установка DS
    PUSH CS
    POP DS
 ; проверка параметров
    MOV AL ,  CS:80H
    CMP AL, 0
    JNE INIT_UND       ; Есть параметры
; Проверка в мамяти
INIT1:     PUSH CS
         POP DS
        MOV AH, 0EEh
       MOV AL, 1  ; Проверка наличия
       INT 2Dh
       CMP AL, 0FFh
       JNE INIT2  ; переход если резидент не(!) в памяти (на загрузку)
; Сообщение о том, что уже загружен
    MOV AH , 09H
    MOV DX, OFFSET MSG_INMEM
    INT  21H
 ; Выход из программы  без установки резидента
EXITTSR:
   MOV AL, 0
   MOV AH, 4CH
   STI
   INT 21H   ; Вызрузка  или справка
 INIT_UND:
 ; Есть параметры
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;   разборка параметров
   MOV SI , OFFSET ( PSP + 81H)
```

```
    LODSB      ; первый пробел
    LODSB
    CMP  AL , '/'
    JNE   ERR_PAR
    LODSB       ; символ параметра
    CMP  AL , 'H'
    JE  INIT_HELP
    CMP  AL , 'h'
    JE  INIT_HELP
    CMP  AL , 'U'
    JE  INIT_UND1
    CMP  AL , 'u'
    JE  INIT_UND1
    JMP ERR_PAR
 ; Выгрузка    резидента
 INIT_UND1:
    MOV AH , 0EEH
    MOV AL,2
    INT 2Dh
    CMP AL , 0FFH
    JE INIT3
 ;   Сообщение о том что нет выгрузки
    MOV AH , 09H
     MOV DX, OFFSET MSG_NOUNLD
     INT  21H
     JMP  EXITTSR
 ; Сообщение о выгрузке
 INIT3:
    MOV AH , 09H
     MOV DX, OFFSET MSG_UNLD
     INT  21H
     JMP  EXITTSR
; Справка
INIT_HELP:
     MOV AH , 09H
     MOV DX, OFFSET MSG_Help
     INT  21H
     JMP  EXITTSR
ERR_PAR:
     MOV AH , 09H
     MOV DX, OFFSET MSG_Param
     INT  21H
     JMP  EXITTSR
; Получение адреса старого обработчика    9
INIT2: MOV   AH, 35H
     MOV  AL, 09H  ; Номер прерывания
     INT 21H
; Сохранение адреса старого обработчика
     MOV   WORD PTR SAVEINT9 , BX
     MOV   WORD PTR SAVEINT9+2 , ES
; Установка нового обработчика в вектор прерывания      9
     MOV   AH,25H
     MOV   AL, 09H  ; Номер прерывания
     MOV   DX, OFFSET  NEWINT9
```

```
    INT   21H
; Получение адреса старого обработчика    2F
    MOV  AH, 35H
    MOV  AL, 2DH  ; Номер прерывания
    INT 21H
; Сохранение адреса старого обработчика
    MOV   WORD PTR SAVEINT2D , BX
    MOV   WORD PTR SAVEINT2D +2 , ES
; Установка нового обработчика в вектор прерывания      2F
    MOV   AH,25H
    MOV   AL, 2DH  ; Номер прерывания
    MOV   DX, OFFSET  NEWINT2D
    INT   21H
; Вывод сообщения о загрузке резидента
    MOV AH , 09H
    MOV DX, OFFSET MSG
    INT  21H
; Завершить и оставить резидентной (TSR)
    MOV   AX, 3100H
    MOV   DX, (init –  begin +10FH)/16  ; Размер резидента
    STI        ; Разрешение прерываний
    INT    21H
; Данные части инициализации
MSG   DB 'Start TSR program!', 10,13,'$'
MSG_INMEM DB 'TSR already is in memory!', 10,13,'$'
MSG_UNLD DB 'TSR was unloaded (Init part)!', 10,13,'$'
MSG_NOUNLD   DB 'TSR is NO in memory!', 10,13,'$'
MSG_Help   DB 'Test –  sample TSR –  2010', 10,13,'Parameters: /H or /h –
help , /U or /u –  unload TSR ', 10,13,'€– д®a¬ жЁпя!',10,13,'$'     ; По–
следний текст "Информация" –  введен в коде ASCII!!!
MSG_Param   DB 'Error –  command line parameters!', 10,13,'$'
CODEPR  ENDS
END   BEGIN
```