The ultimate aim of this chapter is to demonstrate how to have an assembly language program converted to machine code and executed by the host 8086-family based microcomputer. The process will be illustrated by means of a specific example program, namely one which turns the computer into a very simple adding machine. But there is much to describe before even this is intelligible, so actual program execution is very much the climax of the chapter.

Before that, because we want our very simple adding machine to be able to display a message on the screen and return to DOS control when it has finished, we examine some of the DOS operating system functions, a set of ready-made routines which carry out jobs such as displaying characters on the screen automatically – provided we bring them into action in the appropriate way. One of these will even display a complete message at one go, so long as the message has first been stored in memory. Consequently, the chapter begins by explaining how storage is assigned to fixed items of data (such as a message which is to be displayed repeatedly during the execution of a program) in an assembly language program.
4.1 Allocating storage space

The allocation of storage in an assembly language program is carried out by use of **pseudo-ops**. Pseudo-ops give extra information to the assembler, beyond the raw program instructions for the task to be carried out, which it uses to convert our assembly language into machine code. The main pseudo-op used to allocate storage is the **define** pseudo-op which takes five basic forms:

- **DB**: Define Byte
- **DW**: Define Word
- **DD**: Define Doubleword
- **DQ**: Define Quadword
- **DT**: Define Tenbytes

These allow, respectively, multiples of one, two, four, eight and ten bytes of storage to be allocated and initialized at one time. Thus:

- **DB 16H** ;allocates one byte of store with initial value 16H
- **DB ?** ;allocates one byte of store with indeterminate ;initial value
- **DB 'Z'** ;allocates one byte of store with initial value ;the ASCII code for the letter Z
- **DW 1992D** ;allocates two bytes of storage, 1992D ;being automatically converted into the ;signed 16-bit form (07C8H) and the two bytes ;initialized to that value consistent with the ;low-order byte in the lowest memory address ;principle described in Chapter 3:

  - address  n    n+1
  - contents C8    07

- **DD 1294967295D** ;allocates four bytes of storage, 1294967295D ;being automatically converted into signed ;32-bit form and the four bytes ;initialized to that value (OFFA12F4DH) ;consistent with the low-order ;byte in the lowest memory address ;principle described in Chapter 3:

  - address  n    n+1  n+2  n+3
  - contents 4D    2F    A1    FF

If several define pseudo-ops are used one after another, then consecutive memory locations are allocated for the objects defined. Thus, given the sequence:

- **DB 14H**
- **DW 1000D**
if the first value (14H) is stored in location 200H relative to DS (say) then the contents of the ten locations 200H, 201H, to 209H will be as follows (all values are hexadecimal):

<table>
<thead>
<tr>
<th>Location</th>
<th>200 201 202 203 204 205 206 207 208 209</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>14 E8 03 48 45 4C 4C 4F 0A 00</td>
</tr>
</tbody>
</table>

since 1000D = 03E8H, 10D = 000AH, and the ASCII codes for the letters in HELLO are 48H, 45H, 4CH and 4FH respectively.

Abbreviated multiple definitions are also permissible. Thus:

```
```

allocates seven bytes, with initial values determined just as if seven separate DB pseudo-ops had been used one after the other:

```
DB 21H
DB 45H
DB 73H
DB 'A'
DB 'B'
DB 11H
DB 'Z'
```

Similarly,

```
DB 'HELLO'
```

allocates five bytes of storage in exactly the same way as:

```
DB 'H'
DB 'E'
DB 'L'
DB 'L'
DB 'O'
```

and

```
DW 1,10,100,1000,10000
```

allocates ten bytes of storage, initialized to the signed 16-bit forms of 1D, 10D, 100D, 1000D and 10000D.

Note that:

```
DB 'ALL GOOD MEN'
```

is equivalent to:

```
DB 'A', 'L', 'L', 'L', 'G', ...
```

so that the fourth and ninth bytes allocated will be initialized to the ASCII code for a space, namely 20H.
The `DUP( . . .)` pseudo-op permits multiple initializations to the same value. Thus we have:

DB 10H DUP('A') ;allocates the next 10H locations with initial value 41H (the ASCII code for the letter A)

DB 80H DUP(?) ;allocates the next 80H locations with initial value indeterminate

DW 50H DUP(100D) ;the next 50H words are set to 0064H (the signed 16-bit form of 100D)

### 4.1.1 Common confusions about storage allocations

There are three very common pitfalls in understanding how storage allocation is made in 8086-family assembly language. Consider the program fragment:

```assembly
DATA SEGMENT
  MY_BYTE DB 32H
  MY_DIGITS DB '3','2'
  ...

The first thing to emphasize is that in associating offsets with the corresponding memory locations for our data, MASM starts at zero. Thus, the offset of `MY_BYTE` from the start of the DATA segment will be zero. Moreover,

```
MY_BYTE DB 32H
```

reserves just a *single* byte of store (containing 00110010B = 32H) and *not* two. Lastly,

```
MY_DIGITS DB '3','2'
```

reserves *two* bytes of store containing 00110011B = 33H and 00110010B = 32H respectively—the ASCII codes for the digits 3 and 2. To summarize, as a result of the above DATA segment pseudo-ops, storage will be allocated as shown in Figure 4.1.

![Figure 4.1](image_url)

**Figure 4.1**
Allocation of storage as a result of certain DATA segment pseudo-ops.

<table>
<thead>
<tr>
<th>Start of DATA segment</th>
<th>Offset 0</th>
<th>Offset 1</th>
<th>Offset 2</th>
<th>Offset 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110010</td>
<td>00110011</td>
<td>00110010</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

One byte One byte One byte

---

### EXERCISES

#### 4.1 How many bytes in total are defined by the following groups of define pseudo-ops:
4.2 If the first location which is to be used to store the data items defined in
the following sequence of define pseudo-ops has offset 200H relative to
DS, complete the table:

<table>
<thead>
<tr>
<th>Location</th>
<th>200</th>
<th>201 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>41</td>
<td>41 ...</td>
</tr>
</tbody>
</table>

showing the outcome of these definitions.

```
DB 'AA.BBB,CCC','h','e','l','l','o',24H
DW 1024,2048,65535
DW 5 DUP(10660)
```

4.3 Give suitable define pseudo-ops to initialize a group of locations to
specify each of the months in the year in calendar order by means of:
(a) the name of the month (using the same fixed number of bytes for
each such name); and
(b) an unsigned 8-bit number representing the number of days in the
month.

---

4.2 Accessing data items in assembly language programs

4.2.1 Specifying addresses

In a large payroll program written in Pascal it is better to declare the rates of
tax as constants at the beginning of the program and then refer to them via the
corresponding variable names rather than to use the actual values. For then, if
the tax rates change (as they inevitably will!), updating the program would
simply involve changes to the dozen or so constant declarations rather than
searching through the entire 30,000 lines of Pascal for every occurrence of a tax
rate (and quite possibly missing several of them).

Similarly, having written a large assembly language program which
makes some complicated calculations based on the contents of a certain group
of memory locations, if we avoid using actual numeric addresses in the
program itself, changing the program to work on a different group of locations
will be much easier. Consequently, 8086-family assembly language does not
allow addresses of the form [200] as in the perfectly valid instruction MOV
AX,[200] – all addresses must be represented by variables which are assigned to the locations to which they refer in the appropriate segment of the program.

To see how this is done in practice, suppose we wish to store two very simple messages in memory ready to be output to the display screen. If the messages are Y and N (for Yes and No) then we allocate storage using DB as above:

```
DB 'Y'
DB 'N'
```

In this case, it does not matter which actual memory locations are used to store the message since it makes no difference to the outcome we are after (provided, of course, that the message is not stored so that it interferes with either DOS or our program). Since it is quite sufficient to have a name by which to refer to them we assign a variable name to each DB statement:

```
AFFIRMMESSAGE DB 'Y'
DENYMESSAGE DB 'N'
```

Notice the difference between a label (which ends with a colon) and a variable (which does not). Otherwise names for variables and labels are constructed in exactly the same way.

This variable name may subsequently be used (rather than an actual address inside [ and ]) to refer to the location in which each message is stored. Thus:

```
MOV AL,AFFIRMMESSAGE
```

will load AL with the ASCII code for the letter Y. Similarly, to overwrite the Y which stands for Yes with a J standing for the German ‘Ja’ we could use:

```
MOV AL,4AH ; the ASCII code for the letter J
MOV AFFIRMMESSAGE,AL
```

This is a luxury afforded us by assembly language. During conversion to machine code, the actual address corresponding to AFFIRMMESSAGE will be worked out and, if we suppose this to be 12AB:9876, then

```
MOV AL,AFFIRMMESSAGE
```

will be converted into the machine code equivalent of

```
MOV AL,[9876]
```

automatically.

The same idea is used for a longer message such as HAPPY NEW YEAR so that:

```
DEC.31MESSAGE DB 'HAPPY NEW YEAR'
```

will associate the variable name DEC.31MESSAGE with the beginning of the message, that is to say, the location in memory in which the ASCII code for the letter H is stored.
4.2.2 Linking data items with the DS register

The definitions made using define statements will most often be given in the data segment definition in our assembly language program. Following our conventional layout, the data segment of an assembly language program containing our definitions of AFFIRMMESSAGE and DENYMESSAGE, the message 'Good morning readers' and the signed 16-bit forms of 1066 and 1984 would be written:

```
DATA SEGMENT
AFFIRMMESSAGE DB 'Y'
DENYMESSAGE DB 'N'
     DB 'Good morning readers'
     DW 1066
     DW 1984
DATA ENDS
```

4.2.3 Establishing data segment addressability

If a program uses items of data stored in a data segment, the DS register must be set to the value DOS chooses for that data segment before execution can commence. The start of the data segment of a program which follows our naming conventions is easily accessed since the segment name DATA has the relevant address as one of its attributes. We need simply make sure that the sequence:

```
MOV AX, DATA
MOV DS, AX
```

is executed before any data is transferred to or from the data segment defined in DATA.

4.2.4 OFFSET

There are occasions when we don't need to access actual addresses but we do need to know such things as the difference between two addresses. For example, this might be the case if we were trying to count the number of locations containing some given letter of the alphabet by subtracting the address of the first location containing the letter from the address of the last location which contains it.

To make this kind of thing possible, associated with each variable are two values: the address of the segment to which the location it denotes belongs, and the offset within that segment. Consequently the assembler can be told to use one of these two numbers in arriving at the machine code form of an instruction. To specify that the offset is what is required we use the pseudo-op OFFSET.

To be explicit about what happens, let us suppose that AFFIRMMESSAGE in fact corresponds to the actual address 102A:0200 and DENYMESSAGE to the actual address 102A:0201. Then the assembly language statements:
MOV BX,OFFSET AFFIRM\_MESSAGE
MOV CX,OFFSET DENY\_MESSAGE

would load BX with 200H and CX with 201H.

Having now seen how to allocate storage to a message, the next section will show how DOS provides facilities to have such a message printed on the display screen with very little further work on our part.

---

**EXERCISE**

4.4 Given:

```
START\_DATA DB 2,3,5,7,11,13,17,19
     DB 1,2,4,8,16,32,64,128
TEST\_DATA DB 11,21,31,41,51,61,71,81
RESULTS DB 8 DUP(?)
```

and that the location to which the variable START\_DATA refers has offset 50H relative to DS, make a trace of the execution of the following program fragment showing clearly the values of all registers and memory locations involved:

```
MOV DX,OFFSET RESULTS
MOV CX,OFFSET TEST\_DATA
MOV AH,8
MOV BX,0
NEXT: ADD BX,OFFSET START\_DATA
      MOV AL,[BX]
      SUB BX,OFFSET START\_DATA
      ADD BX,CX
      ADD AL,[BX]
      SUB BX,CX
      ADD BX,DX
      MOV [BX],AL
      SUB BX,DX
      INC BX
      DEC AH
      JNZ NEXT
```

---

**4.3 DOS functions**

DOS provides a whole range of **functions** which can be brought into action by assembly language programs. These functions work by allowing the programmer to interrupt the normal, line-by-line execution of an assembly language program. It is then possible to branch into the operating system's own machine code instructions. Within the operating system are routines to read a
character from the keyboard, print a character on the screen, read some data from a disk, ... After the execution of one of these routines, control is returned to the program which caused the interruption. Execution then resumes on the usual line-by-line basis.

The 8086-family instruction which allows this to happen is the INT instruction (INT is short for INTerrupt). Altogether 256 different sorts of interruption are allowed so it is necessary to specify which one is required. The interrupt instruction which allows operations such as reading a character from the keyboard, printing a character on the display screen and handling files on disk is number 21H. Other software interrupts are discussed in Chapter 21.

Thus, executing the instruction INT 21H gives you access to a whole range of input and output functions. Specifying which function you want involves putting a code number in register AH. Table 4.1 specifies the effects of more than a dozen of the functions available under software interrupt 21H. Others are listed in Chapter 19.

Table 4.1 Specifications of DOS functions available under software interrupt 21H.

<table>
<thead>
<tr>
<th>Function number</th>
<th>Description</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Keyboard input</td>
<td>Wait until a character is typed at the keyboard and then put the ASCII code for that character in register AL. Whatever is typed is also printed on the display screen wherever printing last left off. If CTRL-BRK is pressed, an automatic return is made to DOS control.</td>
</tr>
<tr>
<td>2</td>
<td>Output on the display screen</td>
<td>Print on the display screen the character whose ASCII code is contained in register DL. The character is displayed immediately after the last character displayed.</td>
</tr>
<tr>
<td>3</td>
<td>Asynchronous input</td>
<td>Waits for a character to be input via the asynchronous communications adapter card and places the received character in AL.</td>
</tr>
<tr>
<td>4</td>
<td>Asynchronous output</td>
<td>The character in DL is sent to the asynchronous communications adapter card.</td>
</tr>
<tr>
<td>5</td>
<td>Print character</td>
<td>The character in DL is sent to the printer.</td>
</tr>
<tr>
<td>6</td>
<td>Keyboard input/character display</td>
<td>If DL = 0FFH, the zero flag is set to 0 if a keyboard character is ready and the character is placed in AL. If no character is available the zero flag is set to 1. No waiting takes place. If DL &lt;&gt; 0FFH the contents of DL are interpreted as ASCII code and the appropriate character is displayed on the screen.</td>
</tr>
<tr>
<td>7</td>
<td>Keyboard input/no display</td>
<td>Waits for an input character from the keyboard and, when it arrives, places it in AL. The character is not automatically printed on the screen.</td>
</tr>
</tbody>
</table>
**Table 4.1 (cont.)**

<table>
<thead>
<tr>
<th>Function number</th>
<th>Description</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Keyboard input/no display</td>
<td>display screen. CTRL-BRK cannot be used to return to DOS.</td>
</tr>
<tr>
<td>9</td>
<td>Display string</td>
<td>As in function 7 except that normal CTRL-BRK service is provided.</td>
</tr>
<tr>
<td>A</td>
<td>Read keyboard string</td>
<td>Prints a whole series of characters stored in memory starting with the one in the address given in DX (relative to DS). Stop when a memory location containing the ASCII code for a $ sign is encountered. Does not print the $ sign.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A character string is read into store beginning at the address given in DS:DX. The first byte typed by the user specifies the maximum number of bytes which have been made available for the string and the second byte the length of the string. The actual characters of the string are stored in the third and following bytes of the designated group of locations until either the ENTER key is pressed or one less than the stated maximum number of locations has been filled. If the ENTER key is pressed, then the second byte of the storage pointed to by DS:DX is set to the number of characters read (excluding the ENTER). The last byte of the string is set to CR. Extra characters are ignored and the bell is rung if an overflow condition is about to occur.</td>
</tr>
<tr>
<td>B</td>
<td>Keyboard status</td>
<td>AL is set to 0FFH if a character is available from the keyboard. Otherwise AL is set to 0. A check for CTRL-BRK is made.</td>
</tr>
<tr>
<td>C</td>
<td>Keyboard buffer clear</td>
<td>The keyboard buffer within the keyboard is cleared, and the INT 21H function (only 1,6,7,8 and 0AH are allowed) specified in AL is performed.</td>
</tr>
<tr>
<td>4C</td>
<td>Return to DOS</td>
<td>Terminates program execution and returns control to DOS. An 'error level' may be set in AL: for normal use it is best to set AL to 0.</td>
</tr>
</tbody>
</table>

At this stage, the INT 21H functions in Table 4.1 which are of most interest to us are function 1 (keyboard input), function 2 (output on the display screen) and function 9 (display string).
The following sequence of instructions causes the microprocessor to wait for a character to be typed at the keyboard and then displays the character that was typed on the display screen and leaves the ASCII code for that character in AL:

```
MOV AH,1 ;specifies function number 1
INT 21H ;do it
```

Similarly, the following sequence causes the letter A (ASCII code 41H) to be printed on the display screen straight after the last character displayed there:

```
MOV DL,41H ;ASCII code for the letter A
MOV AH,2 ;specifies function number 2
INT 21H ;do it
```

The combination

```
MOV DX,offset relative to DS of starting address of message
MOV AH,9
INT 21H
```

will output the characters whose ASCII codes are stored in the locations with the given starting address (relative to the DS register) and following until the ASCII code for a $ sign is encountered. At this point the INTerrupt will cease and execution will continue as normal from the instruction after INT 21H.

Thus, given

```
DATA SEGMENT
.
.
.
DEC.31 MESSAGE DB 'HAPPY NEW YEARS'
.
.
DATA ENDS
```

then the sequence

```
MOV AX,DATA
MOV DS,AX
.
.
.
MOV DX,OFFSET DEC.31 MESSAGE
MOV AH,9
INT 21H
```

will cause HAPPY NEW YEAR to be output on the display screen starting wherever output last finished.

As a final example, the following sequence will turn the computer into a stuttering typewriter. It repeatedly waits for a lower-case letter a,b,c,...,z to be typed at the keyboard, outputs that character on the display screen and then outputs the upper-case equivalent of that letter A,B,C,...,Z. To do so it uses
the fact that the ASCII codes for lower-case letters are 20H greater than those for upper-case letters (see Appendix I):

```assembly
NEXT_CHARACTER: MOV AH,1 ;read a character into AL
                INT 21H ;(simultaneously
                SUB AL,20H ;displaying it on the screen)
                MOV DL,AL ;convert it
                MOV AH,2 ;display the converted character
                INT 21H
                JMP NEXT_CHARACTER
```

### 4.3.1 Printing new lines

When using an old-fashioned mechanical typewriter, starting a new line involves two operations; returning the carriage which holds the paper to the beginning of the line (carriage return) and moving the paper up one line (line feed). The display of text on a computer's screen has been designed to emulate this, so taking a new line involves 'printing' the ASCII codes for carriage return (in the table in Appendix I as CR) and line feed (in Appendix I as LF). As with a typewriter, the operations of carriage return and line feed can be done in either order, but there may be problems with some makes of printer if the carriage return is not done first.

The following sequence of instructions therefore outputs the message HI to the display screen followed by a new line:

```assembly
MOV AH,2 ;select output to screen function
MOV DL,48H ;ASCII for H
INT 21H ;display it

MOV DL,49H ;ASCII for I
INT 21H ;display it

MOV DL,0DH ;ASCII code for CR
INT 21H ;do it

MOV DL,0AH ;ASCII code for LF
INT 21H ;do it
```

---

**EXERCISE**

4.5 Write assembly language program fragments using DOS functions and incorporating all necessary pseudo-ops as follows:

(a) The user types any of the characters A,B,..,Z and it and the next character in the ASCII code definition (Appendix I) are displayed on the screen followed by a new line. Then the sequence of events is repeated. Thus, if the user types the letters S,T,R,A,N,G,E the resulting display should be:
(b) The user types one of the digits 1,2 or 3 and the corresponding message in:

1  - HELLO  
2  - GOODBYE 
3  - HAVE A NICE DAY

is displayed followed by a new line, and then the sequence of events is repeated. Thus, if the user types 1,3,2 the resulting display should be:

1HELLO
3HAVE A NICE DAY
2GOODBYE

(c) The user types two digits in the range 0,1,2,3, 4, 5 and then either the character Y or the character N is displayed followed by a new line. If two digits are equal then Y should be displayed, otherwise N should be displayed. Then the sequence of events is repeated. Thus, if the user types 8,8, 4, 5 the resulting display should be:

88Y
45N

4.4 Returning to DOS control

Through one particular function of INT 21H, DOS provides a mechanism whereby a program can be called into action in such a way that, when the program has been executed, control can once again be returned to the operating system. All that is required is that function 4CH of INT 21H is called into action with AL set to 0. The following example shows how this can be carried out in a program:

MOV AX, 4COOH
INT 21H

To emphasize: execution of these two instructions terminates the current program and returns control to DOS.
4.6 Write and test an assembly language program which asks its user to type in a name and then prints out:

    Roses are red
    Violets are blue
    How are you
    <the name typed in>?

and then returns to the operating system.

4.7 Rewrite the programs in Exercise 4.5 so that control returns to the operating system immediately after each task has been carried out rather than looping round again.

4.5 **Running assembly language programs**

We have already covered enough of the instruction set of the 8086 family to enable the reader to write some powerful programs, but we have yet to describe how to get programs into the computer and actually executed by the machine. When you have written your 8086-family assembly language program, three steps are involved in actually converting the program to machine code and then having it executed in such a way that you can check that it has performed according to plan:

- **Step 1** In order that the assembler knows where to store the program and any data, extra statements will have to be added to the bare 8086-family instructions – as first described in Section 4.4. Having added these extra assembler directives, run the assembler program with your prepared assembly language program as input. The output from the assembler will be a machine code version of your assembly language original.

- **Step 2** As with high-level language programs, a large machine code program is best developed in modules which can be independently tested and then linked together to provide the final desired result. The linking together is done by a program known as the **linker**. Because the assembler outputs machine code in a form suitable for the linker but not ready for immediate
execution, even a single program module must be submitted to
the linker for final conversion into executable form.

- **Step 3** Run the executable machine code form of your program under
  the control of a debugging program.

---

**Step 1 Preparing for the assembler**

**File names**

We shall use one of the most popular 8086-family assemblers: MASM. MASM
requires that your assembly language program is already in a disk file. It will be
assumed that the reader already knows how to organize this. Using MASM is
easiest when the file containing your assembly language program has the
second name ASM. Thus you should put programs for conversion into files called

```
EXAMPLE.ASM
CALC123.ASM
TEST.ASM
```

but not in files called

```
JUNK.TYP
WROT37.AB4
DONT.TRY
```

Should you want to, it is possible to override this convention. For details see
the MASM manual.

**Contents of the assembly language file**

The contents of the file containing the assembly language program to be
converted should be as follows:

```
DATA SEGMENT
  Insert here any necessary allocation
  and/or initialization of data storage
DATA ENDS

WORKING STORAGE SEGMENT STACK
  DW 100H DUP(?)
WORKING STORAGE ENDS

CODE SEGMENT
  ASSUME CS:CODE,DS:DATA,SS:WORKING STORAGE

PROG.START: Insert here the 8086-family instructions
  to perform the task which is to be carried out

CODE ENDS
END PROG.START
```
The **END** pseudo-op tells the assembler that it has now come to the end of the assembly language statements requiring conversion to machine code. Immediately after the **END** pseudo-op we must specify a label which identifies the first instruction in the program which is to be executed – in our case we shall conventionally use **PROG.START** for this purpose. (Of course, with just one program the assembler does not need to be told that the first instruction to be executed is the one at the beginning. But if a large program is built up out of several modules, each contained in a separate file, then it will no longer be obvious which module contains the first instruction to be executed.)

**Conversion to machine code**

We now assume that your assembly language program (with all the necessary pseudo-ops) is contained in a file of the right type. For the purposes of illustration we shall assume the file is called **FIRSTEX.ASM**. Remember that it is just the name of the file type which must be fixed; you can choose the file’s first name.

**Is everything on the right disk?**

It simplifies matters considerably if your assembly language program and the assembler program which is going to do the conversion are both on the same disk in drive A or in the same directory on a hard disk. For practical purposes it is a good idea to make yourself up a disk or directory with MASM, an editor program (EDIT, WORDSTAR or whatever you use), a linker (that supplied with MASM is called LINK and comes in a file called LINK.EXE), and a debugging program (see Chapter 5) all in the same directory before you start trying to work in assembly language. (Check that this does not break anyone’s copyright – one day you may be a software author yourself and value every cent of royalties).

From now on it will be assumed that this is the case. Thus, if you issue a **DIR** command to DOS, the response should at least include the following files together with an editor or word-processing program:

```
MASM.EXE
LINK.EXE
```

Invoking the assembler is easy. You type the name of the assembler after the operating system prompt, then a space, and then the first name of the file containing your assembly language program and a few extra characters at the end. For example, if your assembly language program is in a file in the root directory on drive A: then you type:

```
A>MASM FIRSTEX,...
```

Once the assembler has finished its job the operating system prompt will be displayed again and two new files will have been created because of the MASM options selected by the four commas. The contents of these files are described in Table 4.2.
Table 4.2  Contents of files created by MASM FIRSTEX, ... 

<table>
<thead>
<tr>
<th>Name of file</th>
<th>Contents of the file</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRSTEX.LST</td>
<td>A printout showing your assembly language program, its machine code equivalent in hexadecimal notation and a list of any errors detected in your assembly language.</td>
</tr>
<tr>
<td>FIRSTEX.OBJ</td>
<td>The machine code equivalent of your assembly language program, not quite in a state in which it can be executed.</td>
</tr>
</tbody>
</table>

**Step I reviewed**

Let us illustrate with a simple assembly language program which accepts two digits in the range 0 to 4 typed at the keyboard and displays the result of adding them up. It then returns to DOS. Thanks to the define pseudo-op

```
GOODBYE MESSAGE DB 'CALCULATION COMPLETE',ODH,OH,'$'
```

the message CALCULATION COMPLETE will be displayed after the result and output to the display screen will continue on a new line:

```
DATA SEGMENT
     GOODBYE MESSAGE DB 'CALCULATION COMPLETE',ODH,OH,'$'
DATA ENDS

WORKING STORAGE SEGMENT STACK
     DW 100H DUP(?)
WORKING STORAGE ENDS

CODE SEGMENT
ASSUME DS:DATA, SS:WORKING STORAGE, CS:CODE
; establish data segment addressability
PROG START: MOV AX, DATA
     MOV DS, AX
     ;read a digit
     MOV AH, 1
     INT 21H
     ;convert to a number
     SUB AL, 3OH
     ;save it in DL
     MOV DL, AL
     ;read second digit
     INT 21H
     ;convert it to a number
     SUB AL, 3OH
     ;add the two
     ADD DL, AL
     ;convert to a digit
     ADD DL, 3OH
     ;display the answer
     MOV AH, 2
     INT 21H
     ;and a message
```
MOV DX,OFFSET GOODBYE_MESSAGE
MOV AH,9
INT 21H
;return to DOS
MOV AX,4C00H
INT 21H

CODE ENDS
END PROG.START

Putting this complete text into a file of the right type (call it ADDER.ASM), we next invoke the assembler:

MASS ADDER,,,,

As the conversion proceeds, the messages shown in Figure 4.2 are printed on the display.

---

Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

47482 + 412225 Bytes symbol space free
0 Warning Errors
0 Severe Errors

---

When the conversion is complete, the A> prompt will be displayed. The file ADDER.LST now contains a fairly complete record of the assembler's work, the first page of which is shown in Figure 4.3.

This shows, among other things, that the machine code equivalent of SUB AL,30H is 2C30 in hexadecimal notation or 0010110000110000 in binary and that the first of these subtraction instructions is to be stored in locations 9H and 0AH relative to the program’s starting point and the second in locations 0FH and 10H.

Page 2 of the contents of the file ADDER.LST is not very meaningful to us at this stage but at least it says there are no errors. It is shown in Figure 4.4.

---

Step 2 Invoking the linker

To convert the ‘rough and ready’ machine code version the assembler has produced into its final form, just one more command is needed:

A>LINK FIRSTEX,,,,

The linker produces two more files, FIRSTEX.MAP and FIRSTEX.EXE. FIRSTEX.MAP contains a list of the starting and finishing addresses of the various program segments, relative to a zero initial location:
Microsoft (R) Macro Assembler Version 5.10 Page 1-1

DATA SEGMENT
1 0000               DATA SEGMENT
2 0000 20 20 20 43 41 4C GOODBYE MESSAGE DB 'CALCULATION COMPLETE',ODH,OAH,'$'
3 43 55 4C 41 54 49 4F 4E 20 43 4F 4D
4 50 4C 45 45 44 0D
5 0A 24
6 001A
7 0000    DATA ENDS
8 0000
9 0000 WORKING STORAGE SEGMENT STACK
10 0000 0100 [ DW 100H DUP(?)
11  ?? ??   WORKING STORAGE ENDS
12  14 0200 CODE SEGMENT
13  16 0000 ASSUME DS:DATA,SS:WORKING STORAGE,CS:CODE
14  16 0000 ;establish data segment addressability
15  0000 B8 ---- R PROG.START: MOV AX,DATA
16  0003 8E 08 MOV DS,AX
17  0005 B4 01 ;read a digit
18  0007 CD 21 MOV AH,1
19  0009 2C 30 ;convert to a number
20  000B 8A 0D SUB AL,30H
21  000D 8A 0D ;save it in DL
22  000F CD 21 MOV DL,AL
23  0011 8A 0D ;read second digit
24  0013 CD 21 INT 21H
25  0015 2C 30 ;convert it to a number
26  0017 8A 0D SUB AL,30H
27  0019 8A 0D ;add the two
28  001B CD 21 ADD DL,AL
29  001D 8A 0D ;convert to a digit
30  001F CD 21 ADD DL,30H
31  0021 B4 02 ;display the answer
32  0023 CD 21 MOV AH,2
33  0025 CD 21 INT 21H
34  0027 8A 0D ;and a message
35  0029 8A 0D MOV DX,OFFSET GOODBYE MESSAGE
36  002B CD 21 MOV AH,9
37  002D CD 21 INT 21H
38  002F B4 02 ;return to DOS
39  0031 CD 21 MOV AX,4COOH
40  0033 CD 21 INT 21H
41  0035 B4 02 CODE ENDS
42  0037 CD 21 END PROG.START
43  0039
44  003B 8A 0D
45  003D CD 21
46  003F CD 21
47  0041 CD 21

Figure 4.3
Page 1 of a record of the assembler's work.
Segments and Groups:

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Align</th>
<th>Combine</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>0026</td>
<td>PARA</td>
<td>NONE</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>0001A</td>
<td>PARA</td>
<td>NONE</td>
<td></td>
</tr>
<tr>
<td>WORKING STORAGE</td>
<td>0200</td>
<td>PARA</td>
<td>STACK</td>
<td></td>
</tr>
</tbody>
</table>

Symbols:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Attr</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOODBYE_MESSAGE</td>
<td>.L BYTE</td>
<td>0000</td>
<td>DATA</td>
</tr>
<tr>
<td>PROG_START</td>
<td>.L NEAR</td>
<td>0000</td>
<td>CODE</td>
</tr>
<tr>
<td>CPU</td>
<td>.TEXT</td>
<td>0101h</td>
<td></td>
</tr>
<tr>
<td>FILENAME</td>
<td>.TEXT</td>
<td>new</td>
<td></td>
</tr>
<tr>
<td>VERSION</td>
<td>.TEXT</td>
<td>510</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4
Page 2 of the record of the assembler's work.

Start Stop Length Name Class
000000H 00019H 0001AH DATA
00020H 0021FH 00200H STACK
00220H 00245H 00026H CODE

Program entry point at 0022:0000

In our present context, the actual locations used for a program and its data are determined by DOS or the debugger (see Chapters 3 and 5) but the relative displacements in the linker's .MAP file are preserved.

The file FIRSTEX.EXE contains the EXEcutable machine code version which can now be executed by typing:

FIRSTEX

Provided the reader has entered the program exactly as it was given above, it will then be possible to type 2 and 3 and have 5 displayed as the result of 2 + 3, and so on. However, if even a fairly minor typing error has been made, the program may 'hang' the computer and/or not work at all. Indeed, it is more usual when developing a program oneself that this is the stage where debugging begins.
Step 3 Running a program under debug control

Experience with other languages such as Pascal should indicate that programs of any reasonable length will not work first time – a little ‘fine tuning’ is invariably necessary. With assembly language programming, matters are actually a little worse in that a slight change in the contents of a register may be the only outcome of a program. Without a device that tells you what is going on in the registers and memory it will therefore be impossible to determine if your program has worked! Consequently DOS provides a special tool for debugging assembly language or machine code programs.

Taking a look at your program and/or data in memory, modifying either of them and actually running the program are all handled by debugger commands, the most important of which we shall describe in the next chapter.

EXERCISES

4.8 Verify your solutions to Exercises 4.5–4.7 by assembling and running them. (If anything has gone wrong, the computer may exhibit very strange behaviour and you may have to switch off and turn on again to bring things back under control. Chapter 5 will explain how to locate errors.)

4.9 Write, assemble and test-run assembly language programs as follows:

(a) The user types a digit and the program displays Y if twice that digit is equal to 18 or N otherwise, followed by a new line. Then the sequence of events is repeated two more times before returning to DOS.

(b) The user types two letters and if either of them is Y the program displays the word YES followed by a new line, otherwise it displays NO followed by a new line. Then the sequence of events is repeated three more times before returning to DOS.

4.10 Type the following error-ridden program into a file, being careful to type it exactly as it appears below:

DATA SEGMENT
  TEST_DATA DB 50,100,150,200
  ANSWERS: DW 34H DUP( )
DATA END

WORKING STORAGE SEGMENT STACK
  DW 100H DUP( )
WORKING STORAGE ENDS

CODE SEGMENT
ASSUME CS.CODE,DS.DATA,SS.STACK
establish data segment addressability
    MOV DS,DATA
    PROG.START
    MOV AX,4K
    MOV AH,AX
    SUB AX,2
    SUB AH,0123456H
    MOV BX,AX
    ADD AX,BL
    SUB 3,AX
    MOV BX,OFFSET DATA
    MOV AX,(BX)

    CODE END
    END PROG.START

Try to assemble it and see what error messages you get. Can you say what has caused each error?

4.6 Insisting upon exact addresses

If necessary we can insist that certain absolute locations in memory are used for program or data storage. Care must be taken when using absolute addresses in this way as it is possible to overwrite DOS or your debugger (or both) and have the system hang up on you.

We can load any segment register (indirectly) with any value we choose to specify a segment. Thus, to swap the contents of actual locations 9FF00H and 9FF01H we could load DS with 9FF0 via

    MOV AX,9FF0H
    MOV DS,AX

and then swap the contents of locations 0 and 1 relative to DS using MOV instructions in the ordinary way.

Within a given segment, allocation of storage will begin from offset zero by default. To change this we use the pseudo-op ORG. It tells MASM to continue the allocation of storage to instructions or data as if it had already reached the offset specified in the ORG command.

For example, the following little program swaps the contents of locations 200H and 201H (relative to the start of the data segment of our program). We reach these specific locations by using the ORG pseudo-op. Thus the combination

    ORG 200H
    FIRSTLOCATION DB ?

in a segment addressed by the DS register makes the variable FIRST_LOCATION refer to location 200H relative to the start of the data segment DATA.

DATA SEGMENT
    ORG 200H
    FIRSTLOCATION DB ?
    ORG 201H
SECOND_LOCATION DB ?
DATA ENDS

WORKING_STORAGE SEGMENT STACK
    DW 100H DUP(?)
WORKING_STORAGE ENDS

CODE SEGMENT
ASSUME DS:DATA, SS: WORKING_STORAGE, CS: CODE
; establish data segment addressability
PROG START: MOV AX, DATA
        MOV DS, AX
; swap the contents
        MOV AL, FIRST_LOCATION
        MOV BL, SECOND_LOCATION
        MOV FIRST_LOCATION, BL
        MOV SECOND_LOCATION, AL

CODE ENDS
END PROG START

SUMMARY

In this chapter we have explained what must be done to have an assembly language program converted to machine code. The next chapter covers debugging, the final stage necessary to obtain a correctly working program. We illustrated the initiation of the conversion process by means of a very simple adding-machine simulator which was largely constructed from DOS functions: routines accessible within the operating system’s own program by means of the INT instruction. These DOS functions are of crucial importance later in this book. In the context of utilizing them for our specific example we learnt how to arrange the storage of data items in an assembly language program and studied the mechanism by which the assembly language versions of 8086-family instructions must refer to such data items. One DOS function allows us to terminate execution of our program and return to DOS Control.

SELF-CHECK QUIZ

1. Using a single suitable DB or DW pseudo-op in each case, write an assembly language statement to initialize the contents of a group of locations, the first of which is to be referred to by MY_BYTES, as follows:

   (a) The decimal numbers 0, 1, 2, ..., 10 as unsigned 8-bit numbers.
   (b) The decimal numbers -10, -20, -30, ..., -100 as signed 8-bit words.
2. Draw a diagram showing the storage space allocated and its contents byte by byte, given the following assembly language statements:

(a) MY_BYTES DB 'NUMBER',16,-16H,6
DUP(0,?,2,?)
(b) MY_WORDS DW 6 DUP(0,1,2),?,-5,'WO','ORD',
347H

3. Given the following data segment:

```
DATA SEGMENT
D1 DW 1600
D2 DW 4800
D3 DW 1300
D5 DW 3900
D6 DW 7200
D7 DW 1300
D8 DW 5600
D9 DW 750
DA DW 1500
DATA ENDS
```
make a trace of the execution of the following program fragment:

```
MOV AX,DATA
MOV DS,AX
MOV BX,2
MOV AX,[BX]
MOV CX,D6
SUB AX,CX
ADD BX,6
MOV AX,[BX]
ADD AX,D9
ADD BX,8
MOV AX,[BX]
SUB AX,D9
```

4. Given

```
ITEMA DB 26H
ITEMB DW -123
ITEMC DB 'Welcome'
```
which of the following would be legal assembly language statements:

(a) MOV AX,ITEMA
(b) MOV ITEMB,AL
(c) MOV AX,ITEMC
(d) MOV ITEMC,AL
(e) MOV ITEMC,AX
(f) ADD ITEMA,AL
(g) ADD ITEMA,AX

5. Given

```
CCC DW 2BAFH
DDD DW A441H
BBB DW 289AH
```
what will be the contents of AX after execution of:

```
MOV AX,CCC
ADD AX,BBB
SUB DDD,AX
```

6. Given the following data segment:

```
DATA SEGMENT
DAT1 DW 16 DUP(3)
DAT2 DW 11 DUP(21)
DAT3 DB 13,14,17,19
DAT4 DB 'Welcome'
DAT5 DW 1300,5600,750,1500
DATA ENDS
```
make a trace of the execution of the following program fragment:

```
MOV AX,DATA
MOV DS,AX
MOV AX,OFFSET DAT1
MOV BX,OFFSET DAT5
MOV CX,[BX]
ADD CX,DAT5
ADD BX,OFFSET DAT2
MOV BX,OFFSET DAT3
MOV AL,[BX]
```

7. Write a sequence of assembly language instructions and corresponding DB pseudo-op to display the message california welcomes careful drivers on the display screen on each of four separate lines.

8. Write a sequence of assembly language instructions and corresponding DB pseudo-op which would allow the user to type in a 20 letter message (possibly including spaces) and store the message in a group of memory locations the first of which is to be referred to by USER__MESSAGE.
9. What is wrong with the following attempt at a complete assembly language program which is intended to add together the two stored digits and print the answer on the display, and then return to DOS control:

```
DATA SEGMENT
  M1 DB 3
  M3 DB 2
ENDS

CODE

ASSUME CS,DS:DATA, CODE
  MOV AL,M1
  ADD AL,M3
  ADD AL,30
  MOV AH,2
  INT 21H

ENDS
```

10. Write a sequence of assembly language instructions to add the contents of absolute locations 23579H and 40AAFH.