than one sequence of maximum length, set DX to the start address of the first such sequence.) Thus if the text was:

a mathematician confided
the mobius band is one sided
if you cut it in half
you'll get quite a laugh
for it stays in one piece though divided

then the longest sequence of letters in alphabetical order contains three letters and DX should be set to point to the 'b' of 'mobius'.

(b) Following inspiration from real-estate advertisements, secret agent Clotsky decided to code all messages by omitting all vowels (a, e, i, o, and u) except those which occurred at the beginning of a word. Thus:

come quickly before all is lost
would be coded as

cm qckly bfr all is lst

Write an assembly language program to carry out this coding procedure. Assume that the text to be coded consists of 255 characters or less and is stored by means of a DB statement:

MESSAGE_TO_BE_SENT DB ' . . '

The coded message is to overwrite the original text stored in memory.

8.2 Subroutines

In Pascal, procedures make it possible to break a large program down into smaller pieces so that each piece can be shown to work correctly without reference to any of the others. In this way the final program is built up of lots of trusty bricks and is easier to debug since, if there is an error, it can only be caused by the interlinking of the bricks. In 8086-family assembly language the nearest equivalent to Pascal's procedures are subroutines.

Making a sequence of assembly language instructions into a subroutine is easy - we simply label the first instruction in the sequence as if we were going to make it the object of a jump instruction, and add the 8086-family instruction RET to the end of the sequence. Consider the example below, which contains a program fragment which leaves the sum of the unsigned 16-bit numbers in AX, BX, CX and DX (assuming this sum can be represented in 16-bit form) in SI, a 16-bit register, new to us, the special purpose of which will be introduced in Chapter 10.
MOV SI,0
ADD SI,AX
ADD SI,BX
ADD SI,CX
ADD SI,DX

Let us make this into a subroutine called REGISTER_SUMMATION:

REGISTER_SUMMATION: MOV SI,0
ADD SI,AX
ADD SI,BX
ADD SI,CX
ADD SI,DX
RET

Having done so, we can call the subroutine into action via the 8086-family CALL instruction, thus: CALL REGISTER_SUMMATION. This causes the microprocessor to begin executing the instruction sequence starting with the label REGISTER_SUMMATION until a RET instruction is encountered, at which point execution RETurns to the instruction next in sequence after the CALL instruction itself (see Figure 8.1).

In the program fragment below, comments show the sequence of values taken by SI during the execution of the given program fragment:

MOV AX,0
MOV BX,0
MOV CX,0
MOV DX,1
CALL REGISTER_SUMMATION ;SI has now been set to 1
MOV AX,2
CALL REGISTER_SUMMATION ;SI has now been set to 3
MOV BX,3
MOV CX,4
CALL REGISTER_SUMMATION ;SI has now been set to QAH

---

**Figure 8.1**
Execution of a subroutine.
EXERCISES

8.3 Make a hand trace of the execution of the following program fragment, assuming that the subroutine PRINT_ONSCREEN prints on the display screen the character whose ASCII code is in register AL and does not change any register values. Give a clear indication of what appears on the display screen.

```
MOV CX,5H
MOV AX,40H
NEXT_STEP:
INC AX
CALL PRINT_ONSCREEN
CALL CODE_AND_PRINT
DEC CX
JNZ NEXT_STEP
HLT
```

```
CODE_AND_PRINT:
MOV BX,20H
ADD BX,AX
MOV DX,AX ;save the value of AX
MOV AX,BX
CALL PRINT_ONSCREEN
MOV AX,DX ;restore the value to AX
RET
```

```
PRINT_ONSCREEN:
.
.
.
RET
```

Notice that, as with procedures in Pascal, one subroutine may call another.

8.4 Write an assembly language subroutine called CALC_FRANC_EQUIV which, given an unsigned 16-bit number in AX representing a value in dollars, will leave in AX an unsigned 16-bit number representing the nearest whole number of French francs equivalent to this amount using the hypothetical conversion rate of $1 = 8FF.

8.3 The stack and its role in the subroutine mechanism

As stated in Chapter 3, a stack is a group of locations in memory which the programmer reserves for the (temporary) storage of important items of data. The mechanism by which subroutines are implemented in the 8086 family uses a stack. Briefly, when a subroutine is called up, the address of the instruction
following the call is saved on a stack until the `RET` instruction in the body of the subroutine is encountered. Then execution continues from the instruction following the call by retrieving the return address from that stack. Before examining the mechanism in detail, a thorough discussion of 8086-family stack implementation is necessary, and it is to that which we now turn.

### 8.3.1 The stack

The method by which the 8086 family provides a stack facility is a common one. Given our conventional format for an assembly language program, MASM, LINK and DOS arrange that the stack begins just before the start of our program and grows backwards in memory as demand warrants, as shown in Figure 8.2.

Copies of the contents of registers can be stored on and retrieved from a stack thanks to the `PUSH` and `POP` instructions which have the general forms:

```
PUSH <16-bit register name>
```

(for example `PUSH AX`) and

```
POP <16-bit register name>
```

(for example `POP DX`). `PUSH` stores the copy in memory, and `POP` loads it back again. For the flags register (a 16-bit register in its own right) special forms of the `PUSH` and `POP` instructions must be used: `PUSHF` stores and `POPF` retrieves.

The top of a stack is where items are added and removed – compare with the dinner plate dispenser model of Chapter 3. In order that `PUSH` and `POP` can operate successfully it is necessary only to: specify how big a memory stack is required; set the stack segment register SS to the segment address of the start of the stack; and set the 16-bit Stack Pointer (SP) register to the offset address of the top of the stack. Both SS and SP are initialized automatically by MASM and LINK thanks to the pseudo-ops:

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

which instruct the linker to set SS to a value compatible with the memory allocation plan of Figure 8.2 and to initialize SP to the appropriate top of stack value (2 * 100H = 200H, relative to SS). Of course, reserving more words of store for the stack – up to 64 Kbytes is allowed – will alter the value to which SP is initialized.

Let us suppose that the stack pointer register, SP, has been initialized to 0200H (relative to the SS register) and consider the effect of the instruction `PUSH BX` in detail. Suppose further that register BX contains 1A9EH. Then the action of `PUSH BX` is as follows:

1. The SP register is decremented by 1.
2. The high-order byte of BX is stored in the memory location addressed by SP (relative to the SS register).
The stack and its role in the subroutine mechanism

Figure 8.2
Typical memory allocation when using a stack.

(3) The SP register is decremented by 1.
(4) The low-order byte of BX is stored in the memory location now addressed by SP (relative to the SS register).

(Notice that SP is thus left pointing to the last element stored in the stack, that is, to the top of the stack.)

<table>
<thead>
<tr>
<th>Before</th>
<th>SP 0200</th>
<th>After</th>
<th>SP 01FE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX 1A9E</td>
<td>01FE ??</td>
<td>BX 1A9E</td>
<td>01FE 9E</td>
</tr>
<tr>
<td>01FF ??</td>
<td>01FF 1A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0200 ??</td>
<td>0200 ??</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On the other hand, the action of an instruction such as **POP DX** releases memory from the stack, in that the SP register is incremented. Its action is as follows:

(1) Copy the byte stored at the address given in the SP register (relative to SS) into DL.
(2) Increment the SP register by 1.
(3) Copy the byte now pointed to by the SP register (relative to SS) into DH.
(4) Increment the SP register by 1.

Assuming that the SP register currently contains 009EH, and that locations 009EH and 009FH relative to the SS register contain 0A2H and 0B3H respectively, then the effect of **POP DX** will be to put 0B3A2H into DX and to change the contents of the SP register to 00A0H.
8.3.2 PUSH and POP in daily use

Once everything is initialized, PUSH and POP can be used freely wherever temporary storage is required. It is necessary to remember just two points:

1. It is essential that any subroutine should leave the stack in the same state as when that subroutine started using the stack. Otherwise, strange things may happen when your program is executed since, for example, a value which you have PUSHed but not POPped could be taken as the return address for that subroutine – see Section 8.6.

2. The last item put on the stack is the first off. Thus:

PUSH AX ; copy AX onto stack
PUSH BX ; copy BX onto stack
POP AX ; copy the item currently on the top
      ; of the stack into register AX and
      ; then remove that item from the stack
POP BX ; copy what is on the top of the stack
      ; into register BX and then remove
      ; that item from the top of the stack

Effectively swaps the contents of registers AX and BX.

EXERCISES

8.5 Make a detailed trace of the execution of the following program fragment, giving the contents of all the registers and memory locations involved. Assume that before execution begins the SS register contains 1000H, and the SP register 0200H, and that the contents of AX, BX, CX and DX are (respectively) 0100H, 123AH, 0FE1BH and 0A981H.

PUSH AX
PUSH BX
PUSH CX
POP DX
POP CX
PUSH DX
POP AX
POP BX

8.6 Registers AX, BX and CX all contain the ASCII codes for two letters as follows:

<table>
<thead>
<tr>
<th>AX</th>
<th>BX</th>
<th>CX</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>BH</td>
<td>CH</td>
</tr>
<tr>
<td>AL</td>
<td>BL</td>
<td>CL</td>
</tr>
<tr>
<td>S</td>
<td>R</td>
<td>M</td>
</tr>
<tr>
<td>O</td>
<td>T</td>
<td>E</td>
</tr>
</tbody>
</table>
Using `PUSH` and `POP`, write a subroutine called `REARRANGE` which leaves AX, BX and CX containing the letters in alphabetical order:

<table>
<thead>
<tr>
<th>AX</th>
<th>BX</th>
<th>CX</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>BH</td>
<td>CH</td>
</tr>
<tr>
<td>AL</td>
<td>BL</td>
<td>CL</td>
</tr>
<tr>
<td>E</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>M</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

### 8.4 A complete program using subroutines

To make matters absolutely clear, this section gives a complete assembly program which uses subroutines and which prints a small message on the computer's display screen. The actual printing on the screen is done — as before — by one of the DOS functions, but this time we incorporate it into a subroutine.

```
PRINTCHAR: MOV AH,02H ;function number 2
           INT 21H     ;of INTerrupt 21H
           RET
```

The program given below is a complete program which uses the subroutine `PRINTCHAR` to print the three-letter message `IBM` on the screen. (It uses the fact that with `PRINTCHAR`, each time something is printed, printing continues from wherever it finished last time.) Type it into your machine and verify that it works. Try to amend it so that the message printed out is `ABC` and check your work by re-assembling your amended version.

In the program we have used a new MASM pseudo-op, `COMMENT`. `COMMENT` lets you enter comments about your program without having to precede each comment line with a semicolon. Instead, the actual comment is enclosed by the first non-blank character after the `COMMENT` pseudo-op and the next occurrence of that character.

```
COMMENT *This program prints a three-letter message
       on the PC screen. The message can be changed
       by altering the values of FIRSTLETTER,
       SECONDLETTER and THIRDLETTER.*

FIRSTLETTER EQU 49H ;49H is the ASCII code for the letter I
SECONDLETTER EQU 42H ;42H is the ASCII code for the letter B
THIRDLETTER EQU 4DH ;4DH is the ASCII code for the letter M
PRINTFUN EQU 2H
CHARFNS EQU 21H
```
DATA SEGMENT
DATA ENDS

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

CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:WORKING STORAGE
PROG.START: MOV DL, FIRSTLETTER
CALL PRINTCHR
MOV DL, SECONDLETTER
CALL PRINTCHR
MOV DL, THIRDLETTER
CALL PRINTCHR
; return to DOS
MOV AX, 4C00H
INT 21H

PRINTCHR: MOV AH, PRINTFUN
INT CHARFN
RET

CODE ENDS
END PROG.START

---

EXERCISE

8.7 Type the above program into your computer and verify that it works. Inspect the .LST file produced by MASM to see the effect of EQU. Amend the program so that the message printed is:

DOS
RULES
OK !!

---

8.5 CALL and RET – the mechanism of the basic (intra-segment) forms

As pointed out in Section 3.3 it is from the IP register (relative to the contents of the CS register) that the 8086 family gets the address of the next instruction to be obeyed. Generally speaking, the programmer does not access the IP register directly. Before execution of a program the operating system initializes CS:IP to the first of the locations containing that program. After execution of any particular instruction, IP is automatically incremented to point to the
### CALL and RET – the mechanism of the basic (intra-segment) forms

<table>
<thead>
<tr>
<th>Before</th>
<th>CALL 0001</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>010DH</td>
<td>0111H</td>
</tr>
<tr>
<td>SP</td>
<td>01E4H</td>
<td>01E2H</td>
</tr>
<tr>
<td>01E1H</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>01E2H</td>
<td>??</td>
<td>10H</td>
</tr>
<tr>
<td>01E3H</td>
<td>??</td>
<td>01H</td>
</tr>
<tr>
<td>01E4H</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RET</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>0125H</td>
<td>0110H</td>
</tr>
<tr>
<td>SP</td>
<td>01E2H</td>
<td>01E4H</td>
</tr>
<tr>
<td>01E1H</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>01E2H</td>
<td>10H</td>
<td>10H</td>
</tr>
<tr>
<td>01E3H</td>
<td>01H</td>
<td>01H</td>
</tr>
<tr>
<td>01E4H</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

**Figure 8.3**
The effect of CALL and RET on the contents of the IP register.

The address of the next instruction. CALL and RET disturb this mechanism by changing the contents of the IP register (and sometimes the CS register).

As a specific example let us consider the following extract from the listing of an assembler:

```
  ...
  ...
  010B 8B C1 MOV AX,CX
  010D E8 01 00 CALL A.SUB
  0110 F4  HLT
  0111 F7 E3  A.SUB: MUL BX
  0113 8B DD  MOV DX,AX
  ...
  ...
  0124 C3  RET
```

By the time these instructions come to be executed the assembler will have replaced the label A.SUB in CALL A.SUB with one of two things. If the subroutine with first instruction labeled A.SUB is stored within 0FFFFH locations of the CALL instruction, then the label A.SUB is replaced by a 16-bit unsigned number. This number specifies what will need to be added to the contents of the IP register after execution of the CALL has begun in order for CS:IP to point to the instruction labeled A.SUB. Thus, in this case, the label in CALL A.SUB will be replaced by 0001. This is shown in Figure 8.3.

Notice from the assembler listing above that the instruction CALL A.SUB
(really, CALL 0001) has the machine code equivalent E8 01 00. When execution of the CALL begins, IP will – as normal – be incremented to point to the next instruction in sequence (in this case it is a HLT instruction). But during execution of CALL 0001, the 16-bit unsigned number following the code for the CALL instruction itself (0E8H) is added to the current contents of the IP register. Since the next instruction to be obeyed after the CALL will be taken from the location specified by the IP register (relative to the contents of CS), the next instruction executed will indeed be that labeled A.SUB. But this is not the only effect of CALL. In order that a return to the main program can be effected once a RET instruction is encountered, CALL also pushes the address of the instruction which follows it on to the stack. Execution of the RET instruction simply entails popping the top 16-bit element of the stack into the IP register. Provided as many PUSHes as POPs have occurred within the body of the subroutine, the word at the top of the stack will be the address of the instruction following the CALL.

CALL and RET instructions of this type are known as intra-segment (or within segment) instructions, since the body of the subroutine is in the same segment (that is, in the same collection of 10000H memory locations) as the CALL.

To summarize the effect of CALL and RET intra-segment instructions:

- **CALL <label>** The <label> following the instruction is replaced by an unsigned 16-bit number which is the difference between the address of the first location after those containing the CALL instruction and the address of the first location containing the body of the subroutine (both relative to CS).

  When the CALL instruction is executed, the unsigned 16-bit number is added to the contents of the IP register and the address of the instruction following the CALL instruction is pushed onto the stack.

- **RET** When the RET instruction is executed, the top of the stack is popped into the IP register.

### 8.6 The inter-segment subroutine mechanism

If the instruction following the CALL and the body of the subroutine called are more than 0FFFFH locations apart, the inter-segment form of both CALL and RET must be used. Fortunately MASM keeps track of which form is necessary so we never actually have to count locations ourselves. The format of the CALL instruction does not change but CALL <label>, where the <label> refers to a subroutine in another segment, will be translated by MASM into the form CALL <address> where the <address> is a 4-byte quantity representing the address of the body of the subroutine. Of these four bytes, the first two are the offset
of the subroutine from the CS register and the second two the setting of the CS register for the segment containing the CALL instruction. When the inter-segment CALL instruction is executed the full (4-byte) address of the instruction following the CALL is placed on to the stack, CS value first. Then the 4-byte address in the CALL instruction itself is transferred to the IP and CS registers so that the next instruction to be executed will be the first in the appropriate subroutine.

To return from an inter-segment subroutine a different form of the RET instruction is used. On execution it pops the top two stack bytes into IP and the next two bytes into the CS register.

To know in advance whether a given CALL is to be converted into the machine code form of an inter-segment CALL or an intra-segment CALL, MASM insists that the programmer who wants to work in several segments must cloak each part of the program in such a way that what is near and what is far is obvious to the assembler. Unless explicitly told otherwise, it assumes that all CALL and RET instructions are intra-segment.

Inter-segment CALLs and RETs are achieved thanks to the idea of a procedure, which is simply a subroutine labeled in a careful way and given one of the attributes near or far. Thus:

(1) MY_SUBROUTINE: MOV AX,4
    INC BX
    .
    .
    .
    RET

(2) MY_SUBROUTINE PROC NEAR
    MOV AX,4
    INC BX
    .
    .
    .
    RET
MY_SUBROUTINE ENDP

(3) MY_SUBROUTINE PROC FAR
    MOV AX,4
    INC BX
    .
    .
    .
    RET
MY_SUBROUTINE ENDP

All result in exactly the same machine code, except that the RET instruction in (1) and (2) will be coded as intra-segment and that in (3) as inter-segment (and likewise CALL MY_SUBROUTINE in the first two cases will be translated into an intra-segment CALL whereas CALL MY_SUBROUTINE in case (3) will be inter-segment).

Both popular debuggers, DEBUG and Code View, use different names for the CALL and RET instructions if inter-segment operation is involved. Thus CALLF is used to CALL a Far away subroutine and RETF to RETurn from a Far away subroutine.
EXERCISES

8.8 Rewrite, assemble and run our complete program example above containing subroutines so that it uses procedures (a) in such a way that all CALL and RET instructions are assembled as near and (b) so that all CALL and RET instructions are assembled as far. Compare the .LST files thus obtained.

8.9 Write and test an assembly language program which draws a 4 * 4 chess board on the screen with the ‘black’ squares made up from a 4 * 4 grid of letter Xs:

```
XXXX  XXXX
XXXX  XXXX
XXXX  XXXX
XXXX  XXXX
```

8.7 Subroutines with parameters

The PRINTCHAR subroutine of Section 8.4 was an example of the technique of using registers to pass a parameter across to a subroutine: register AL contained the ASCII code for the character to be printed by PRINTCHAR. In this way, general-purpose registers can be used to hold the parameters themselves or they may hold the addresses where parameters are located in memory. The main advantages of such an approach are that it is relatively easy to pass a small number of parameters and that, when a register is used to pass the initial (or base) address of a group of memory locations, the calling subroutine does not need its own copy of the data.